A Windows program for airfoil design using B-splines

Matthew MacLean

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
A Windows Program for Airfoil Design

Using B-Splines

by

Matthew G. MacLean

A Thesis Submitted in Partial Fulfillment of the Requirement for

the

MASTER OF SCIENCE

IN

MECHANICAL ENGINEERING

Approved by:

Dr. P. N. Venkataraman
Department of Mechanical Engineering

Dr. K. B. Kochersberger
Department of Mechanical Engineering

Dr. A. D. Ogut
Department of Mechanical Engineering

Dr. C. Haines
Department of Mechanical Engineering

(thesis advisor)

(department head)

DEPARTMENT OF MECHANICAL ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
AUGUST 1999
Permission Granted

I, Matthew G. MacLean, hereby grant permission to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial profit.

Signature of author: 

Date: 2/1/9
Special Thanks

I would like to thank my advisors for their academic inspiration:
Dr. Venkataraman, Dr. Kochersberger, Dr. Ogut, and Dr. Haines.

I would also like to thank my parents for their financial and moral support throughout my college career.

I would like to thank Phi Delta Theta International Fraternity for the inspiration and chivalrous character it has allowed me to develop and build in myself.

I would like to thank a few of my closest friends who have taken an interest in my progress on this thesis, and their willingness to help in any way they could:

Finally, I would like to thank all of my friends that I have known throughout my college career (and will continue to know) – the people who have made it easier to do everything I have done at RIT:
For all those on the previous page that this is dedicated to:

"Time is a companion that goes with us on the journey, and reminds us to cherish every moment, because they'll never come again. What we leave behind is not as important as how we've lived."

- Patrick Stewart, Star Trek: Generations.
Abstract

The objective of this thesis has been threefold. First, the formulation of splines has been studied and their development into a computer algorithm has been implemented. Splines represent a powerful concept in computer modeling and geometric representation, for, as parametric curves, they provide a compact way to store the information defining a curve or surface. B-splines have been exclusively used in this thesis, although other types of splines exist.

The second goal of this thesis was to learn and utilize C++ as a programming tool in the demonstration of B-spline techniques. C++ was chosen because it is object-oriented, and because it is the chosen language of the Microsoft Windows PC platform. Many languages are object-oriented, but C++ was chosen to make use of its libraries to build standard Windows interfaces and objects.

The third piece of this thesis is an effort to explore the fundamentals of inter-language communications. Many old scientific codes are already written in older languages like FORTRAN, so it is advantageous to re-use those codes where possible. Digital Visual FORTRAN, a module of the Microsoft Visual Studio, has provided a powerful tool in their integration of multiple programming languages for Windows applications. Using Visual Studio, it is possible to re-use existing FORTRAN code and envelop it in a C interface using a dynamic link library (DLL) file.

This thesis uses a C++ application for defining any typical airfoil using B-splines. The software package calls XFOIL, a code written in FORTRAN to evaluate the aerodynamic characteristics of those airfoils. Further, those characteristics have been compared to those of the original geometry to evaluate the interpolation process used by the splines.
# Table of Contents

1.0 Introduction to B-spline ................................................................. 1  
  1.1 B-splines ......................................................................................... 1  
  1.2 Matrix Form of Splines ................................................................. 10  
  1.3 Curve Approximation with B-splines .......................................... 11  

2.0 Selected Topics of C++ ................................................................. 13  
  2.1 Structures ....................................................................................... 14  
  2.2 Classes .......................................................................................... 15  
  2.3 Inheritance .................................................................................... 21  
  2.4 Polymorphism ............................................................................... 23  
  2.5 Pointers and Arrays ..................................................................... 24  
  2.6 Multi-dimensional Arrays .............................................................. 29  

3.0 MFC programming in C++ ............................................................ 38  
  3.1 The CWinApp and CMainWnd Objects ........................................... 40  
  3.2 Message Maps ............................................................................... 42  
  3.3 Resources ..................................................................................... 46  
  3.4 Wizards ......................................................................................... 52  

4.0 Advanced Coding and Modularization ......................................... 54  

5.0 A Description of the Programming Algorithms used ...................... 62  

6.0 Results .......................................................................................... 67  

7.0 Conclusion/Recommendations ...................................................... 81  
  Reference List ..................................................................................... 83

Main program source files .............................................................. A-1  
Data reading source files ............................................................... B-1
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A Typical Spline Curve</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Plot of Example 1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Plot of Example 3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>Memory Mapping Configuration</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Multidimensional Array Mapping</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>2-Dimensional Array Mapping</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>1-Dimensional Array Mapping</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>A Typical Dialog Box</td>
<td>47</td>
</tr>
<tr>
<td>9</td>
<td>Data Read Utility Program(main window)</td>
<td>67</td>
</tr>
<tr>
<td>10</td>
<td>Standard Windows 95/NT Open File Dialog Box</td>
<td>68</td>
</tr>
<tr>
<td>11</td>
<td>Airfoil Spline Program</td>
<td>68</td>
</tr>
<tr>
<td>12</td>
<td>New Airfoil Dialog Box</td>
<td>69</td>
</tr>
<tr>
<td>13</td>
<td>Spline Airfoil Program with N0012 Airfoil Active</td>
<td>70</td>
</tr>
<tr>
<td>14</td>
<td>Typical Polygon Point Box</td>
<td>71</td>
</tr>
<tr>
<td>15</td>
<td>N0012 Airfoil Comparison</td>
<td>73</td>
</tr>
<tr>
<td>16</td>
<td>N2411 Airfoil Comparison</td>
<td>73</td>
</tr>
<tr>
<td>17</td>
<td>Selig 1223 Airfoil Comparison</td>
<td>74</td>
</tr>
<tr>
<td>18</td>
<td>Boeing 707 0.54 Span Airfoil Comparison</td>
<td>74</td>
</tr>
<tr>
<td>19</td>
<td>RAE(NPL) 5215 Airfoil Comparison</td>
<td>75</td>
</tr>
<tr>
<td>20</td>
<td>cl vs. $\alpha$ for N2411 Airfoil</td>
<td>79</td>
</tr>
<tr>
<td>21</td>
<td>cd vs. $\alpha$ for N2411 Airfoil</td>
<td>79</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NACA 0012 Airfoil</td>
<td>76</td>
</tr>
<tr>
<td>2</td>
<td>NACA 2411 Airfoil</td>
<td>76</td>
</tr>
<tr>
<td>3</td>
<td>Selig 1223 Airfoil</td>
<td>76</td>
</tr>
<tr>
<td>4</td>
<td>Boeing 707 Airfoil</td>
<td>76</td>
</tr>
<tr>
<td>5</td>
<td>RAE(NPL) 5215 Airfoil</td>
<td>76</td>
</tr>
<tr>
<td>6</td>
<td>NACA 0012 Airfoil</td>
<td>77</td>
</tr>
<tr>
<td>7</td>
<td>NACA 2411 Airfoil</td>
<td>77</td>
</tr>
<tr>
<td>8</td>
<td>Selig 1223 Airfoil</td>
<td>77</td>
</tr>
<tr>
<td>9</td>
<td>Boeing 707 Airfoil</td>
<td>77</td>
</tr>
<tr>
<td>10</td>
<td>RAE(NPL) 5215 Airfoil</td>
<td>77</td>
</tr>
<tr>
<td>11</td>
<td>RAE(NPL) 5215 Airfoil</td>
<td>78</td>
</tr>
<tr>
<td>12</td>
<td>Bezier Curve Tests</td>
<td>80</td>
</tr>
</tbody>
</table>
1.0 Introduction to B-splines

The word spline is a generic term used to describe a vast family of mathematical curves with a variety of properties, behaviors, and characteristics. Splines are piecewise polynomials, meaning that they are defined differently at various points along their length. This thesis uses generic B-type splines.

Although not specifically used in this thesis, one important type of B-splines is known as the non-uniform rational B-spline (NURBS). NURBS are used by many commercial software programs, especially for defining CAD geometry. They are also often utilizes by three-dimensional modeling and drawing programs. NURBS are also important in the field of multi-disciplinary design; they possess the compactness of all splines, and their implementation gives them flexibility while remaining stable and simple to control. This thesis has implemented generic B-splines to investigate their potential for similar applications.

A description of B-splines and their creation and implementation is provided in this section. A special subset of B-splines called Bezier curves will also be introduced. Several examples of splines will be given. After the method of calculating them is explained, the analytical representation of curve fitting B-splines is developed. This method does not employ an approximate fit but rather interpolates from the data directly. The Airfoil Spline Application program, the source code provided in Appendix A, makes use of this method in creating airfoils for submission to XFOIL.

1.1 B-splines

B-splines are parametric curves governed by a set of equations known as the de Boor recursion formulas. The curve is actually in terms of a parametric variable \( t \), which is converted to the classic \( x,y,z \) Cartesian coordinates through a set of defining polygon points. The number of dimensions of the curve is directly determined by the number of dimensions of the polygon points since each dimension is calculated independently. This set of defining points are referred to as such because they can all be connected to form a polygon with as many sides as there are points.
An interesting property of B-splines is that the curve will generally follow the polygon points in space. It should be noted here that the defining polygon points do not necessary have to follow a particular ordered sequence in space (for example, in order of increasing x coordinate); these points can lie in any order desired, and the curve will follow that spatial pattern. The curve can even intersect or cross over itself. Figure 1 shows a typical B-spline with the defining polygon outlined.

![Figure 1. A Typical Spline Curve](image)

The actual formulas themselves are given as follows, taken from Reference [1]:

$$P(t) = \sum_{i=1}^{n+1} B_i * N_{i,k}(t)$$  \hspace{1cm} (1)

where $P(t) = [x(t) \ y(t) \ z(t)]$ and $B = [B_x \ B_y \ B_z]$. 

page - 2
\[ N_{i,j}(t) = \begin{cases} 1 & \text{if } x_i \leq t < x_{i+1} \\ 0 & \text{otherwise} \end{cases} \quad (2) \]

\[ N_{i,j}(t) = \frac{(t-x_i) N_{i,j-1} + (x_{i+j} - t) N_{i+1,j-1}}{(x_{i+j} - x_j)} \quad (3) \]

In equations (1) - (3), there are several symbols that need to be defined. First, in equation (1), the \( B \) term represents the polygon points. These points are designated as \( B1 \) through \( B5 \) in Figure 1, and are known as polygon points, vertex points, polygon vertices, control points, etc. The data point of the curve is just a summation of these polygon points times a shape factor associated with each one. Since the data point, \( P(t) \), and the polygon points are both vectors, the data point can be calculated one dimension at a time. The summation of \( B_x \)'s times their shape factors yields the resulting \( x \)-value of the data points.

The \( x \)-terms in equations (2) and (3) are the terms of a knot vector. They are not related to the Cartesian coordinate \( x \). This knot vector adds an interesting twist to this type of mathematics, as the knot terms determine how far a polygon vertex's influence extends through the curve. These knot vector terms appear in both equation (2) and (3), which determines the value of the shape factors. This, in turn, controls the influence a particular polygon point has at different data locations along the spline. A spline generally follows its polygon shape because of this – points close to a particular data location have more control and can pull the data toward it. The only requirement for knot vector terms is that each successive term must be greater than or equal to the previous term.

The lower case \( n \) is related to the number of control points that determine the curve. There must always be one more than \( n \) polygon points to control the spline. For instance, in the case of Figure 1, there are five control points (\( B1 \) through \( B5 \), so \( n \) equals 4 for that particular spline. Additionally, \( k \) is called the curve order. The third equation (3) must be iterated successively until the \( k \) level is found. This directly defines the order of the polynomial shape factors, which are just parametric polynomial curves of the variable \( t \). As with all polynomials, a higher order shape factor allows for more complex curvatures and fitting. It works out that the knot vector must always have \( n+k+1 \) terms in it to satisfy the formulations in (1) through (3).
It seems appropriate to help clarify this process of curve generation with a simple example carried out by hand to generate a B-spline.

**Example #1:** generation of formulas for a B-spline with specified parameters.

**specified** -
- \( n = 3 \) (control points)
- \( k = 4 \) (curve order)
- \( \mathbf{x} = [0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1] \) (knot vector - \( n+k+1 \) terms)
- \( B_1 = (0, 0) \)
- \( B_2 = (1, 1) \)
- \( B_3 = (2, 1) \)
- \( B_4 = (3, 0) \) (control point coordinates)

For B-spline generation, \( n, k, \) the knot vector, and the polygon vertices must be defined. They have been chosen for this example as shown.

**shape factor calculation** -

for \( 0 \leq t < 1, k=1: \)
- \( N_{11} = 0 \)
- \( N_{21} = 0 \)
- \( N_{31} = 0 \)
- \( N_{41} = 1 \)
- \( N_{51} = 0 \)
- \( N_{61} = 0 \)
- \( N_{71} = 0 \)

for \( 0 \leq t < 1, k=2: \)
- \( N_{12} = 0 \)
- \( N_{22} = 0 \)
- \( N_{32} = (1 - t) \)
- \( N_{42} = t \)
- \( N_{52} = 0 \)
- \( N_{62} = 0 \)

for \( 0 \leq t < 1, k=3: \)
- \( N_{13} = 0 \)
- \( N_{23} = (1 - t)^2 \)
- \( N_{33} = 2 t \) (1 - t)
- \( N_{43} = t^2 \)
- \( N_{53} = 0 \)

for \( 0 \leq t < 1, k=4: \)
- \( N_{14} = (1 - t)^3 \)
- \( N_{24} = 3 t (1 - t)^2 \)
- \( N_{34} = 3 t^2 (1 - t) \)
- \( N_{44} = t^3 \)

**curve equation** -

\[
P(t) = B_1 \cdot N_{14} + B_2 \cdot N_{24} + B_3 \cdot N_{34} + B_4 \cdot N_{44}
\]

\[
P(t) = B_1 \cdot (1 - t)^3 + 3 \cdot B_2 \cdot t \cdot (1 - t)^2 + 3 \cdot B_3 \cdot t^2 \cdot (1 - t) + B_4 \cdot t^3
\]

\[
P_x = 0 \cdot (1 - t)^3 + 3 \cdot 1 \cdot t \cdot (1 - t)^2 + 3 \cdot 2 \cdot t^2 \cdot (1 - t) + 3 \cdot t^3
\]

\[
P_y = 0 \cdot (1 - t)^3 + 3 \cdot 1 \cdot t \cdot (1 - t)^2 + 3 \cdot 1 \cdot t^2 \cdot (1 - t) + 0 \cdot t^3
\]
There are a variety of things that should be noted in looking at Example 1. First, the equations are defined from $0 \leq t < 1$. From equation (2), this can be seen as the only range for which any shape factor will take on a non-zero value. Hence, $t$ will always range between the first knot vector term value and the last knot vector term value.

Second, the spline starts at the first polygon vertex and ends at the last polygon vertex. This is not true of all B-splines, but is determined by the selection of the knot vector. This selection will be discussed further later. The range for $t$ is technically specified not to include the value of $t = l$. For this particular example, that corresponds to the Cartesian point $(3, 0)$. This point is typically included on the end of the final definition range to allow the curve to touch the final vertex point for practical reasons. The last $t$-range is made a less than or equal to condition rather than simply less than.

The knot vector used in Example 1 was chosen for simplicity. The $t$-variable was defined uniformly throughout its entire range. With a more complicated knot vector, there may be several ranges in which $t$ is defined differently in each. Such an example will be seen in Example 3.

Bezier curves are a special subset of B-splines; a Bezier curve is a B-spline with $k$ equal to $n + 1$, and an open, uniform knot vector. An open, uniform knot vector is one with $k$ terms that are identical on each end of the vector. The knot vector from Example 1 is such an
entity. With \( n = 3 \), and \( k = n + 1 = 4 \), there are eight terms in the knot vector. This knot vector starts with 4 identical terms, and ends with 4 identical terms. Usually, these terms are zero and one respectively for mathematical simplicity, although they could take on any value as long as they meet the condition above. Hence, Example 1 was actually a Bezier curve.

The B-spline formulas can be greatly simplified for Bezier curves. This is what makes them so popular. The Bezier formulation is given as follows:

\[
P(t) = \sum_{i=0}^{n} B_i J_{n,i} \tag{4}
\]

\[
J_{n,i} = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i} \tag{5}
\]

The notation here has been kept the same as in reference [1], which starts numbering the vertices with zero for a Bezier curve. In dealing with these formulations, (1) through (5), the first vertex of a B-spline starts with \( B1 \) while the same first vertex using the Bezier curve formulas begins with \( J0 \).

**Example #2:** generation of a B-spline curve using Bezier formulas.

specified - \( n = 3 \)

\( k = 4 \)

\( B_0 = (0, 0) \quad B_1 = (1, 1) \quad B_2 = (2, 1) \quad B_3 = (3, 0) \)

Note that there is no knot vector specified (it is implied in the formulation to be half zeros and half ones - \([0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \])).

shape factor calculation -

\( J_{3,0} = 6/6 \ast t^0 \ast (1-t)^3 = (1-t)^3 \)

\( J_{3,1} = 6/2 \ast t^1 \ast (1-t)^2 = 3 \ast t \ast (1-t)^2 \)

\( J_{3,2} = 6/2 \ast t^2 \ast (1-t)^1 = 3 \ast t^2 \ast (1-t)^1 \)

\( J_{3,3} = 6/6 \ast t^3 \ast (1-t)^0 = 3 \ast t^3 \)

curve equation -

\( P(t) = B_0 \ast J_{30} + B_1 \ast J_{31} + B_2 \ast J_{32} + B_3 \ast J_{33} \)

\( P(t) = B_0 \ast (1 - t)^3 + 3 \ast B_1 \ast t \ast (1 - t)^2 + 3 \ast B_2 \ast t^2 \ast (1 - t) + B_4 \ast t^3 \)
\[ P_x = 0 \ast (1 - t)^3 + 3 \ast 1 \ast t \ast (1 - t)^2 + 3 \ast 2 \ast t^2 \ast (1 - t) + 3 \ast t^3 \]
\[ P_y = 0 \ast (1 - t)^3 + 3 \ast 1 \ast t \ast (1 - t)^2 + 3 \ast 1 \ast t^2 \ast (1 - t) + 0 \ast t^3 \]

The curve equations are, of course, identical to those found in example #1, illustrating that the two sets of formulas are interchangeable. The main advantage of the Bezier subset is its simplicity of formulation. The disadvantage here, however, is that the curve is made up of only one section; hence, any changes to one of the defining polygon points have a global impact on the shape of the entire curve. General B-splines have the advantage of local control. As will be shown in Example 3, the knot vector can be selected to cause the curve to be defined in multiple sections, each defined by only the vertices in that area. A change to one of those vertices has absolutely no effect on the other sections of the curve. This allows for more precise curve control and can be exploited to develop much more complex geometries and features.

On a related note, it was mentioned previously that a generic B-spline does not necessarily have to connect to the two end vertices. An open knot vector causes it to connect to the two extreme vertices, rather than hanging freely in space. The Bezier curve utilizes an open knot vector which is also uniform. Example 3 will also utilize an open knot vector, but \( k \) will not equal \( n+1 \) as in a Bezier curve. There are obviously many applications where it will be prudent to make use of an open knot vector to better predict the curve shape.

**Example #3**: generation of formulas for a B-spline with an open uniform knot vector.

**specified** -

\( n = 4 \)
\( k = 3 \)
\( \mathbf{X} = [0 \ 0 \ 0 \ 1 \ 2 \ 3 \ 3 \ 3] \)
\( B_1 = (0, 0) \quad B_2 = (1, 1) \quad B_3 = (2, 1) \quad B_4 = (3, 2) \quad B_5 = (3, 0) \)

An open knot vector is used, although this is NOT a Bezier curve.

**shape factor calculation** -

for \( 0 \leq t < 1, \ k=1: \)
\( N_{11} = 0 \quad N_{21} = 0 \quad N_{31} = 1 \quad N_{41} = 0 \quad N_{51} = 0 \)
\( N_{61} = 0 \quad N_{71} = 0 \)

for \( 0 \leq t < 1, \ k=2: \)
\[ \begin{align*}
N_{12} &= 0 & N_{22} &= 1 - t & N_{32} &= t & N_{42} &= 0 & N_{52} &= 0 \\[4pt]
N_{62} &= 0 \\
\text{for } 0 \leq t < 1, \ k = 3: \quad & N_{13} = (1 - t)^2 & N_{23} &= t(1 - t) + 0.5 \cdot t(2 - t) & N_{33} &= 0.5 \cdot t^2 & N_{43} &= 0 & N_{53} &= 0 \\
\text{for } 1 \leq t < 2, \ k = 1: \quad & N_{11} &= 0 & N_{21} &= 0 & N_{31} &= 0 & N_{41} &= 1 & N_{51} &= 0 & N_{61} &= 0 & N_{71} &= 0 \\
\text{for } 1 \leq t < 2, \ k = 2: \quad & N_{12} &= 0 & N_{22} &= 0 & N_{32} &= 2 - t & N_{42} &= t - 1 & N_{52} &= 0 & N_{62} &= 0 \\
\text{for } 1 \leq t < 2, \ k = 3: \quad & N_{13} &= 0 & N_{23} &= 0.5 \cdot (2 - t)^2 & N_{33} &= 0.5 \cdot t(2 - t) + 0.5 \cdot (3 - t)(t - 1) & N_{43} &= 0.5 \cdot (t - 1)^2 & N_{53} &= 0 \\
\text{for } 2 \leq t \leq 3, \ k = 1: \quad & N_{11} &= 0 & N_{21} &= 0 & N_{31} &= 0 & N_{41} &= 1 & N_{51} &= 0 & N_{61} &= 0 & N_{71} &= 0 \\
\text{for } 2 \leq t \leq 3, \ k = 2: \quad & N_{12} &= 0 & N_{22} &= 0 & N_{32} &= 0 & N_{42} &= 3 - t & N_{52} &= t - 2 & N_{62} &= 0 \\
\text{for } 2 \leq t \leq 3, \ k = 3: \quad & N_{13} &= 0 & N_{23} &= 0 & N_{33} &= 0.5 \cdot (3 - t)^2 & N_{43} &= 0.5 \cdot (t - 1)(3 - t) + (3 - t)(t - 2) & N_{53} &= (t - 2)^2 \\
\end{align*} \]

curve equation -

\[ P(t) = B_1 \cdot N_{14} + B_2 \cdot N_{24} + B_3 \cdot N_{34} + B_4 \cdot N_{44} + B_5 \cdot N_{54} \]

\[ 0 \leq t < 1 \rightarrow \]

\[ P(t) = B_1 \cdot (1 - t)^2 + B_2 \cdot [t \cdot (1 - t) + 0.5 \cdot t \cdot (2 - t)] + B_3 \cdot 0.5 \cdot t^2 \]

\[ P_x = 0 \cdot (1 - t)^2 + 1 \cdot [t \cdot (1 - t) + 0.5 \cdot t \cdot (2 - t)] + 2 \cdot 0.5 \cdot t^2 \]

\[ P_y = 0 \cdot (1 - t)^2 + 1 \cdot [t \cdot (1 - t) + 0.5 \cdot t \cdot (2 - t)] + 1 \cdot 0.5 \cdot t^2 \]

\[ 1 \leq t < 2 \rightarrow \]

\[ \text{page - 8} \]
\[ P(t) = B_2 * 0.5 * (2 - t)^2 + B_3 * [0.5 * t * (2 - t) + 0.5 * (3 - t) * (t - 1)] + B_4 * 0.5 * (t - 1)^2 \]
\[ P_x = 1 * 0.5 * (2 - t)^2 + 2 * [0.5 * t * (2 - t) + 0.5 * (3 - t) * (t - 1)] + 3 * 0.5 * (t - 1)^2 \]
\[ P_y = 1 * 0.5 * (2 - t)^2 + 1 * [0.5 * t * (2 - t) + 0.5 * (3 - t) * (t - 1)] + 2 * 0.5 * (t - 1)^2 \]

\( 2 \leq t \leq 3 \rightarrow \)

\[ P(t) = B_3 * 0.5 * (t - 3)^2 + B_4 * [0.5 * (t - 1) * (3 - t) + (3 - t) * (t - 2)] + B_5 * (t - 2)^2 \]
\[ P_x = 2 * 0.5 * (t - 3)^2 + 3 * [0.5 * (t - 1) * (3 - t) + (3 - t) * (t - 2)] + 3 * (t - 2)^2 \]
\[ P_y = 1 * 0.5 * (t - 3)^2 + 2 * [0.5 * (t - 1) * (3 - t) + (3 - t) * (t - 2)] + 0 * (t - 2)^2 \]

Figure 3. Plot of Example 3
As Example 3 illustrates, the spline is not made of one function, but three independent functions, each covering a separate section of the curve. Any change in $B_1$, for instance, only affects the section defining $t$ from zero to one. The other two sections are unaffected. Once again, it is worthwhile to note that the $t = 3$ point is part of the curve. The last set of equations defining $P(t)$ has been manually set to include this last point, which happens to be $(3, 0)$, or $B_5$.

The pieces of the curve are continuous to the order of $k-1$. For Example 3, the various sections of the curve have second order continuity to each other. The separation points match, as do the slopes of the curve. This can be very important to note in applications involving geometric surfaces, as the Airfoil Spline Program does.

### 1.2 Matrix Form of Splines

For purposes of implementing the use of these splines on a computer, the matrix representation of the curves will be used. Since each data point along the curve is just a summation of the polygon vertices times the corresponding shape factors, the matrix representation is simple.

$$[N] [B] = [P] \tag{6}$$

As before, $N$ is the shape factor matrix, $B$ is the matrix of polygon points, and $P$ is the matrix of resulting data points. Using equations (2) and (3), it is possible to calculate the shape factor matrix for $t$ over specified intervals. In this matrix formulation, the shape factor matrix has a number of rows equal to the number of points desired and columns equaling the number of polygon points $(n+1)$. The $B$ matrix has rows equaling the number of polygon points and columns equal to the number of spatial coordinates used. Finally the resulting data point matrix has a number of rows equal to the number of points specified and a number of columns equal to the number of spatial coordinates in question.

If the letter $d$ is arbitrarily selected to represent the number of data points and $C$ for the number of coordinates that a particular spline will have, then equation (6) becomes:
\[ [N]_{d \times (n+1)} [B]_{(n+1) \times C} = [P]_{d \times C} \]  

(7)

### 1.3 Curve Approximation with B-splines

It is easy to implement this matrix formulation to generate B-spline data, but this relation is also useful to fit a spline to existing data. In this case, the \( P \) matrix is known. The \( N \) matrix is actually not known, but \( t \) intervals must be assumed so that the \( N \) matrix can be generated. A typical choice would be to choose uniform spacing of \( t \), or to space \( t \) in proportion to the spacing of the data. Different choices will result in very different polygon points, and some choices of \( t \) may result in a more useful set of polygon coordinates for a particular application than others, but any choice of \( t \) will result in an exact and valid curve.

Really, this is just a case of solving a set of simultaneous equations. Unfortunately, this would require the number of data points to be equal to the number of polygon points, which is not usually a practical scenario. Typically the polygon points number much less. In this case, there are actually far more equations than unknowns. It is, therefore, necessary to convert this to a square matrix by multiplying both sides of equation (6) by the transpose of \( N \) results in the following:

\[
[N]^T [N] [B] = [N]^T [P]
\]

(8)

This preserves the equations, but results in the unknown matrix, \( B \), being multiplied by a square matrix. From this point, any method that solves simultaneous equations can be used to calculate \( B \). The software in Appendix A uses a traditional method of matrix inversion, although any faster numerical technique like Gauss-Jordan elimination could be used to solve this system of equations. The matrix inversion takes on the following form:

\[
\]

(9)

It is important to notice here that the method uses a direct solution of the curve parameters based on the input data. There is no curve fitting approximation error for this sort of operation, as using equation (6) at the outset of this calculation will just result in getting
the exact data points back out of the spline. Traditional curve fitting methods approximate data by decreasing approximation error until it is within acceptable range. In fitting splines to data, the solution is direct, so there is only the possibility of rounding errors. Depending on the application of the spline after it is curve fit, this offers a highly reliable and accurate method of storing and using data.

Since the software in the appendices makes use of splines, these topics will be revisited in later sections in looking at the algorithms used in the code. This introduction has been provided to develop a basic understanding of the analytical reasoning behind the numerical implementations of the computer code. Actual curve fitting as applied to airfoils will be discussed later.
2.0 Selected Topics of C++

C++ has been a significant part of this work, and this section deals with some of the aspects of the language that have been used extensively in this thesis. No attempt will be made to provide a comprehensive discussion of the language, and readers are encouraged to refer to the reference list for detailed information on the subject of C++ command structure and syntax. References [2], [3], [4], [5] and [6] deal extensively with the details of C and C++.

The codes of Appendix A and Appendix B represent the complexity and richness of the programming language which has constituted much of the work. Only a few details will be discussed here to help understand some of the useful syntactical and organizational tools utilized in developing those two programs. A more detailed description of the actual algorithm followed will be given in a later section. This section will touch upon a few of the useful realizations that are somewhat confusing in learning C++, or that have not been explained explicitly in any of the references.

The C++ programming language is basically an extension of the C programming language. C was developed in the 1970's as a high-level, structured programming language, and is still one of the most popular in use today. C++ utilizes all of the standard C command and libraries, as well as the majority of the syntax, but adds a variety of new features in an effort to make C faster, simpler, and more powerful to program with.

C++ includes a wealth of new commands and operators designed to clean up some difficult constraints left by C. As an example, the new command in C++ greatly simplifies the procedure of memory allocation in which C programmers are forced to use the malloc command. Many examples can be found in the area of input/output (IO) and file streaming, in which C++ objects have replaced the difficult and confusing commands of C. A standard ANSI C++ compiler is capable of using the older standard C commands like malloc if needed. Many of the basic commands are interchangeable between C and C++, but it should be noted that this thesis employs C++ structures, which implies that some of the code algorithms would have to be re-written to compile using a standard C compiler.

All of the programming in this thesis was completed using exclusively the Microsoft Visual Studio, which is an encompassing development environment that can compile code
written in several languages. The reasons for this choice will become more apparent in analyzing the possibilities of multi-language interaction and collaboration. The bulk of the programming has been done in C++, which is one of the standard modules of Visual Studio. This is important to realize, for there are subtle differences in different C++ compilers. While effort has been made to follow standard ANSI C++ practice, there is no guarantee that another compiler would handle all of this code in the same way. In particular, the MFC libraries (discussed later) are uniquely Microsoft's product.

This is the second reason for selecting the Microsoft brand C++ compiler. There are some other companies offer similar packages to the MFC libraries, which are object extensions designed to be used for developing Windows programs. However, the MFC libraries are guaranteed to be completely compatible and integrated into the Windows environment in the most efficient way, since it has been produced by the same company that produced Windows itself.

2.1 Structures

Most software written in recent times is known by the term object-oriented. In order to define and understand objects, it is first necessary to back up one step to a relatively simpler construction -- the structure. The structure is a user defined variable type that allows for more complex types of variable storage. It is important to remember that a structure is used just like any of the basic variable types, but is significantly more complex.

A structure is quite simply a collection of variables that are related. These variables, sometimes called members or fields, make up the attributes of something. These variables are bound together and referenced under the single structure name. An analogy to this would be the concept of a toolbox. The tools inside each have a separate function or use, but they are collected for a common use of fixing things. Because the various tools are typically used conjunctively, they are stored inside one easily transported case to keep them all together. The variables of a structure can be thought of as the tools, while the structure itself is represented by the toolbox that binds them together. Reference [4] and [3] give more complete discussions of structures, including implementation procedures.

While structures are nice and it is easy to see that a structure could have a multitude of uses, a glance at Appendix A and B reveals that there is not one single structure in total in
those codes. Instead, these programs make extensive use of something called a class. The class represents an evolution of structures into a newer, better form. Classes utilize the strengths of a structure and add some additional features to enhance its usefulness. A convenient way of doing this is to look at some of the faults with the use of structures.

A big concern with using structures is that the data that it contains is available and accessible by any part of the software. This might sounds like a good thing at first, but consider the complications of dealing with a large program like the Microsoft Word this document has been typed in. In a large program such as that, errors can be almost impossible to track down. If the data members of a structure are corrupted or contain the wrong information, literally any part of the code could be the cause of the malfunction. In this sense, it would be better if the data could be protected in some way, so that the data could only be modified under certain conditions. An error in that case would immediately be traceable to one of those channels of access.

Second, grouping data together inside a shell is a very good idea. That data can then be transmitted and used in the program as a unit. This has been described above in the use of the structure. However, sometimes it is not only variables that have a common theme. Sometimes, it would also be nice to add functions to the toolbox that fit the theme. It would be nice to be able to tie those types of functions to the structure in some way since they are all related.

2.2 Classes

These concerns over the use of structures leads to the concept of the class. A class incorporates both of these features, and builds on the existing idea of a structure. That is why it is necessary to understand the implementation of a structure first. All a class really does is build on that concept. The Cay Horstmann book [2] is immensely useful in learning about the origins of the class, as well as a comprehensive description of the associated terminology and design criteria.

The difference between declaring the class and the structure is that the list of data members is not just a simple list as it is in the case of the structure. For a class, the members are divided into three sections: public, private, and protected. Going back to the toolbox analogy, these three sections can be thought of as drawers in the toolbox which separate the
tools (or data members). These three sections are explicitly declared using three very special keywords: public, private, and protected. Not all classes have every section type. Many have only one or two.

In a given class, the public section operates exactly like a structure does. When a variable of that class type is declared, things in the public section are accessed using the dot operator exactly as they are in the case of structures. This includes both variables and functions.

Anything in the private section can only be accessed by members of the class itself. Nothing outside the class may make use of variables or functions in the private section. This solves the whole problem of protecting data. The proper way of setting up a class involves using a public function to access anything in the private section of the class. In this way, there is only one method of accessing that piece of data. Any errors in the data handling can be tracked directly to that access function.

Proper programming etiquette states that any variables in a class should never be placed in the public section. They should always be set and accessed by public functions. While it is occasionally useful to place a variable in the public section, this practice is generally discouraged and considered bad style. Functions are found in all sections of a class, depending on their purpose. Anything that is used only internally by the class should probably be placed in the private section for the same reasons that the variables are secured.

As [2] discusses, the data members are usually referred to as fields in a class, while the functions of a class are sometimes called operations. This can be a bit confusing, because they are different than operators like the dot operator or the arrow operator. For this reason, effort will be made in this thesis to clearly state whether a class operation or an operator is the subject of discussion. The term class function will sometimes be used to make the distinction.

Of course, it is easiest to demonstrate all of this with an example, so, as an extremely simple example, a class called RectData, will be developed. This example will help to understand the more complicated classes that appear later in the code of Appendix A and B.

class RectData //sets the class name
{

public:  //begin public section of class
    void SetValues(double a, double b, double c, double d);
    double GetValue (int PointNum, int XorY) const;
private:  //begin private section of class
    double x1;
    double y1;
    double x2;
    double y2;
};

void RectData::SetValues(double a, double b, double c, double d)
{
    //initializes the four private variables defining the
    //rectangle to the input parameters of the function
    x1 = a;
    y1 = b;
    x2 = c;
    y2 = d;
}

double RectData::GetValue (int PointNum, int XorY) const
{
    //retrieves a particular point and coordinate for outside
    //use.
    //use PointNum = 1 or 2 for that point number
    //use XorY = 1 for x-coordinate; use XorY = 2 for y-
    //coordinate
    if (PointNum == 1)
    {
        if (XorY == 1)
            return (x1);
        if (XorY == 2)
            return (y1);
    }
    if (PointNum == 2)
    {
        if (XorY == 1)
            return (x2);
    }
return (x2);
if (XorY == 2)
    return (y2);
}
return (0);  //functions returns a default of 0 if the wrong
//input is given
}

This is an example of a completely defined class that can be used in a program. The class is used exactly like a structure is, as a newly defined type of variable. To make use of this construction, a variable of RectData type must be declared in the program. A variable of a class type is called an object.

First, there is one keyword used in this class which has not been mentioned thus far. It is the word const; the use of const is a convention that helps make code a little bit easier to read and more robust. Any function designated with const in this way cannot modify any of the fields of the class. Doing so would result in a compiler error. It would be possible to drop the const, but simply not take advantage of the ability to modify a field. This can create confusion though, for there are two types of functions in a class. Those that can be used with the const keyword do not modify any fields in a class. They only access and report their findings; these are called accessors. Other types of functions inside a class set or modify the data fields in a class; these are called mutators. The const keyword helps split these two groups up for easy, quick identification.

Of course, this is just the beginning of the development of classes. They are capable of a lot more. Classes typically have two special functions in place. In fact, proper programming procedure is to include these two special functions even if they are just empty. These two special functions are called the constructor and the destructor. Each is called exactly once in an object (provided they exist at all). The constructor is called when the object comes into existence, and the destructor is called when it goes out of scope and is deleted.

There is a lot of special consideration with regard to these two special functions. Both are very unique in the C++ world, with regard to both syntax and usage. A constructor always has the exact same name as the class itself. For instance, in the case of the RectData
class above, the constructor for the class would also be RectData(). The constructor and destructor are the two exceptions to the rule that every function must have a type. Really, a type of void is sort of understood, for the constructor and destructor returns no value, but these are special cases; no type is specified in the declaration.

A constructor is set up to take parameters to initialize the object when it is declared. It is automatically called if a constructor is defined, so there is no need to do anything special, with the exception of sending the right arguments during object creation.

It is necessary here to define another term that comes up very often when dealing with constructors. Overloading can be done to just about anything in C++ (functions, class operations, operators, etc.), but it is probably used most often in dealing with constructors. Overloading means that a particular function or operator or whatever is defined in more than one sense, i.e. that it is defined to handle more than one variable type. A good example of this is the plus sign. It is a basic operator that comes pre-defined in C++, and it is overloaded in the compiler. The plus sign can be used to add two integers, or two double precision numbers. The compiler performs each with the same proficiency, but the operations are different. This might not seem that spectacular, but consider the addition of two strings. The strings cannot be added like a pair of numbers, but the computer can perform the operation anyway. It simply joins the strings together, one after the other. In fact, the plus sign can be defined to "add" any type of structure, variable, or class, including adding together two of the RectData classes defined above (in some sense anyway). This is overloading.

When overloading a function, especially constructors, the function is defined multiple times with different input arguments in each context. When the function is called, the computer checks the input arguments to match it up type-wise with one of the defined versions of the function.

As it stands, the RectData class requires two steps to make it useful. First, an object of that type needs to be declared. Then, its four values need to be set using the SetValues() class operation. This can be shortened to one step by implementing a constructor that will automatically set the four values upon creation. Its implementation would look like this:

RectData::RectData(double a, double b, double c, double d) {
    //constructor for RectData
// sets the fields upon object initialization
   x1 = a;
   y1 = b;
   x2 = c;
   y2 = d;
}

Now, the object is declared and set in one step. The only problem with this is that it is now always necessary to declare objects of this class type with the four fields attached. It is now impossible to declare an object without passing any parameters. If some other programmer picks up this class and tries to use it without knowing that, a program error will result. For this reason, there is almost always more than one constructor, making it overloaded.

Actually, to solve this problem, the easiest way is to make a default constructor. The default constructor is one with no input arguments at all. If the computer cannot match a constructor call up to one of the predefined constructors, it uses this default constructor automatically. The object may not be initialized in the way the programmer would like, but at least there is no error. This default constructor usually sets the object fields to some default value. For RectData, depending on its context, it might make sense to set all four fields to zero until they are initialized later using SetValues().

The destructor is a little simpler to use, although just as important. It is automatically executed like the constructor, except that the destructor is called when the object is deleted, destroyed, or goes out of scope. Going out of scope refers to the condition of the piece of code that declares a variable leaving memory. If the variable is declared in a function, this happens when the function ends; if the variable is declared globally, it goes out of scope when the program ends.

The destructor never takes any parameters, so it cannot be overloaded. There is always only one. The destructor is also named the same as the class, except that the name is preceded by a tilde. In the example used thus far, the destructor would be known as ~RectData(). The destructor is used only for clean up purposes. The main reason is to reclaim any memory space captured by the class. Appendix A has many examples of reclaiming memory space in destructors. Further, it might be used to write data to a file,
close any active files or sockets, or a variety of other such purposes to eliminate the object without causing the any problems with the program.

A note should be made here about some of the artistry and aesthetics involved in programming. While computer programs are logical, functional entities, there is a certain amount of beauty and symmetry to the code as well. There are always a hundred ways to do something in C++, and, while some may work better than others, it is usually up to the programmer's sense of style to pick his or her particular favorite method for each instance. Also, many things are done in a program not to serve a necessary functional requirement, but for completeness. Things are done to make the code consistent, or to make it a little easier to modify later on.

Adding a destructor to the RectData class falls into this category. There is no cleanup to perform on the class, but a destructor should be added nonetheless, and simply left blank. As just mentioned the reasons for this are consistency and giving a future programmer a place to add code in case RectData might require some cleanup in case it is ever modified. For completeness, here is a blank destructor for RectData.

```cpp
RectData::~RectData()
{
}
```

### 2.3 Inheritance

There has been a lot of discussion about classes thus far, and one can already see the power of classes, but there is a lot more. It is prudent now to define a word that is often used in dealing with classes, and which describes another interesting property of classes. A class possesses a unique ability called inheritance.

**Inheritance** allows a class to extend the attributes of another class. If two classes are going to have a great deal of fields and class operations in common, there is no point in retyping the common information. Instead, it is better to have them inherit their commonality. When a class inherits from another one, it gets all the properties of its base class. There are a lot of subtle implications involved here. Examples involving inheritance can get involved, so the example that will be used here will be taken from Appendix A, in a
pair of files called spline.h and spline.cpp. In these files, there are several classes defined. The first, Base_spline, is much like what has been discussed so far. It has some constructors, a destructor, some fields, a few class operations, etc. The Bspline class, also has the usual constructors and destructors, but has very few member fields and class operations. However, what is implied in the code is that Bspline has everything that Base_spline has.

```cpp
class Bspline : public Base_spline

The public Base_spline keywords are the declaration mechanism for inheritance.

Notice that OnlyGetN_spline has an identical declaration, for it too inherits from Base_spline. The interesting part is that Base_spline now has a constructor, and Bspline also has its own constructor. In reality, both are used, as shown here:

```cpp
Bspline::Bspline (int init_n, int init_k, int init_num_points, double *init_knot) : Base_spline (init_n, init_k, init_num_points)
```

When a Bspline object is constructed, the first thing that happens is that it calls the corresponding Base_spline constructor that is listed after the colon. After the base class constructor has completed execution, it returns to the Bspline constructor and it proceeds as usual. Note that the Base_spline declaration passes the arguments as a normal function would do. This is the only instance where a constructor can be called without being in direct object creation.

Destructors are a little easier. Since there's only one, there is no need to specify in destructor calls to call the base class destructor. In this case, ~Bspline() takes care of anything specifically restricted to the Bspline class while ~Base_spline() takes care of all of the inherited properties. Upon object destruction, both are called in succession automatically.

The inheritance property is related to the protected section mentioned before, for the files in Appendix A make frequent use of the protected section rather than using a private section. As stated before, the public section can be accessed by any part of the program that can access the object. The private section is restricted to only the inner workings of the
class. The protected section can be accessed only by the class internally or internally by any other class that inherits from it. Bspline, for example, uses many of the fields of Base_spline. If Base_spline had been private instead of protected, this would not be possible.

2.4 Polymorphism

There's one more aspect of class inheritance that deserves mentioning, although it was not directly used in any of the examples so far. There is a term called polymorphism that is inevitably linked to inheritance. There are cases where it would be nice to inherit from another class with the exception of one or two class operations. Polymorphism allows a programmer to overwrite those few troublesome operations for the derived class.

Polymorphism is tied to something called virtual class. It is easy to overwrite or modify operations in a derived class, but those class functions must have been declared as being virtual. The issue here is that redefining the function means that there are now two versions of that function available to the class, the one it inherited, and the one that explicitly defined. Being virtual tells the computer to use the defined version unless otherwise specified.

Of course, a particular operation does not necessarily need to be completely re-written; it could be extended instead. There is a way to call the original version of the function by preceding the function call with the name of the base class followed by two colons. This means that a polymorphic class can begin by calling its ancestor and then proceed with additional code.

The Microsoft MFC libraries are almost completely made of classes, and they make exceptionally heavy use of inheritance, polymorphism, and virtual functions. As an idea of the interactions happening in MFC, the main application object is always created by overriding a virtual function called InitInstance(). In contrast, dialog boxes exchange data between themselves and the greater program through a function called DoDataExchange(). This function is always implemented by immediately calling the parent CDialog version of DoDataExchange() to take care of the unseen items followed by any data exchange specific to that particular dialog box.

The whole concept of classes and all of these related terms will be seen in much greater detail in section 3 on MFC in this thesis. Learning about the MFC methodology has
consumed a great deal of this effort. MFC requires a surprisingly unique mentality, even for an experienced C++ programmer. Before MFC is discussed, the most notable aspect of C and C++ must be mentioned, which is pointers. Considering the nature of the software written for this thesis, the discussion of pointers will have special consideration toward their relationship to multi-dimensional arrays, for this is a point that has not been discussed well in any of the references. Implementing two-dimensional arrays was a major obstacle in creating the software of Appendix A, and took some innovative and creative thinking in manipulating the syntax of C++.

2.5 Pointers and Arrays

A pointer is simply a variable that points to another variable; that is, a pointer stores the address of another variable's location. Anyone who deals with computers regularly probably knows that a computer relies on memory resources to store relevant information for the processor. This memory is divided up into many sections, each with a specific and unique address, so that the processor knows where to look when it needs a particular piece of information.

A pointer is a variable just like any of the basic types discussed so far, except that it stores one of these memory addresses instead of a number or letter like usual. This might sound trivial at first, since it would seem senseless to store the location of a variable when one can just directly use that variable anyway. Actually, pointers probably have more uses than the actual variables to themselves. Some of these uses will be described.

It is important to remember that a pointer contains the address to a variable, not the variable itself. A pointer has its own address, which is occasionally useful, but its own physical location in memory has nothing to do with what it stores -- the location of the variable it is associated with.

To assist the programmer, C++ has overloaded two more operators specifically for use in dealing with pointers: the asterisk and the ampersand. It is interesting to note that this is why the and operator consists of two ampersands; it has to be differentiable from the pointer operator. The ampersand can be translated as meaning "the address of" while the asterisk can likewise be interpreted as meaning "the variable located at."
A pointer is considered a type of variable just as the normal variables are. In fact, pointers are defined to hold a specific variable type of memory address. There are pointers of all the basic types: int, double, float, and char. There are also void pointers, which is a pointer whose type is not known. It is further possible to define a pointer of any of the definable types of variables, such as structures or classes. Each type of pointer still stores a memory address no matter what the case. The type of pointer it is defined as reflects what type of variable can be found at the memory address it stores.

So far, this discussion might seem redundant or trivial, but the first main use of pointers can be illustrated by considering a simple situation. Suppose it is necessary to develop an algorithm to swap the values of two variables. The best way to do this using a conventional approach is to use a temporary value to store the first number while the second and first are switched, then set the second to the temporary variable. Of course, the smart thing to do in this case would be to build a function to swap the numbers so that it can be reused many times. Unfortunately, it is impossible to use a function to modify these two numbers directly.

This algorithm can easily be re-written, however, using pointers to build an adaptive function. This example also makes use of the void function type. Void functions do not return any value, and a return statement in them actually can cause an error. In this case, the function does not need to return anything.

```c
void swap (int *a, int *b)  
{
    int templ = 0;
    templ = *a;
    *a = *b;
    *b = templ;
}
```

To fully understand this methodology, it is important to also look at how this function could be used, for it is a little different that in normal function calls. To facilitate this, an example of a call from a main function will be given.
swap (&a, &b);    //calls the swap function

Notice that the swap function's arguments are not simply a and b. Instead, the ampersand
operator is used for both to submit the address of both variables instead. The actual function
itself is prepared for this, as its two input arguments are declared as pointers instead of
normal variables.

Passing pointers to a function instead of the actual variable is known as passing by
reference. This term is self-explanatory, for a reference to the variable in question is passed
instead of the variable itself. Passing by reference also eliminates any constraints in function
usage, because a function can access a specific variable indirectly through its pointer.

The second major use of pointers is in the declaration and access of arrays, or
matrices. Very often, especially in mathematical operations, it is desirable to create a group
of objects or variables rather than just one. Arrays can save on variable declarations, and it is
easy enough to set up loops to access the individual elements in a repetitive fashion without a
large amount of manual lines of code.

Arrays and pointers are invariably linked, although many books fool their readers into
believing otherwise. There are two ways to declare an array. The common way that books
show is almost identical to the declaration of a variable.

    int x[10];    //defines an integer array of 10
                  //elements (0-9).

This declaration defines an array of ten integer variables in a sequence. Those variables are
numbered x[0] to x[9], and can easily be accessed in that way.

What is sometimes difficult to understand, however, is that x is actually a pointer.
This pointer points to the first element of the array, and the index notation simply moves
ahead that many elements from that address. The brackets are actually overloaded to provide
an easy way to perform pointer arithmetic. This is a concept that must be understood before
the relationship between pointers and arrays can be fully exploited.

Pointer arithmetic does not correlate well to any physical mathematical operation.
The operating system of any computer indexes and labels the memory space of the system
for easy access. The resulting pattern of memory addresses form a linear stack of memory
blocks; pointers take full advantage of this feature by allowing common arithmetic operations -- excepting that memory addresses are manipulated instead of simple numbers.

Referring to the previous example, note once more that the previously declared variable $x$ is actually a pointer to the first element in the array that it indexes. For instance, $x[5]$ and $(x + 5)$ both produce access to the same variable location in the array. They are completely interchangeable. The bracket notation with the index inside it is just another example of operator overloading to make program code a bit easier to read. This illustrates that taking the pointer $x$ and adding 5 to it yields the address of the sixth variable in the array. A physical picture of this will probably clarify things somewhat:

![Figure 4. Memory mapping configuration](image)

This is a pictorial representation of how the memory space can be accessed. The pointer $x$ -- which occupies some memory itself -- holds the address to the first element of the array. The elements are stacked accordingly, so that addition and subtraction of the pointer address can slide along between the elements.

In fact, this pointer/array relationship is so intricate that there is an alternate way to use arrays by declaring a pointer and then dynamically allocating the size of the array as needed. C++ provides two keywords, `new` and `delete`, that can generate dynamic arrays at runtime. The `new` operator reserves memory space for arrays, and sets a specified pointer to the first element of the array. It is used in the following manner:

```c++
int *x;  // declares an integer pointer
x = new int [10];  // declares a dynamic array of 10 integers
```
The *new* command actually does a great deal for a programmer. In the old C language, it was necessary to manually reserve memory blocks using a keyword known as *malloc*. Visual C++ will still respond to *malloc* commands, but this is totally unnecessary. The problem was that *malloc* reserved a specified number of bytes for an array. The different variable types (*int, double, float, char*, etc.) each take up a different amount of space, and, depending on the processor architecture, these space requirements might differ from computer to computer. It was necessary to run some preliminary lines of code to test the size of a variable type before it could be reserved using *malloc*.

The *new* command is easy to use. Note the specification of the desired variable type immediately following *new*. It performs the necessary internal checking and allocates exactly enough space to hold 10 integers, regardless of how many bytes that might be. As it is used in a command of the form "x = ", it then returns an address to the pointer x for the beginning of the reserved memory space.

Defining an array or variable this way is known as declaring a variable on the heap. This means that it was declared using dynamically allocated memory and assigned to a pointer. Using the other, simple method of defining variables or even arrays is known as declaring a variable on the stack.

Declaring variables on the heap has several advantages, some of which will be seen later, but it also has one incredibly huge disadvantage. Any variable declared on the stack will be automatically deleted when it goes out of scope. A variable's scope refers to the circumstances under which it was declared. Any variable declared inside a function goes out of scope when the function ends. Any globally declared variable or one declared in the main section goes out of scope when the program ends. Unfortunately, a variable or array declared on the heap does not automatically get removed from memory. The memory space must be manually freed after it is done being used. Failing to do this will leave the memory flagged as in use, and will not be used by the system for other things. This is what is known as a memory leak, and it can make the system crash or become unstable. In order to free memory allocated from the heap, it is necessary to follow any *new* declaration with a *delete* command when the variable or array is no longer useful.

Windows 95 has implemented some memory management features, and will automatically clean out and reclaim an application's memory space when it ends, so the
operating system is not in danger. A memory leak will still cause the application to run out of memory though, making memory leaks locally dangerous.

### 2.6 Multi-dimensional Arrays

Fortunately, C++ allows the declaration of multidimensional matrices just as a single dimensional matrix is formed. The term dimension has a slightly different meaning in the programming sense than it does in the mathematical sense. When dealing with arrays in programming, the dimension refers to the number of subscripts used to define elements in the array. To add dimensions, simply add the appropriate dimensions in brackets in sequence. For example, to declare multiple dimensional array on the stack,

```c++
int x[10];       //declares a 1-D array
int y[10][10];   //declares a 2-D array
int z[10][10][10]; //declares a 3-D array
```

Dimensions can be stacked up in this way infinitely, and accessing the array is exactly the same as before; for instance, the first element in z is `z[0][0][0]` while the last element is `z[9][9][9]`.

Although declaring a multidimensional array on the stack is quite similar, declaring a multidimensional array on the heap requires a little bit more discussion. Throughout the declaration, only the first dimension remains dynamic. Unfortunately, C++ is not yet able to handle multidimensional dynamic allocation. Essentially, adding a second dimension can literally be interpreted as an array of arrays. The syntax for declaring the above arrays `x`, `y`, and `z` dynamically looks like the following:

```c++
int numelem = 10;
int *x;
int (*y)[10];    //declares a 2-D array
int (*z)[10][10];
x = new int [numelem];
y = new int [numelem][10];
z = new int [numelem][10][10];
```
This declaration illustrates a useful concept -- that the declaration of an array can involve a variable instead of a number. The problem is that, for multidimensional arrays, only the first dimension can be a variable; all subsequent dimensions must be constants. Also, note that in the pointer declaration, the pointer name itself is in parentheses. This indicates a pointer to an array of arrays, which is completely legal; leaving the parentheses out causes an array of pointers to be declared. This is also legal syntactically, but is not what is needed for the array. It would cause an error.

The implications here are rather subtle, so special care will be taken in describing the situation with respect to multidimensional array variables. The issue that needs to be addressed is the method of passing arrays to functions as input arguments. It is easy to pass single dimensional arrays to a function simply by passing the pointer as a function argument. Then, the array can be accessed by either method already described. As an example, consider a function to automatically sum a series of numbers already stored in an array called $x$ (the array from before):

```c
int SumArray (int *x)
{
    int total = 0, index = 0; //declares integers
    for (index=0; index <10; index++) //begin for loop
        total = total + x[index]; //calculates //summation
    return (total); //return value
}
```

This function totals and returns the sum of the ten integers stored in $x$. This function, however, requires that $x$ be exactly of ten integers in length. A good programming strategy would be to make this function generic so that it can accept arrays of any length and calculate their total. This can also be done very easily by adding a second input parameter to the function.

```c
int SumArray (int *x, int arraylength)
```
{ 
    int total = 0, index = 0; //declares integers
    for (index=0; index < arraylength; index++) //begin
        //for loop
        total = total + x[index]; //calculates summation
    return (total); //return value
}

The function is now set up to take an array of any length, as long as the function receives both the number of elements that the array holds and a pointer to the beginning of it. In the case of x, the main program would that calls this function would make a statement something like the following:

    int totalsum = SumArray (x, 10); //declares and initializes
        //an integer variable

The variable totalsum will receive the returned value of the function. Obviously, this second version of the function SumArray offers a great deal more flexibility than the first version.

This algorithm was easy to implement with x, but y and z offers a greater challenge. Passing these two arrays to a function requires the input argument to be structured exactly like the pointer declaration was. In the case of y,

    int SumArray2 (int (*y)[10], int arraylength) 
    {
        int i = 0, j = 0;
        int total = 0;
        for (i=0; i<arraylength; i++)
            {
                for (j=0; j<10; j++)
                    total = total + y[i][j];
            }
        return (total);
    }
This situation presents a problem; except for the first, all subsequent dimensions must be constant for the function to work. To prove the inconvenience of this situation, it is now time to look at a more practical problem. Calculating the determinant of a matrix is a highly complex and intensive operation in mathematics and engineering. Calculating the determinant of large matrices is a task that requires a computer.

First, it is important to look at the method of calculating a matrix determinant. A determinant can be thought of as a measure of the distortion of a matrix. Only square matrices have determinants, so implementing this operation on a computer definitely requires the use of a two-dimensional array. Unlike the matrix it is calculated on, a determinant is simply a number, which can be negative, positive, or zero. From reference [7], mathematically a determinant is the sum of all of the signed elementary products of the elements in a matrix. An easy way to calculate a determinant is to pick a row or column. Any selected row or column will result in the same answer. For each element in this selected vector, multiply this element by the determinant of the matrix formed by deleting the element's row and column (known as the minor for that element). Make this value positive if the row and column subscripts of the element total an even number; make it negative if the total is odd. Take this positive or negative product and sum up these values for each element in the selected row or column.

Since a determinant is dependent on a series of determinants of matrices one dimension smaller than itself, a 5 by 5 matrix will invariably lead to several 4 by 4 matrix determinants and so forth until one is solving for 2 by 2 determinants. These are easy to find, being equal to the product of the top left element multiplied by the bottom right element minus the product of the top right element by the bottom left.

With the second dimension of an array fixed, implementing a function or program to do this is difficult since the size of the two-dimensional arrays must be specified at compilation time. This means that a separate algorithm would need to be written for each size matrix that is to be solved. And, since the determinant of any size matrix depends on the determinants of all of the sizes below itself, all of these algorithms would need to be implemented into one huge program. Anytime someone would want a matrix larger than the program allows, the program would need to be modified. Fortunately, there is a way around
this problem; it is easy to implement, but somewhat cryptic to understand. In Appendix A, there is a complete function that calculates the determinant of any sized matrix dynamically. It is located in a file called matrix.cpp. Note that this function is of the *double* type, as is the *ret* variable declared inside. It is this variable that the function returns to its calling point.

Second, Determinant receives two arguments as input. The first is a pointer to the very first element of the matrix in question, which is indeed a two-dimensional matrix even though it might not appear to be. The second is an integer indicating the dimension of the matrix (note that the number of its rows and columns must be the same since only square matrices have determinants).

**Example #4**: find the determinant of A.

specified - \( A = \begin{bmatrix} 1 & 0 & 2 \\ 3 & -1 & 1 \\ -2 & 0 & -1 \end{bmatrix} \)

solution - by the first row:

\[
\text{det}(A) = +1 \cdot [(-1)(-1) - (1)(0)] - 0 \cdot [(3)(-1) - (1)(-2)] + 2 \cdot [(3)(0) - (-2)(-1)]
\]

\[
\text{det}(A) = +1 \cdot (1) - 0 \cdot (-1) + 2 \cdot (-2)
\]

\[
\text{det}(A) = -3
\]

If one wanted to use this function to calculate the determinant of matrix A from Example 4, for instance, one might write a small main program like the following piece of code. Looking at how the *Determinant* function might be called is an important part of understanding the relationship of the matrix activity.

```c
int main()
{
    double A[3][3] = {1, 0, 2, 3, -1, 1, -2, 0, -1};
    double det = Determinant( &(A[0][0]), 3);
    cout << det << endl;
    return (0);
}
```
This simple main function, when placed in a project with the Determinant function, displays the expected result of -3 on the screen. Since A is a 3 by 3 matrix, Size is passed as 3. The pointer is specified as the first element of A (inside parenthesis), with the ampersand surrounding it for a pass by reference. Note that A alone is a pointer, but A[0][0] is a double type variable. Simply passing A in the function call would be disastrous, for it is important to remember that A is not a double pointer. It is a pointer to a double type array — a significant difference!

The major effort in coding arrays is understanding pointer arithmetic, coupled with a bit of knowledge about how a multidimensional array is stored in memory. As stated previously, the computer treats a two-dimensional array literally as an array of arrays. Each array occupies a linear portion of memory as shown in Figure 4, inside a larger array that also occupies a linear portion of memory. A graphic can be shown as follows for the case of an array defined as B[2][3]:

![Figure 5. Multidimensional Array Mapping](image)

Each element of the array is lined up, forming a continuous block of all six elements. In this arrangement the elements can be accessed using pointer arithmetic. For instance, simply adding 5 to the address stored by B can retrieve the last element in the array.

The process on which the entire Determinant function is based can be illustrated in the comparison of Figure 6 and Figure 7. These figures consist of two arrays, one being
single-dimensional and the second two-dimensional. Inside each box has been placed the corresponding pointer address of that element based on a pointer $p$.

$$A = \begin{bmatrix}
* (p + 0) & * (p + 1) & * (p + 2) \\
* (p + 3) & * (p + 4) & * (p + 5) \\
* (p + 6) & * (p + 7) & * (p + 8)
\end{bmatrix}$$

Figure 6. 2-Dimensional Array Addressing

$$A = \begin{bmatrix}
* (p + 0) & * (p + 1) & * (p + 2) & * (p + 3) & * (p + 4) & * (p + 5) & * (p + 6) & * (p + 7) & * (p + 8)
\end{bmatrix}$$

Figure 7. 1-Dimensional Array Addressing

Looking at the addresses of these elements, it is clear from the two figures that the arrays contain the same element. The rows of the second array are simply placed in a line instead of stacked together. If it is known that the matrix $A$ from Figure 7 is really a 3 by 3 square matrix, it is very easy to locate any of $A$'s elements simply by the pattern. Similarly, while a matrix input to the previously discussed program is actually a 3 by 3 matrix like Figure 6, the Determinant function treats it like the matrix of Figure 7. With the Size argument also passed to Determinant, it is easy to find all of the elements of $A$. With this knowledge available, it is much easier to decode the methodology behind the Determinant function. This is a powerful example of something that is impossible to do without the use of pointers, pointer arithmetic, and dynamic memory allocation. By analogy, it is now possible to imagine many other similar situations that might arise of this type. In fact, the software of Appendix A has several matrix functions including transposition, matrix multiplication, determinant, and matrix inversion.

One thing that needs to be very carefully monitored is the use of the asterisk operator in dealing with pointers. Looking at Determinant, there are a great deal of pointer assignments and operations taking place. In what was probably a poor choice for the pointer
operator, the asterisk also happens to be the multiplier operator. This can be very confusing for the programmer as well as the computer. While there are no exact rules about how to deal with this, a few simple conventions help to make it more readable. Look at the following line of code from the `if` statement in `Determinant`:

```c
ret = (*p) * (*(p+3)) - (*(p+1)) * (*(p+2));
```

It could be written this way also:

```c
ret = *p** (p+3)-*(p+1)**(p+2);
```

Putting all the pointer assignments in parentheses, and putting spaces around the asterisk when it is being used as a product sign helps to make the first form a bit easier to understand.

A brief look at `Determinant` yields that it operates in the following fashion. The function begins with a group of declarations. The pointer `m` is a pointer to the dynamic array for the minor matrix, which is allocated below the declarations. This array is a square array of `Size` minus one dimension. The pointer `t` is set to the same address as `m`. This is somewhat redundant, but was done for code readability and to insure that the pointer is to a single element at the array beginning.

The function then checks to see if it was passed a 2 by 2 matrix. Since these are easy to calculate, the 2 by 2 determinant is done in one line. If this is not the case, the function uses the previously described method of calculating a determinant by moving across the first row. For each element in the first row, it goes through a somewhat messy process of eliminating the row and column occupied by that element and placing the remaining rows and columns in the newly created minor matrix.

The interesting thing is that this minor matrix is sent as a parameter in another `Determinant` call. Calling a function within itself is perfectly legal, and is known as a recursive function call. Using recursive function calls is an encouraged practice in C++. Each instance of the function occupies its own memory space, so there is no mixing up of the variables or anything like that. Since the minor matrix is one size smaller, `(Size-1)` is sent as the second parameter. Obviously, for a matrix like a 10 by 10, there will be many function instances in memory at once in a large chain, each one filtering down by one mathematical
dimension until the minor matrix is reduced to a 2 by 2, which can be solved. Only then can
the function above it in the chain be resolved and removed from memory.

The variable cofact always takes on a +1 or -1 value. It is simply there because the
summation of the elements across the row times their minor determinant must alternate
adding and subtracting as discusses before. The variable cofact simply alternates to perform
this service. Finally, it should be noted that the m matrix is deleted before the function exits.
It is here that this could be dangerous, for not deleting this allocated space would leak that
must memory each instance of the function, causing a memory leak hundreds of times worse
than a single function alone would produce.

Understanding this methodology of using two-dimensional arrays is essential in
gaining an understanding of the operating principles behind the Airfoil Spline Program
(Appendix A). There is obviously a wealth of other issues in C++, and readers are once
again encouraged to refer to all of the listed references for a better description of the syntax
and keywords of C++. The goal here has been to demonstrate a deep understanding of C++,
and to help fill in a few details that arose in building the Airfoil Spline Program so that it
might better be understood by future readers.

The discussion of computer programming will now turn to the Windows MFC
libraries, which is much like a separate programming language in its own right. MFC is
difficult to learn in the beginning. An introduction is given in the next section to help further
understand the algorithms of Appendix A and B.
3.0 MFC Programming in C++

The Microsoft Foundation Class (MFC) is a set of objects created by Microsoft for use in Visual Studio. The MFC library has been designed to provide programmers with standard Windows objects. These objects are provided to make it easy to construct programs with a Windows interface. Using MFC insures that a program will be completely compatible with Windows 95 or NT. Further, for serious programmers, who want to sell or distribute their software, using the MFC library makes it much easier to obtain Windows certification for their product. This refers to the Window decal that appears on the side of many boxes indicating that the product is Windows compatible.

On a more practical aspect, trying to program in Windows without the MFC library is not an easy job. It is confusing, complex, difficult, and takes an incredible number of lines of code to do each simple task. MFC allows a programmer to do a task with one or two simple lines of code instead of several pages of traditional code. This is because the MFC libraries already contain those pages of code.

Of course, using MFC is not as simple as it may sound at this point. Microsoft has written uncountable lines of code to create this set of standard objects. In order to use these projects, it is necessary to adapt and conform to Microsoft's methodology rather than writing code using one's own methods. Although MFC is simply C++, programming it requires a different mode of thinking than simple console programming. Using MFC is not really like coding in C++; it is simply creating, placing, and removing objects. There is not too much else involved, except that it requires a programmer to learn the objects and how to create, place, or destroy them.

Microsoft has remained as consistent as possible in the use and control of the various objects it implements. It is still necessary, however, to have a basic understanding of the common objects and how to use them. This section will provide an overview to using MFC, with specific reference to the programs in Appendix A and B. Since much of the demonstration of MFC is rather involved, these two appendices will serve as the two major standing examples to the MFC methodology. Limited examples will be given where appropriate, with references to these two programs for major examples. After a brief introduction to the common objects of MFC, a brief discussion will follow on some of the
additional features of the Microsoft Visual Studio (like its wizards, resource viewers, etc.). Applicable references in which a reader might learn more about MFC than what is given in this thesis should consult references [3], [5], [8], and [9]. Further, Microsoft maintains an extensive web-based help system [9] for their Visual Studio products, which is particularly useful in looking for support with MFC.

As mentioned previously, there are two types of executable files for PC's – DOS based programs, and Windows based executables. In Visual Studio, these two types of formats are called console applications and Win32 applications respectively. When starting a new program, Visual Studio provides two project formats corresponding to the type of file to be created. Obviously, since MFC is a library of Windows objects, a Win32 project needs to be created to make use of the library. Note that Microsoft also provides a host of other types of projects that involve DLLs, wizards, frameworks, etc. These projects will automatically utilize the application format appropriate to each case.

There is a small technical note about MFC that must be dealt with before any attempt at actually building an executable file can be made inside a Visual C++ project. Anyone who creates a new, empty Win32 project without using any of the wizards and begins using MFC code in Visual Studio and attempts to compile it will find that the compiler generates a series of errors during the linking phase. These errors give some error codes and some related information, most of which centers around the beginning and end of something called a thread. To avoid this error, the Project menu offers a selection titled Options. In this dialog box, on the first tab, there is a section that has the following three options: not using MFC, use MFC in a shared DLL, or use MFC in a statically linked library. The default is to not use MFC, which is the source of the problem. For optimization, the compiler defaults to not loading the extra MFC module. Using MFC in a shared DLL causes the executable to get its MFC code from a DLL that comes with Visual Studio. Depending on the version, this file is called mfc42.dll. The advantage here is that the actual executable is small, but it requires that DLL on a machine to run properly. Using MFC statically linked means that the executable file is completely independent and capable of running on any standard Windows computer without extra files. It will be a larger file if static linking is used.

It is useful here to define the word thread. There will not be any further mention of this term, for threading is a very advanced Windows topic. Threading refers to the ability of
Windows to multitask, or use the processor to perform many tasks simultaneously. As a good example, Windows is continuously checking both the keyboard and the mouse for input. It does this many times a second, which is why there is no noticeable delay when a key is pressed or the mouse is clicked. This is possible because the mouse and the keyboard run on different threads, and can be checked simultaneously. The benefit to using MFC is that it automatically threads the application for sending systems messages, receiving system messages, screen redrawing, and a variety of other things that all happen at once. For most applications, there is no need for additional manual threading, so it will not be further discussed here.

The first thing to notice about MFC programming is that there is no main function. Standard DOS/console applications utilize a main function as a focal point for program execution. For memory reasons, Microsoft developed a slightly modified executable to use for standard Windows applications. This type of program utilizes a Winmain function that serves essentially the same purpose. MFC, however, uses neither of these.

In reality, there is a Winmain. It is just buried inside the preexisting code. In programming MFC, one never sees this Winmain and there is no need to worry about it. Upon declaration, an application is registered with the system, initialized, and set up to send and receive system messages. All of this is transparent to the programmer.

To utilize the MFC library, it is necessary to include the afxwin.h file in the project. This file is the key to linking with the actual worked out code of MFC. This code is actually located in a huge number of files, but the mentioned file is the gateway to all of them. Including the afxwin.h file in the project will provide access to all the basic features of MFC.

### 3.1 The CWinApp and CmainWnd Objects

All of MFC centers around an object called CWinApp. This is the default object for applications, and is actually pretty extensive, although one never sees much of it. A program is created by creating an object that inherits from CWinApp, and modifies a single virtual function called InitInstance().

Appealing to Appendix A, there is a single line in the file called WinFunc.cpp that consists of the following:
Notice that this declaration is listed globally. It is picked up by the compiler, and will eventually lead to a Winmain function somewhere in the endless lines of code. Moving to the WinFunc.h file, The CTheApp object is declared, inheriting its abilities from CWinApp. All MFC programs should do exactly this. CTheApp has its own destructor which is empty. Other than that, only InitInstance() is defined. It is actually overridden from CWinApp.

A look at this single function reveals that is of bool type, returning TRUE if there are no problems. The body of this function is also relatively brief. There are a few lines which load and initialize some minor items. After that, there is a series of three lines that are also essential to the running of a program in MFC.

For the CWinApp function, this is it. In fact, it is advisable not to modify this object further. Virtually all of its work takes place transparently. By contrast, the other pivotal object in all MFC programs is much more modifiable. This is the main window, known as CMainWnd for the program of Appendix A. CMainWnd is very active in the code, appearing in many forms and doing many things. Note that the main window for the program of Appendix B is slightly different. Appendix B is known as a dialog based program rather than a frame based one; frame based applications are much more common, so it is better to start there.

The CMainWnd object is derived from CFrameWnd, as are most application main windows. The two objects, CWinApp and CFrameWnd represent the two most important objects in MFC, and form the basis of almost every program. CWinApp handles all of the underlying work in making a program run (sometimes called an engine), while CFrameWnd handles most of the visual, functional, and interactive aspects of the program.

CMainWnd comes into creation in those three lines in InitInstance(). Those three lines are among the most essential in MFC, so they will be reproduced here for study.

```cpp
m_pMainWnd = new CMainWnd();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
```
It is possible to declare another main window object for use in the program, but the
\texttt{m\_pMainWnd} pointer is a pointer that has been inherited from \texttt{CWinApp}. It is already
prepared to handle the responsibility of a main window. The first line sets it up as a variable
of type \texttt{CMainWnd}. It should be noted that one of the properties of inheritance is that any
place a class type can be passed into can also accept an object of a type inherited from that
class. In other words, a pointer like \texttt{m\_pMainWnd} can be declared as a type \texttt{CFrameWnd}
and still be initialized to \texttt{CMainWnd} since \texttt{CMainWnd} is a descendent of \texttt{CFrameWnd}. The
same is true for passing arguments to or from a function.

The second line runs the \texttt{ShowWindow()} operation of the main window object. The
parameter passed to it is an integer field in \texttt{CTheApp}. It holds the initial state of the main
window, i.e. minimized, maximized, etc. The function sets the active state of the window in
this regard. Finally, the third line calls the function \texttt{UpdateWindow()} to redraw the window
using the newly set parameters.

\subsection*{3.2 Message Maps}

Another highly essential piece of information that is necessary to use and develop an
MFC program is the use of message maps. Everything that has been discussed so far is very
nice for creating and displaying a program, but this application must interact with Windows.
Windows is built intelligently in that any contact with the system takes place through the use
of messages. The system controls all input devices, and whenever the keyboard or mouse
enters any commands, they are processed in a global system queue. The first thing that
happens in an MFC program is that the application registers itself with the system and
receives a unique identification. The system queue then sends out messages telling all
registered applications what has been entered, pressed, etc.

The point of all this is that an application needs a way to interact with the system,
have a way to receive and react to system messages, and similar interactions. This is done
through the use of a message map. The message map works somewhat like a macro, and it is
rather complex in its implementation. Some brief analogies will be drawn here to help
illustrate the concept without delving into the specifics of the message map. It is also quite
particular in its syntax and implementation, but, fortunately, it is relatively short once the
rules are described.
Any class that can trace its ancestry back to a class called CCmdTarget can make use of message maps. This includes CWinApp, CFrameWnd, any dialog boxes, any view or document objects, OLE objects, and any controls. This is a pretty extensive list. There are more than 100 pre-defined message map macros, and it is possible to define new ones.

The simple reasoning behind the message map and the way it is constructed is that it is very unlikely that any particular program will make use of all 100+ macros. Therefore, it is desirable to only load the ones that an application needs. This saves the application paging through a list of macros that have no value. This offers greatly reduced overhead and memory space than if the objects had been set up to use virtual functions for the system message map. The application stores a type of dynamic array that lists all of the loaded message map macros. When an appropriate message comes in, it scans that short list and runs the corresponding code dictated by the message map to react to the message. Without getting too specific, some typical messages might be a left mouse button click, a right mouse button click, keyboard input, or the message to repaint the screen.

It is fortunately not necessary to understand much of the inner workings of the message macros to implement them. It is only necessary to understand that the system queue sends a message to the program telling it what has happened to the computer, and it is necessary for the application to decide if that message affects it. If the message does, it needs to take appropriate action.

Declaring the message map to exist begins in the declaration of the class. The declaration usually appears as the last line of the class declaration.

```c
DECLARE_MESSAGE_MAP()
```

Note that there is no semi-colon after this declaration. One should not appear here. This needs to be typed exactly as written, or the message map will fail.

The implementation of this message map is also very particular. The message map from Appendix A appears as follows:

```c
BEGIN_MESSAGE_MAP(CMainWnd, CFrameWnd)...
END_MESSAGE_MAP()
```
Appendix A has many lines in between, but this is the basic implementation of a message map for any application. Inside the parenthesis is the class to which this particular message map is attached (CMainWnd in this case), followed by the name of the class from which the one in question is derived (CFrameWnd). This second piece of information is required to identify the message map of the parent class, since message maps are also properties of inheritance. In other words, the compiler finds the parent message map, and transparently adds its contents to the macro list.

By similarity, creating a message map for the application object would require passing the names of CTheApp followed by CWinApp (the parent class). Creating a message map for one of the dialog box classes would require the class name of that particular dialog box, followed by CDialog. Similar parallels can be made for any object descended from CCmdTarget.

There are a great variety of macros set up that perform all sorts of tasks. Most of them will obviously not be covered here, but a flavor of a few common ones will be given with the understanding that most of them operate in a similar manner. Certainly one of the most useful is the one that tells the application that a command has been selected. This is how the application ties together the selection from a menu bar, for example, with the action that it performs in response. In fact, another way to think of these message maps is as a way to tie together a cause and an effect in Windows.

The macro appears as follows inside the message map:

\[
\text{ON\_COMMAND}(\text{IDM\_NEW}, \text{OnNewData})
\]

This is only a sample taken from the Appendix A program. The keyword \textit{ON\_COMMAND()} is the macro name. The first parameter is a unique, defined constant. It is defined in the resource.h file, and is the message for a particular menu option from the pop-up menu along the top. This particular menu option happens to generate a new airfoil in the program. The second parameter indicates the function to call in response to this action of selecting \textit{New...} from the menu. These “response” functions are called message handlers. Note once more that the line of code has no semi-colon after it, which is true of all message map macros.
This function is a special function. The OnNewData() function is declared in the following way in Appendix A.

```c
afx_msg void OnNewData();
```

All of these response functions must take no input arguments of any kind. They are all of type `void`. They are also prefaced by the keyword `afx_msg`. Currently, this keyword actually does nothing. It is still common practice to use it, however, as it clearly designates the function as a message handler.

Windows also has a host of macros that respond directly to system messages. These all begin with the letters `ON_WM_`. These differ slightly from the command macros in that they have no arguments with them. Looking at the message map of Appendix A shows the following system macros:

```c
ON_WM_PAINT()
ON_WM_SIZE()
ON_WM_CREATE()
```

The way this actually works is that when a message is received of these particular macros, there are specifically named default functions that are called in response. While each individual macro should be looked up to insure accuracy, the message handlers are typically the same name without the `_WM_` letters in the middle (and in lower case).

For instance, the function corresponding to `ON_WM_PAINT()` is known as `OnPaint()`. `OnPaint()` is also declared in the `CMainWnd` declaration as an `afx_msg` function, but the other two rules in command macros do not necessarily apply here for Windows message handlers. Some of the handlers are of other return types than `void`, and some take on parameters.

In the case of `ON_WM_PAINT()`, the system sends out a message to all of its registered applications anytime anything on the screen changes. This is the signal to repaint the application window. When the `WM_PAINT` message comes in, the macro automatically calls `OnPaint()`, which should contain all the information necessary to regenerate the
window's looks. A glance at the WinFunc.cpp file reveals that this function is present, and is indeed responsible for things like redrawing the axes and the airfoil shape.

The other two defined macros call the \texttt{OnCreate()} and \texttt{OnSize()} message handlers. The \texttt{OnCreate()} function is called upon the window's creation. \texttt{OnSize()} is called every time the window is resized. Dynamic resizing is a basic function of all Windows frame windows. The \texttt{OnSize()} function sends a pair of integers indicating the new size of the window, which are used in scaling the airfoil and axes to fit in the frame.

The code in Appendix B reveals that it processes some additional Windows \texttt{WM} messages. In all, there are well over 100 separate messages that can all occur under different circumstances. Most of them are relatively logical and easy to figure out now that the basic methodology has been explained. Once again, it is useful to note that these two programs are only using a handful of the 100+ messages available, so it definitely makes sense not to have to check for all of them. It is much more efficient just to worry about the few necessary messages.

A quick note will be made of a few other messages that have not been used in either program. There is a message for every type of action that the mouse can take. In a standard PC, there are messages for left clicking, right clicking, left and right pressing, left and right releasing, left or right double-clicking, clicking both buttons at once, and other related messages. This host of messages is sent out every time one of those actions occurs. It is through these messages and their related handles that many programs utilize the mouse. For instance, the Microsoft Word this document has been typed in can highlight text based on pressing and not releasing the left mouse button. A menu also appears wherever the mouse is right clicked. These things and may others like it are all possible simply through these message macros.

### 3.3 Resources

The area of resources has been brought up in dealing with message maps, and it is an aspect of MFC programming that is worth exploring briefly. While there are many deeply rooted issues in dealing with Windows applications such as creating and destroying objects, registering application windows, etc., there are also many more mundane issues as well. For
example, consider a typical dialog box in a program. It is necessary to understand resources before the specific details of dialog boxes can be presented.

![Plotting Scale Dialog Box](image)

Figure 8. A Typical Dialog Box

It should be easy to guess by now that any dialog box is an object just like everything else in MFC. It is created and destroyed. However, there is a great deal of specific, visual detail that corresponds to this dialog box. The dialog box, for instance, is a specific number of pixels in size, has all of its buttons and various text items in specific locations, has certain colors, and possesses things like that as part of its properties. This would be almost impossible to guess at for a programmer. It would require many runs of specifying the parameters, running the program to see what the box actually looked like, then adjusting the parameters again.

For this reason, Visual C++ includes a graphical resource editor. Reference [11] is an excellent source for information on resources, as Zaratian has written a book dedicated almost exclusively to developing resources. Things like dialog boxes, menus, icons, and other repetitive, visual items like that can be designed graphically and automatically imported into the program code during compilation. It is much easier to draw the dialog box than to guess at all the parameters required for its display. The actual editor is much like a simple drawing program. Dialog boxes, popup menus, icons, accelerator keys, cursors, toolbars, and many other things are all a part of the resource editor. In the case of the dialog boxes, for example, the editor draws the physical dimensions of the box, and allows a programmer to insert buttons, text, and a host of other various controls called the Windows common controls. These common controls include things like sliders, radio buttons, check boxes, progress bars, and most of the other things that can be found in the typical dialog boxes of any application.
This graphical display saves its work in a file called a resource script. This file will always have an extension of \textit{rc}, and should be included in the project just as the actual code files are. Both Appendix A and Appendix B have resource script files in their projects. These files are scripts because they are not actual code. They just present all of the required information for the various resources in an easy to read list. Looking at one of these files, it is almost as easy to modify this file directly rather than loading it in the graphical editor. The spline.rc file of Appendix A was, in fact, built by a combination of the graphical editor and direct modification.

Looking at a typical dialog box from this file yields the following section of code:

\begin{verbatim}
SCALEBOX DIALOG DISCARDABLE 0, 0, 187, 64
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Plotting Scale"
FONT 8, "MS Sans Serif"
BEGIN
  LTEXT "Y/X scale factor:", IDC_STATIC, 33, 17, 50, 8
  EDITTEXT IDC_EDIT1, 101, 15, 40, 14, ES_AUTOHSCROLL
  DEFPUSHBUTTON "OK", IDOK, 21, 43, 50, 14
  PUSHBUTTON "Cancel", IDCANCEL, 104, 43, 50, 14
END
\end{verbatim}

This section of script produces the plot scale dialog box that is shown in Figure 8. The graphical editor will create this automatically, though the script file itself can be understood and modified with only a little practice. The box title and title caption is given, along with outer dimensions. The \textit{STYLE} section includes some of the Windows standard dialog formats.

The \textit{IDC_STATIC}, \textit{IDC_EDIT1}, \textit{IDOK}, and \textit{IDCANCEL} are defined constants that are the ID numbers of the various \textit{objects} that are part of the dialog box. These constants should be unique in that particular dialog box, and are used in identifying the various \textit{objects} in the actual program code. This will be discussed in a moment. The constants are defined in a file called resource.h. It does not matter what the constant values are, since the actual number will never be used. The graphical editor will, in fact, generate the resource.h file.
automatically if allowed to. For someone manually creating or modifying this file, the only thing to keep in mind is that the ID numbers should start at 100, since the Windows operating system uses 0 through 99. The ID numbers IDOK and IDCANCEL are defined in the MFC library files. The block of resource script defines the style and locations for the standard OK and Cancel pushbuttons, a line of static text, and the input box which takes user input. This box is called an edit box.

The typical menu can also be found in the spline.rc file of Appendix A. This menu begins with the menu name, and then begins naming each of the actual popup lists. The popup list gives the text that is to be displayed, followed by a list of the menu items that the popup menu gives as choices. Each of these menu items is linked to its corresponding ID number defined in the resource.h file.

It is simple to build professional and standard looking dialog boxes, menus, and other resources using the graphical editor, but it is still necessary to link those resources to the code. Fortunately, MFC provides virtually all of the implementation for doing so. The menu is the simplest to describe since there is only one main menu for the Airfoil Spline Program (Appendix A).

In the constructor for the main window of Appendix A, there is a single function from CFrameWnd that is called. This function, Create, is the function that actually does all the work in initializing and setting up the main Window. It takes a variety of parameters, the last of which names the active menu to use. This is the same menu that is defined in the resource script file, and the MFC library knows to look there for the description of this menu. The menu is actually an object just like everything else in MFC, so the Create function declares a CMenu object and sets it up with all of the parameters in the resource file. It is possible to directly create a CMenu object with a specified resource name as well, in the event that a program required several different menus to be loaded.

Dialog boxes work much the same way, simply specifying the resource name upon creation of the box, and letting MFC do all the work of initializing the objects involved. To continue using the same example, the plot scale dialog box is called when the application receives a message that the appropriate menu item has been selected. This is done through an ON_COMMAND message map macro as described before. The menu has a resource ID
called *IDM_SCALE* that identifies that particular menu item. The macro runs an *afx_msg* function called *OnScaleData()* when this message comes in.

The body of this function contains only this:

```cpp
CScaleDataDialog dlgScaleData(this);
if (dlgScaleData.DoModal()==IDOK)
{
    InvalidateRect(NULL,TRUE);
    UpdateWindow();
}
```

The first line declares a new *object* of the appropriate type. Its only argument is a keyword known as *this*. The *this* keyword is a pointer to whatever the current window is. The dialog box needs to know who its parent window is, which would be the main window in the case of this dialog box. After creation, a dialog box is run using the function called *DoModal()* which handles all of the dialog workings. This function is an integer function, which returns a code depending on whether the user clicked *OK* or *Cancel* to exit the dialog box. If the user pressed *OK*, further instructions are carried out which just removes the dialog box, and updates the background window.

A look at the actual dialog box *class* itself reveals that the default constructor carries a parameter that names the resource script to use in building the dialog box properties. The CDlg class constructor does all the work in creating and implementing the *objects* necessary for the dialog box. It should be noted that all of the buttons, the text, and the edit box are all *objects* – MFC just cares for them internally.

There is not much else to the *class*. The only other function is a *virtual* function called *DoDataExchange()* . This is a relatively important function in dealing with dialog boxes. It is very important never to call this function directly. *DoModal()* leads to two internal calls to this function. The first call is immediately upon creation of the dialog box, before it is displayed to the screen. The second occurs only if the user presses *OK* (not *Cancel*).

The idea behind data exchange is that most of the various control *objects* in a dialog box are meant to have user input in them. For the plot scale box, there is an edit box control
for the user to type in number for the scale he or she wants. This data needs to be retrieved from the edit control and stored in a permanent variable for use by the program. Once again, MFC steps in to take care of virtually all of the implementation of this (which is actually relatively complex). A look at the body of the function yields the following:

```cpp
void CScaleDataDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);

    DDX_Text(pDX, IDC_EDIT1, scalefactor);
}
```

The first thing to note is that the first line calls the ancestor `class` version of the function. This is important to remember to include for any use of this function. Then, the edit box data is retrieved in one line using `DDX_Text`. The second two parameters are just the ID number of the box – which is why it was stated before that it is so important to have a unique ID number – and the name of the variable to store to.

The first parameter is the worker of the whole data exchange process. The object `pDX` is what makes everything possible, but is not used for anything by the programmer. A programmer needs only to pass along the object pointer to the ancestor `class` function and to any of the `DDX` functions utilized. There are a variety of other `DDX` functions which can be used. The one above is for edit boxes, but there is one for radio buttons, check boxes, list boxes, and any of the other common controls.

Although not used in either Appendix A or B, there is a cousin to the `DDX` function. There exists a similar set of `DDV` functions which are called validation functions. With these, it is possible to constrain an input in some way. For example, the above scale control could be constrained to be only positive.

The reason that the data exchange function is called twice is that it actually imports data initially, so that a dialog box will show initial values or the previously set values as it comes into existence. Then, if and only if `OK` is pressed that data will be exported back out to reflect the changes made by the user. Data exchange is a two way communication.
The discussion of dialog boxes will close with a brief look at the structure of the DataRead program, or Appendix B. In this application, the main window is not a frame window, but a dialog box itself. The entire process is similar to what has been discussed previously, with a few minor differences. The CWinApp object is still the same as before, but notice that the m_pMainWnd pointer is set to the address of a dialog box. This dialog box represents the main window of this particular program. The declaration of this dialog box uses the default argument of NULL, indicating that it has no parent window. Other than that, this dialog box operates much like any normal dialog box, and it even has its own message map to handle the pull down menu attached to it and the two buttons inside that call a file dialog box.

3.4 Wizards

This is a perfect time to introduce the App Wizard feature in Microsoft Visual Studio, which generates a great deal of the commonly used code to begin a project. Appendix B was made starting with an App Wizard project, while Appendix A was not. App Wizard generates a great deal of code, which can be difficult to understand for a beginning MFC programmer. Once beyond the basics of MFC, however, it can be very helpful.

In comparing Appendix A and B, it is easy to notice that the App Wizard generates code that is more completed and robust. This can be advantageous since the App Wizard code is proven and guaranteed to contain no errors. It provides a stable base to begin a new application. Of course, the project is a Win32 executable.

There is obviously a great deal more to using MFC than has been presented here. This thesis has made no mention of the document/view architecture of MFC. The idea behind this major feature is illustrated in the writing of this report. The entire thesis has been typed in Microsoft Word, and is a document. Only a piece of it can be displayed on the screen at a time though, which comprises the view of the document. A view can change without changing the document, and a document can change without changing the view. The two are extremely intertwined, but distinctive, separate entities. Since the document/view architecture was not implemented for either Appendix A or B, no discussion will be given on this topic.
There has also been no mention of OLE, which is another advanced feature of MFC. MFC provides *classes* to manage most features of OLE and Active X, which generically refer to the ways in which two programs can communicate with each other. OLE standardization is a good reason to program with MFC.

Both of the appendices made use of the code from an existing console program in C++ before enhancing the code with MFC and Windows application functionality. For this reason, not all of the MFC features were utilized. Much of the code was already there, and MFC was just used to wrap and interface around it. Using MFC from the beginning to implement things like the document/view architecture or MFC file and variable types would create a much smoother application from the coding point of view.

This section has discussed only a few of the major *objects* to get someone started in the world of MFC. Most of the references listed at the end are quite helpful, and the reader is recommended to look at them for more information, but they all tend to get too complicated too quickly for someone who knows nothing about MFC. This section has been written in an effort to help understand the code of Appendix A and Appendix B.
4.0 Advanced Coding and Modularization

Learning the C++ programming language, including the MFC extension libraries, has been a major focal point of this thesis. Part of the goal has been to explore some of the issues in software development and engineering. This goal includes more than just the syntax associated with C++.

This section of research looks at the reusability of the XFOIL source code, which is an aerodynamics code written by Professor Mark Drela at the Massachusetts Institute of Technology. This code has been written in FORTRAN, and comprises about 10,000 lines of code. This code has been tested and debugged, and can be assumed to be accurate for analyzing low speed airfoils. To date, it has been mainly used on UNIX workstations, although the source code is transferable to PC FORTRAN compilers.

With the exploding growth of the Windows 98 and NT PC platforms, it would definitely be advantageous to have this code running under these operating systems. Re-writing XFOIL to compile in a C++ compiler with the MFC extension libraries would require a tremendous effort to reproduce the code, followed by an equally tremendous effort to debug the resulting routines. Fortunately, Microsoft Visual Studio has provided the tools necessary to integrate the existing FORTRAN code with a powerful, new MFC interface complete with all of the features of Windows. References [10] and [12] have provided information about the syntax in combining these two languages through DLL loading.

The interest in this possibility has expanded the scope of this thesis to include a brief look at the more general area of code modularization and communications between different programming languages and between different application modules. This section will discuss some of the main points of interest that have been realized within this area.

One of the most interesting features of Windows is something known as dynamic linking. This feature is implemented through the use of a dynamic link library, or DLL file. The dynamic link library provides several functions for Windows applications. This thesis is not meant to be an extensive guide to DLL’s, but they need to be introduced as a solution to the problem of integrating different applications in different programming languages.

This DLL file provides code in a module that is separate from any executable. Looking at the entire Windows scheme, the code and operation of such an extensive
operating system is far too intense to load at once. Loading all of windows in one big file would absorb far too much memory and resources to ever use it for anything. To save on valuable resources, Windows resorts to dynamically loading pieces of code as they are needed, and then removing them from memory to make room for new things. This is the principle behind dynamic linking.

Another reason to dynamically link code is so that common functions and subroutines can be shared by many programs. There is no need to write, debug, or store them twice, since each application can simply load the community DLL file when it is needed.

A third, and more subtle reason for using dynamic link libraries can manifest itself in two ways. If an error is found in some code, or the code is simply updated to reflect new version changes, that part of the code must be recompiled and redistributed. If this code is contained in a DLL, only that DLL needs to be upgraded – not the whole program. This can make upgrading much easier.

A fourth reason for using DLLs is actually the reason that they come into the discussion when dealing with the FORTRAN code reusability issue. A DLL functions as a dynamic addition to a program. It is an already compiled module, which is why it earns its name of dynamic. Once a program, any program, is compiled, it exists and runs in machine code, regardless of what its source code language is. For this reason, it is possible to make a DLL written in any one language communicate flawlessly with a main application written in another language. Both modules are operating in machine code, so there is no discrepancy. Neither knows or needs to know what language the other was written in!

Adding a DLL dependency in Visual Studio makes the whole process of dynamic linking even easier. Visual Studio is structured so that everything happens under one global workspace. A new workspace allows a programmer to create multiple projects, and compile them together. A main application can be set to have a dependency on another project that will usually be a Win32 DLL project.

Dynamic linking in Windows is not particularly complex, but it still takes some care, for the library must be loaded and scanned for the functions or subroutines before they can be used. Visual Studio does all of this loading and linking automatically, so a series of projects created and linked using the project dependency method requires almost zero work input from the programmer to run seamlessly. Both Appendix A and B make use of this Visual
Studio feature, so this is the method that will be discussed and encouraged for the purposes of this paper.

Implementing this method requires only a single line of code on each end of the calling exchange to identify a function as a candidate for dynamic linking. Appendix A illustrates and example with a C++ main application calling a FORTRAN DLL, while Appendix B illustrates an example of a C++ main application calling a C++ DLL.

Physically setting up the projects in C++ is conveniently handled through dialog box options in the project creation. First the main application is made in a project of its own in the fresh workspace. After it is set up with all the usual consideration, a second project is created. Although the New... box will default to a new file, it is possible to move over to the project tab for adding a second project. In the section where the project name is placed, there is a check box that says "dependency of." This should be checked, and the first project named as the dependent in the space provided. This is all that is necessary to link the two project files, and the setup should be evident in the file view of the main project. When the main program is compiled, Visual Studio will automatically compile the projects it is dependent on, and make the necessary dynamic and static links automatically. Since Visual Studio has a variety of language compilers that all run under the same global engine, projects from any of the installed compilers can be added to the workspace. Visual Studio offers C++, Visual Basic, FoxPro, J++, and FORTRAN all as either included or optional modules to its studio.

Appendix A was built with a C++ Win32 main application project with a dependency on a FORTRAN Win32 DLL project. Appendix B was built using the MFC AppWizard (a Win32 application) with a dependency on a Win32 C++ DLL project. The actual code that went into the projects will also be looked at briefly. These two examples should be sufficient to allow a reader to begin building DLL dependent projects in this fashion. Note that most of the FORTRAN code has been left out of Appendix A. It is extremely lengthy, and since it was already an existing code, only the modified section has been placed in Appendix A to help understand the DLL process. Other than what is shown there, absolutely no other modifications were made to the XFOIL code to build the DLL.

In the case of Appendix A, it was necessary to build a FORTRAN DLL. All DLLs need an entry point which tells the machine where to find each function or subroutine when it
is needed. It is therefore necessary to flag each subroutine of a DLL that will be accessed by an outside application. This is done in the FORTRAN source file through a single line:

!DEC$ ATTRIBUTES :: XFOIL

This declaration begins on the first column of the line, which indicates that it is a comment to the FORTRAN code. The linker, however, picks it up as a special comment, and flags XFOIL() as a function that can be accessed from the main program. Everything else automatically compiles, and XFOIL() will receive a call to it, just as if part of the FORTRAN code did so. Note that there is no main function in the FORTRAN DLL. There does not need to be, since the program will never independently run. It only receives calls and is loaded into the memory space of the parent program. All of the parameters passed to this subroutine are floating point variables, with the exception of the number of points, which is an integer. This must be consistent between the calling program and the DLL subroutine program receiving the call.

Implementing this from the C++ end just takes one additional line of code as well. That subroutine is declared in the following way, in the matrix.h file:

extern "C" __declspec(dllexport) void _stdcall XFOIL(int *nact, float *xdata, float *ydata, float *alpha, float *reyno, float *amach, float *acl, float *acd, float *acm, float *airor);

This declaration allows the XFOIL() subroutine to be used throughout the code as any normal C++ function would be used. The keywords preceding the function name can be copied exactly to work with any FORTRAN subroutine, not just XFOIL(). They are explained briefly for the sake of completion. Notice that every one of the arguments to the function are declared as pointers. This is very important in merging FORTRAN and C++. C++ has the ability to deal both in terms of a variable or a pointer. Everything in FORTRAN is a pointer. All variables, regardless of condition, are created by reference. For this reason, anything that C++ passes must be in the form of a pointer. When this function is actually
called in the WinFunc.cpp file, it is easy to see that all arguments are references rather than the actual variable.

Arrays present no problems in this type of arrangement. FORTRAN indexes arrays just as C++ does. Both languages use a pointer to the first cell of the array, and index from there using pointer arithmetic. The arrays are passed by reference just as the single variables are.

The keyword extern in C++ is used for other things besides calling DLL functions. Since most C++ programs contain multiple files, this keyword just provides a check to prevent a programmer from declaring something redundantly. If a variable or function is declared in another file, the linker generates an error on compilation. Declaring the variable or function as extern simply says to the linker that the variable is to be used in this file, but is declared somewhere else. This keyword is appropriate whether that function is in another file or another project all together.

The "C" extension immediately following prevents something called C++ name-mangling. This is a rather obscure entity that lies deep within the compiler routines. Exactly what this is has been somewhat unclear, but the basic idea is that the compiler actually renames functions when compiling C++. This name modification is consistent throughout C++, so it affects nothing between C++ files, but FORTRAN does not do this. The extension keeps the C+ compiler from doing this either so that the two function names are consistent.

The next pair of keywords are what is responsible for importing the function from the DLL. The keywords dllimport and dllexport are called storage-class modifiers. They define the interface between the DLL and its caller, making the interface standard and invisible to the programmer. These keywords are Microsoft creations, and not part of the standard C++ language. For this reason they are wrapped with the __declspec keyword which converts all Microsoft storage-class additions to standard C++.

Following this is the return type of the function, which is void. FORTRAN deals exclusively in void type, since its parameters are all passed by reference. The stdcall keyword indicates that the call is to be made using universal standards for Win32 API. That is all there is to linking C++ code and FORTRAN code in Appendix A.

One note that should be made concerning the process of building a FORTRAN DLL was the cause of a series of problems in developing the Airfoil Spline Program. As a DLL,
XFOIL is called inside the memory space of its parent program. Since the Airfoil Spline Program is an MFC program, there is no console screen. The DLL function, therefore, has no ability to output text to a DOS screen. When it was initially built, XFOIL contained many statements of the following general types:

```plaintext
print *,' \\
write (*,*)' \\
```

These statement types cause errors in the program execution, since the subroutine has only restricted console access. Any FORTRAN code built into a DLL for Windows must not contain these types of statements.

Appendix B operates even more simply, since it communicates between modules of the same source language.

```plaintext
extern __declspec(dllexport)
int DataRead (CString infilenm, CString outfilenm)
```

The datadll.cpp file contains only the single function definition and nothing more. Its declaration is given here, utilizing many of the same keywords that were described before. Here only the storage-class modifier has been used set up the interface inside the DLL for exporting to the main program. The standardization keywords are not needed here since the calling language is the same as the provider language.

A look at the main program call to the DLL shows the following declaration:

```plaintext
extern __declspec(dllimport) DataRead(CString infilenm, 
CString outfilenm);
```

This side of the call looks much the same, and has been described above already. The function `DataRead()` is used as any normal C++ function is. There is no need to pass only pointers since FORTRAN is not involved in this application. Visual Studio does all the rest
of the work for the programmer, making this type of algorithm very powerful, effective, and efficient.

In looking at this issue of program integration, another area that deserves to be mentioned; this is the area of OLE and Active X programming, which is a concept developed by Microsoft for use in Windows. OLE stands for object linking and embedding. OLE and Active X are two words describing aspects of a broad area with involves the ability of one program to communicate with or control the actions of another program for the purpose of data exchange. OLE and Active X is one of the foremost areas of development by Microsoft for Windows. They have decided that this area is worth some serious developmental energy, and see it as part of the future of the PC platform. For this reason, OLE will certainly be an important part of software development and use in the near future.

OLE and Active X are relatively vague terms, as the definition given above actually encompasses both words. Exactly what operations lie within the Active X field, and which lie within the OLE field are typically unclear. Regardless, this is an area that can be quite beneficial to the development that has driven this thesis. OLE looks promising for areas like multidisciplinary design, integrating application components, and more.

OLE provides the ability for one program to control, or automate, another. It also provides a properly set up program to integrate foreign objects or documents into its native documents. This report is a good example of OLE controls, as it contains many pictures, Excel spreadsheets, tables, etc. All of these things are part of the world of OLE.

OLE consists of two parts: a client side, and a server side. The client is what initiates the interaction and makes a specific request. The server is what responds to the client. From a programming standpoint, the client side is much easier to deal with.

OLE and Active X were investigated as possibilities to connect application modules together. The idea behind this was to use the software in Appendix A to control a program like Matlab and use its graphing and matrix calculation power. The problem arose in the complexity of OLE. Visual C++ is a fully OLE capable language, and MFC actually provides objects to make the OLE interface much easier. Unfortunately, OLE is still a very complex area of programming, and a great deal of effort would have been required to develop a stable OLE client to control Matlab. Reference [13] is a manual provided with
Matlab that details the server code set up by the Matlab programmers for use in automating it with an outside client.

For future reference to readers, it was discovered that the object in MFC used to control the client side of OLE is known as COleDispatchDriver. This object contains all the members required to build the interface. A recommended future development to the Spline Airfoil Program would be to include OLE automation algorithms to more fully integrate it into Windows.

Both OLE and dynamic linking are certainly part of the future of the Microsoft Windows operating system. Microsoft seems interested in developing ever more modular and generic objects for Windows to increase its flexibility and speed. Microsoft is driving toward tiny independent modules that dynamically work together to perform a required task using as few system resources as possible. These areas are both worth studying in detail to be a part of the future of computing.
5.0 A description of Programming Algorithms

The previous sections have described many of the developments that have gone into the creation of the software in the Appendices. For practical reasons, this thesis has assumed a general understanding of C++ from the reader, but the preceding five sections have strived to build on that assumed knowledge. This section seeks only to help explain the general algorithms used in Appendix A. One problem with object oriented programming is that it tends to be very non-linear in its execution. The program jumps from function to function and object to object. This can make understanding a complex program a difficult task indeed, even for someone skilled in C++.

This section will contain very little actual code or code description. Instead, it will look at the more general flow of operation of the code. Readers are referred to the preceding sections and to the attached reference list for information on the specific code syntax and programming methods. With that said, it is time to take a brief look at Appendix A.

In Appendix A, the Spline Airfoil Program makes use of the MFC library in typical fashion. There is a CWinApp derived object to run the application framework. Since it is of the common frame window variety of application, it also makes use of a CFrameWnd object. The main window class declares and runs the message map macro to process menu and system commands. The declaration and implementation for both of these classes can be found in the pair of WinFunc files. Declarations and definitions for the various dialog boxes can be found in the WinDlg files. Since this application uses a relatively basic MFC interface, readers are once again referred to the preceding section detailing MFC for assistance in understanding the algorithms of MFC.

What is of more interest to this chapter is the internal working of the Airfoil class. This class is quite complicated and extensive, all of which is defined in the spline.h file. The OnNewData() function of the CMainWnd class is what initializes the airfoil. The object is declared globally as a pointer. There is only the single pointer currentairfoil, so there can be only one airfoil in active memory at a time. If a new airfoil is desired, the old one is simply overwritten.
Dynamic construction of the airfoil can take place through three different constructors. There is also a default constructor, but that is never used by the program. Before looking at these three constructors, it is necessary to examine the two major data fields of Airfoil, which are the topspline and botspline, both objects of the Bspline class. The Airfoil class uses two splines to store its surface data -- one for the top surface, and one for the bottom surface.

The Bspline class is inherited from the class of Base_spline. Base_spline provides only the framework for a spline curve. It has the ability to store both the \( n \) and \( k \) parameters as well as the knot vector. It also stores the number of data points required, although it is never capable of calculating an actual data point. It uses this information to generate the \( N \) matrix based on regular intervals of \( t \) for the number of data points specified.

Bspline inherits this ability, and adds the operations and data fields for the polygon points to the spline. Since it inherits the ability to calculate the \( N \) matrix from its ancestor, it only needs these polygon values to calculate the resulting data of the curve via equation (6). Calculating data points is actually done dynamically. Bspline stores both the \( N \) matrix and the polygon point matrix statically, recalculating \( N \) only when something changes (like a knot vector term, for instance). It has a public class operation that gets a specific data point of the spline. When this is called, Bspline simply calculates that data point using the stored information rather than take up more memory by storing a data matrix statically.

Airfoil has two objects of this class is its primary data fields, setting parameters and calling for data as necessary. Airfoil has several public functions that are used by the MFC interface to set the various parameters to generate data points. Most of these functions are just hollow wrappers around the corresponding Bspline functions, with an extra parameter (called top_or_bottom in most cases) that specifies whether the parameter in question is supposed to go to the top surface spline or the bottom one.

Getting back to the constructors of Airfoil, there are once again three possibilities to generate a new object. These three choices are directly correlated to the three choices presented by the CNewDataDialog dialog box in the form of radio buttons. Radio buttons force the user to select only one of the options presented, precluding the possibility of multiple selections.
The first choice is a straight calculation of polygon points from an existing data file. It is assumed that this data file has been properly run through the Data Read program of Appendix B first to insure that it is in the proper format. If so, this constructor reads in the data and stores it in temporary arrays. It then proceeds with a matrix inversion using Equation (9). Independent functions for matrix operations including transpose, matrix multiplication, matrix determinant, and matrix inverse have been separately included in the global.cpp file. These functions are completely generic, which is why they are included in a separate file. They could all easily be copied to another project to perform their same respective functions there with the same confidence. All of these functions were independently tested away from the Spline Airfoil Program to insure their validity.

Using the functions mentioned, the constructor is able to direct the calculation of the polygon points step by step. The points calculated are then set, one by one, into the topspline and botspline objects. Since it is necessary to have \( N \) specific for the curve fitting process to begin, \( N \) is calculated using assumed evenly distributed intervals of \( t \) by another class called OnlyGetN_spline. This class is just a skeleton framework over the Base_spline class, which it inherits from. Bspline was not a good choice to simply calculate \( N \) because it has extra data fields and operations that take up memory and resources. The OnlyGetN_spline has been stripped down as much as possible to conserve space. For this same reason, the Airfoil constructor makes a check to see if the number of data points read in from the top surface equals the number of data points read in from the bottom surface. If this is the case, then \( N \) will be the same for both surfaces. Making this check saves the computer from calculating the matrix twice, cutting out a significant portion of its work if that is the case.

The second choice of creating an airfoil was, by far, the easiest to implement. It involves the polygon points being directly input by the user for both the upper and lower surfaces. This Airfoil constructor simply sets the polygon vertices directly, based on the two arrays of values it has been given by the MFC interface. Unfortunately, the easiest method to code is also the method that is most rarely used. Practically, it is very difficult to pick the polygon vertices precisely to result in the creation of a smooth airfoil. The only practical use of this option would be to recreate an airfoil that has already been developed inside the program.
The third option for creating a new *Airfoil object* was actually an afterthought in the process of making the Spline Airfoil Program. After creating the first constructor for the *Airfoil class*, it was discovered that the XFOIL program is very sensitive to the data it receives. The slightest inconsistency in the data it receives can cause it to fail and halt program execution. Section 2.0 attempted to allude to the fact that the matrix inversion method provides an exact solution of the polygon points from the data provided; while this is a highly useful method of curve fitting, it has its limitations in that the end result is only as good as the data it receives. Because of rounding, and other factors, the data files do not always produce a spline that begins exactly at the point (0,0) and ends at the point (1,0). To run XFOIL, it usually seems necessary to force this to occur.

This third constructor was implemented to force the two end polygon points to the values of (0,0) and (1,0) respectively. This is actually not significantly different from the first method of initializing an airfoil. Setting the first and last polygon points to specific values means that the first and last columns of the \( N \) matrix are multiplied by known values rather than unknown variables. These products are known, so they are subtracted from the data matrix on the other side of the equation. A typical row will look like Equation (10) for this method of initialization:

\[ \sum_{i=2}^{n} N_{1,k} \cdot B_{i} = P(t) - N_{1,k} \cdot B_{1} - N_{n+1,k} \cdot B_{n+1} \]

(10)

After recalculating the right side of the equation, this set of simultaneous equations is solved with \( n-1 \) unknowns in both the \( x \) and the \( y \) coordinates. The previously mentioned matrix algorithms are again implemented to calculate the remaining unknowns by the matrix inversion method.

This method is the most effective, for it produces a result that XFOIL can readily accept as input. This method successfully forces the two extreme polygon vertices to their desired respective values. The use of this method helps to allow the execution of the *OnXFoilData() afx_msg* macro function of the *CMainWnd object*. This function makes the call to XFOIL, and retrieves the resulting aerodynamic coefficients.
This summarizes the basic flow of the program of Appendix A, which has demonstrated the basic ability to utilize existing FORTRAN code. This small section explains the process of generating an airfoil through the Airfoil Spline Program.
6.0 Results

In the abstract, three general objectives were defined. These goals were somewhat broad and encompassing. The code listed in Appendix A and B achieves those three objectives. This section focuses on some more detailed results, striving to quantitatively identify some of the strengths, weaknesses, and validations of the software.

6.1 The Software

First, this thesis has spent a great deal of pages discussing the methods used to create the two program packages, their functional capabilities, their architecture, and much about their development stages. It now seems appropriate to offer a brief visual look at those applications. The simpler of these two is the Data Read program (Appendix B). It is a dialog-based application, so its main window looks like the following:

![Data Organization Utility](image)

Figure 9. Data Read Utility Program (main window)

Its operation is simple, in that two files are specified -- an existing input file, and a new output file. This application simply converts the data format over for use by the main program. The path and filenames can either be typed into the box directly, or can be selected through the browse buttons on the right. The browse buttons bring up the standard Windows 95/NT Open or Save As file dialog boxes as shown in Figure 10.
These file selection boxes are pre-built objects in the MFC library, and can be set with many options like initial directory, default extensions, etc. The Save As dialog box even prompts for confirmation before an existing file is overwritten.

The Airfoil Spline Program (Appendix A) runs in a standard Windows frame window, looking like the following:
The image shown in Figure 11 shows the main window, with the menu across the top. The open dialog box shown is how the terms of the knot vector are input. As the knot vector is one of the things that must be known to deal with splines, it can be specified from the Data menu prior to importing any data, and it also appears as part of the chain of dialog boxes during a new airfoil selection.

An airfoil file that has been sorted through the Data Read program can be imported under the file menu, from which the import menu item offers the following three options.

The radio buttons in the dialog box of Figure 12 preclude a user from selecting more than one method. The following series of dialog boxes depend on the method selected from Figure 12. If importing from a file, the standard Windows Open file dialog box appears as in Figure 10. Of course, when exporting the spline data to a file, another similar Save As dialog box is also used. The other parameters needed to calculate a spline -- namely $n$, $k$, the knot vector terms, and the number of data points to generate -- are collected in this series of dialog boxes.

The result is that airfoil data points are read in, or the polygon points are inserted directly, and a spline is generated. A useful feature of the program is that it immediately draws the resulting airfoil spline in the window. This is only a basic plot, but gives the user an immediate sense of the airfoil. A more sophisticated program like Matlab or Excel draws better curves, but that requires exporting the data to a separate file.
Figure 13 shows the quick view plot of a 0012 airfoil. A plot scale dialog box in the File menu changes the scale ratio of Y to X, which is 2 by default. A user can change this to anything if he or she wishes to exaggerate the airfoil geometry in a different ratio on the screen.

Figure 13. Spline Airfoil Program with N0012 airfoil active

With an airfoil active, the user has the option to export the data or the current list of polygon points to an ASCII text file using the usual Save As dialog boxes. These options can be found in the File menu. The Data menu offers information and the ability to change the polygon points. The user can manually change them to change the shape of the airfoil, and experiment with different polygon configurations. The change is reflected immediately on the screen. The knot vector box is also located in this menu, but a new airfoil must be created before any changes to the knot vector will be displayed.
The polygon point boxes look like the following Figure 14. With all of these figures, it should be noticeable by now that most of the dialog boxes look basically the same. This is the standardization feature that Windows offers. Using standard Windows dialog boxes provides a less confusing and more professional look to the software than if the dialog boxes were haphazardly created.

![Figure 14. Typical Polygon Point Box.](image)

The Calculation menu offers information on the airfoil statistics (like n and k). Further, and most importantly, this menu is where XFOIL is called. It is a menu option. Above it is the menu to set the aerodynamic parameters that XFOIL uses. Mach number, angle of attack, and Reynolds' number are all set here. The program provides default values like most of the other dialog boxes. XFOIL does all of its calculations internally, and reports its values in a message box after the module execution is finished and returns to the main loop.

### 6.2 Airfoil Testing

The effectiveness of this program was tested using a selection of 5 airfoils, selected to represent a cross-section of typical performance. The original data was collected from the University of Illinois airfoil database [14]. The airfoils used are listed as follows, with the actual original data file name that it is stored under in the database at the time of this writing.
• NACA 0012 symmetric airfoil, n0012.dat
• NACA 2411 cambered airfoil, naca2411.dat
• Selig 1223 high lift airfoil, sl1223.dat
• Boeing 707 0.54 span airfoil, b707d.dat
• RAE NPL 5215 transonic airfoil, rae5215.dat

This selection of airfoils produced some interesting results in the tests that have been included below. Once again, the first objective of this thesis is to access the usefulness of generic B-spline curves in approximating the aerodynamic shape of airfoils for the purpose of future optimization of these airfoils with only a few controlling points.

For these tests, a fixed set of the aerodynamic parameters was chosen. These values happen to be:

\[ \alpha = 0^\circ \]
\[ \text{Re} = 5,000,000 \]
\[ M = 0.1 \]

Further, the number of data points for all airfoil runs was set at 150 points. This value is greater than that of any of the original data file, and, beyond a certain amount, the number of points had little effect on the aerodynamic performance.

With these constants, the value of \( n \) was varied to include \( n = 4, n = 6, \) and \( n = 8 \). The shape factor order, \( k \), was varied to include \( k = 2, k = 3, k = 4 \). For all of these tests, an open knot vector was used. For instance, in the case of \( n = 6, k = 3 \), the knot vector was defined as:

\[ \mathbf{x} = \{ 0 \ 0 \ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 5 \ 5 \} \]  

(11)

All knot vectors were defined consistently in this way. Before looking at some of the results of the tests, the results of the airfoil geometric approximations are presented below in Figures 15 through 19. Each figure shows the original data from the University of Illinois database, the interpolated data that the program produced for the respective case, and the resulting polygon control vertices, connected by a straight line. The original and spline data lies very
close together in all the cases. In the resolution of the graph, there is no distinguishable
difference between the two lines. There are two plots in all cases however

Figure 15. NACA 0012 Airfoil Comparison using k=3, n=6

Figure 16. NACA 2411 Airfoil Comparison using k=3, n=8
Figure 17. Selig 1223 Airfoil Comparison using $k=4,n=8$

Figure 18. Boeing 707 0.54 span Airfoil Comparison using $k=4,n=6$
From a visual perspective, the spline data virtually covers the original data in every case. This should be the case since the spline is a direct solution rather than an iterative curve fitting approach. The difficulty lies in the fact that the numerical aerodynamics algorithms in XFOIL are based on the highly non-linear, and highly sensitive Navier-Stokes equations coupled with an iterative integral method for boundary layer calculations. The pressure distribution and boundary layer development are highly dependent on the curvature and its derivative, particularly near the leading edge. What causes some convergence issues is that, while the spline can mimic the data point values, it is not designed to reproduce the slope. The local first and second order continuity between the original and reproduced data is an important factor.

Before going into any details about the results, the actual numbers obtained in this first experiment should be given. Table 1 through 5 summarize the results. Any cell with "DNC" printed in it refers to the fact that XFOIL did not converge to a solution with that particular set of data.
<table>
<thead>
<tr>
<th>Table 1 - NACA 0012 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2 - NACA 2411 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3 - Selig 1223 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4 - Boeing 707 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5 - RAEnPNU 5215 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
At first glance, this data looks unimpressive, as it is riddled with cases of no convergence. With this data, it appears that the B-splines are unable to simulate the aerodynamic performance of the airfoils. As Table 5 shows, the RAE airfoil did not converge at all for any case. In order to appreciate the significance of this first experiment, however, it is necessary to look at a second experiment that was performed.

This second test involved selecting a representative case from the first experiment. The eighth case of \( n = 8 \) and \( k = 3 \) was relatively successful with the exception of Tables 4 and 5. It was selected for this second test, and held fixed to though values throughout the test. This second test examined the influence of the knot vector on the splined data points. Two representative knot vectors were selected, one weighed heavily toward the beginning of the data range, and one heavily toward the end. These two cases are listed in equation (12) and (13) respectively:

\[
\mathbf{\xi} = \{0 \ 0 \ 0 \ 0.5 \ 1 \ 1.5 \ 2.5 \ 3.5 \ 5 \ 7 \ 7 \ 7\} \quad (12)
\]

\[
\mathbf{\xi} = \{0 \ 0 \ 0 \ 2 \ 3.5 \ 4.5 \ 5.5 \ 6 \ 6.5 \ 7 \ 7 \ 7 \} \quad (13)
\]

<table>
<thead>
<tr>
<th>Table 6 - NACA 0012 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 7 - NACA 2411 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 8 - Selig 1223 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 9 - Boeing 707 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10 - RAE(NPL) S215 Airfoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
The last airfoil still causes the same problems — no convergence. However, by trying different combinations of knot vectors, trial and error reveals that not all of them have this problem. Fixing all parameters as they were in the second test, the knot vector of equation (14) yields a solution from XFOIL.

\[ \mathbf{x} = \{ 0 \ 0 \ 0 \ 0.75 \ 2 \ 2.25 \ 3.5 \ 4.75 \ 6 \ 7 \ 7 \ 7 \} \]  

Table 11 - RAE(NPL) 5215 Airfoil

<table>
<thead>
<tr>
<th>knot eqn</th>
<th>cl</th>
<th>%diff cl</th>
<th>cd</th>
<th>%diff cd</th>
<th>cm</th>
<th>%diff cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0.2953</td>
<td>-</td>
<td>0.00513</td>
<td>-</td>
<td>-0.072</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>0.2885</td>
<td>2.30</td>
<td>0.00877</td>
<td>33.92</td>
<td>-0.0699</td>
<td>2.92</td>
</tr>
</tbody>
</table>

Aerodynamically, these values are definitely close to the expected values. And, more importantly, Table 11 shows that this case converged, while all previous cases of this airfoil did not. In going from Table 10 to 11, only the knot vector was changed — and that made the difference between an unstable solution and a stable one. The point can be further proved by looking at the other data from Table 6 through Table 9. In the case of Table 8, equation (13) converged and equation (12) did not. In the case of Table 6 and Table 7, the two different knot vectors produced dramatically different results, and all of these results are dissimilar from the corresponding case number eight from the first test.

What is to be learned or gained by this? Well, the research thus far has illustrated the importance of the knot vector, and that a good knot vector for one airfoil is definitely not necessarily a good choice for another airfoil. It is again important to realize that this process of using splines to approximate airfoils is one that interpolates rather than fits a best fit curve. This interpolation process lacks the flexibility of the other method in that it is forced into its definition by the choice of knot vector and by the requirement that it passes through the data points (i.e. is a solution to the system of equations for the data).

The next test performed involved selecting one of the more successful airfoils, and testing it at varying angles of attack to develop a cl vs. \( \alpha \) curve and a cd vs. \( \alpha \) curve. For this test, the NACA 2411 airfoil was tested. The Mach number was used as 0.1, and the Reynolds’ number was used as 2,000,000. The case of \( n = 8 \) and \( k = 3 \) was selected to run the test.
Figure 20. N2411 Airfoil with k=3, n=8

Figure 21. N2411 Airfoil with k=3, n=8
From Figure 20 and Figure 21, it is clear that B-splines are capable of emulating the aerodynamic performance of the airfoil. The cambered 2411 airfoil was used to push the envelope of accuracy of the spline more than the symmetric airfoil would have. Informally, the 0012 airfoil was tested at a few angles of attack, and its results were barely differentiable from the original. Here, in Figure 20 and Figure 21, some differences are noticeable, although the airfoil approximation is relatively consistent.

Finally, the airfoils were again tested using a Bezier configuration for the spline. Once again, this implies that the knot vector will consist of the first half of its terms equaling zero, the second half equaling one. Two cases were tested, the first for \( n=7, k=8 \) the second for \( n=11, k=12 \). The software package has had the maximum value of \( k \) set at 12. Even at 12, a Pentium II computer required 1 to 2 minutes to interpolate the spline from the original data. Exceeding this value of 12 becomes too computational intensive for a desktop PC to handle.

The two knot vectors for the cases tested are given in equations (15) and (16).

\[
\mathbf{\xi} = \{ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \}
\]  \hspace{1cm} (15)

\[
\mathbf{\xi} = \{ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \}
\]  \hspace{1cm} (16)

The results are given in Table 12. Only three airfoils have been listed, for the other two did not converge in either case listed.

<table>
<thead>
<tr>
<th>Table 12 - Bezier Curve Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>N0012</td>
</tr>
<tr>
<td>cl</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>n=7, k=8</td>
</tr>
<tr>
<td>n=11, k=12</td>
</tr>
<tr>
<td>original data</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

The original data is also included in Table 12 for ease of reference.
7.0 Conclusion/Recommendations

The data provides some interesting results; it is obvious by the tables of section 6.2 that B-splines do not always provide a good solution to aerodynamic performance using XFOIL. After looking at the situation, there are several recommendations to be made concerning the implementations of the B-spline curves to approximate airfoils.

One thing that stands out about the data collected is that it is not always consistent. In other words, raising the order of the curves or increasing the number of polygon points does not always create a better approximation.

Looking at the software code, the Airfoil class approximates its data by using two B-splines, one for the top surface, and one for the bottom surface. The Airfoil class was built this way because a conservative estimate was made about the ability of a spline to model a curve. Most airfoils have a highly rounded front edge. It was assumed, lacking any concrete information, that forcing a spline to model an edge with such a high curvature would compromise its interpolation values.

As the study proceeded, however, it became apparent from practical experience dealing with B-splines that they are capable of modeling such a curve with precision. Using two spline and forcing them to meet at the point (0,0) creates a discontinuity in the slope and the curvature at the leading edge. XFOIL is very sensitive to leading edge continuity, and the accuracy of this leading edge can greatly affect the aerodynamic performance of the airfoil. This seems to be supported by the fact that the NACA 0012 airfoil was the most successful in the tests performed. Its symmetry implies that the slope and curvature of the leading edge will be the same on both the top and the bottom. As the airfoils become increasing more cambered, their performance inside the software became more erratic.

Given this information, the software should be updated to utilize one smooth B-spline for the entire airfoil, going from trailing edge to trailing edge. This would require extensive modifications to the Airfoil class, but is recommended for further study in creating a future version of this software to greatly improve the reproducibility of data by the spline. One smooth curve would create continuity of a sufficiently high order automatically, to eliminate this as an error inducing factor.
This seems to be the single most important changes to be made, although it is further recommended that another brief study of the effects of knot vectors be made after the modifications are made. In section 6, knot vector selection seemed to have an effect on the aerodynamic characteristics, but different knot vectors will also result in a different curvature discrepancy at the leading edge. It is nearly impossible using the current software configuration, to separate the effects of these two items.

Of course, the software itself could be improved with a multitude of minor visual or interfacing changes. Most of these are minor, except that the curve fit interpolation could be greatly improved by changing the matrix solution method. Using even a simple solution method like Gauss-Jordan elimination would bring about a much quicker curve generation that the matrix inverse method. Calculating inverses is computational intensive, and a reduction of processor time would almost certainly be realized by eliminating the need to use inverses.

Despite the fact that the use of B-spline curves created discontinuity issues at the leading edge, this thesis has been successful at looking at the possibilities and issues involved in using B-splines. Much information has been gathered to make future generations of the software more successful. As for the other two goals, XFOIL has been successfully integrated into an MFC application, and the code of Appendices A and B represent the understanding of C++ and the MFC extensions that has been gained in working with the area of this thesis.
Reference List


    <www.dec.com/fortran/examples/vc-example1.html>.


    <http://amber.aae.uiuc.edu/~m-selig/ads.html>.
Appendix A: Airfoil Spline Program

Constant.h file:

#include <afxwin.h>
#include "constant.h"
#include "Resource.h"

Global.h file:
//
// global.h
//

#ifndef_SPLINE_GLOBAL_VAR
#define _SPLINE_GLOBAL_VAR

/*
*****
** Begin Global Variables.
*****
*/

CString mytitle = "Untitled";
int radio = -1;
int method = 0;
int m_cxClient, m_cyClient;
int globalflag = 0;
int globalaerorun = 0;
double scalefactor = 2.0;

CString globalimport;
CString globalexport;
CString globalCurDir = "C:\Windows\Desktop\";

Airfoil *currentairfoil;
double globalknot[Max_N + Max_K + 1];
int globaN = Max_N;
int globalk = Max_K;
int globalDataP = Max_Num_Points;

double alpha = 0;
double reynolds = 5000000;
double machnum = 0.25;

float cl = 0;
float cdrag = 0;
float cm = 0;
float error = 0;

#endif //_SPLINE_GLOBAL_VAR

Matrix.h file:

#include <string.h>
#include <fstream.h>
#include <iomanip.h>

// Matrix, h file:
#include <afxwin.h>
#include "resource.h"
#include "constant.h"

/**********************************************
// // matrix.h
//
/**********************************************

#define _SPLINE_MATRIX_H

double Determinant(double *p, int Size);
void InvertMatrix(double *inp, double *outp, int MatSize);
double CheckEven(int a, int b);
void GetData(CString location, double (*t)[2], double (*b)[2], int topnum, int botnum);
void ProvideCount(CString location, int *t, int *b);
int CheckCoordValue(int *coord);

int matmult(double *mat1, double *mat2, double *outmat, int Mat1RowSize, int Mat1ColSize, int Mat2RowSize, int Mat2ColSize);
void transpose (double *inmat, double *outmat, int inMatRowSize, int inMatColSize);

/**********************************************
** Begin FORTRAN external calls.          ****
**********************************************
extern "C" __declspec(dllimport) void _stdcall XFOIL(int *nact, float *xdata, float *ydata, float *alpha, float *reyno, float *amach, float *acl, float *acd, float *aem, float *airor);

#endif //_SPLINE_MATRIX_H

Matrix.cpp file:
#include "matrix.h"

////////////////////////////////////////
//
// matrix.cpp
//
// compiled July 15, 1999
// Matthew MacLean
// RIT master's thesis project
// Department of Mechanical Engineering
//
// defines the functions for matrix operations and
// general functions for the program
//
////////////////////////////////////////

//
// General functions
//

double Determinant (double *p, int Size)
{
    double ret=0;
    int a=0;
    double (*m);
    int row=0, col=0, Col=0;
    double *t;
    int cofact = 1;
    int m_Size = (Size-1) * (Size - 1);

    m = new double[m_Size];
    
    t = &(m[0]);

    if (Size==2)
        ret = (*p) * (*p+3) - (*p+1) * (*p+2);
    else
        {
            for (col=0; col<Size; col++)
                {
                    for (row=1; row<Size; row++)
                        {
                            for (Col=0; Col<Size; Col++)
                                {
                                    if (Col < col)
                                        *(t+(row-1)*(Size-1)+Col) = *(p+row * Size+Col);
                                    if (Col > col)
                                        *(t+(row-1)*(Size-1)+Col-1) = *(p+row * Size+Col);
                                }
                }
        }

    return ret;
}

ret = ret + *(p+col) * cofact * Determinant(t, (Size-1));

if (cofact == 1)
    cofact = -1;
else
    cofact = 1;

}
}
det = delete m;

return (ret);
}

//
double CheckEven(int a, int b)
{
    double ret=0;
    int checker=(a + b) % 2;

    if (checker == 0)
        ret = 1;
    else
        ret = -1;

    return(ret);
}

//
void InvertMatrix(double *inp, double *outp, int MatSize)
{
    int elemrow=0, elemcol=0;
    int row=0, col=0;
    int calcdet=0, output=0;

    double DetermInp=0;

double *det;
    double *m;

det = new double [((MatSize-1)*(MatSize-1))];
    m = new double [((MatSize*MatSize))];

for (elemrow=0; elemrow<MatSize; elemrow++)
{
    for (elemcol=0; elemcol<MatSize; elemcol++)
    {
        for (row=0; row<MatSize; row++)
        {
            if (row < elemrow)
\[
\begin{align*}
&\text{for } \text{col}=0; \text{col}<\text{MatSize}; \text{col}++ \\
&\quad \{ \\
&\quad \quad \text{if } \text{col} < \text{elemcol} \\
&\quad \quad \quad \quad *(\text{det} + \text{row} * (\text{MatSize}-1) + \text{col}) = *(\text{inp} + \\
&\quad \quad \quad \quad \text{row} \times \text{MatSize} + \text{col}); \\
&\quad \quad \text{if } \text{col} > \text{elemcol} \\
&\quad \quad \quad \quad *(\text{det} + \text{row} * (\text{MatSize}-1) + \text{col} - 1) = *(\text{inp} + \\
&\quad \quad \quad \quad \text{row} \times \text{MatSize} + \text{col}); \\
&\quad \quad \} \\
&\quad \\
&\text{if } \text{row} > \text{elemrow} \\
&\quad \{ \\
&\quad \quad \text{for } \text{col}=0; \text{col}<\text{MatSize}; \text{col}++ \\
&\quad \quad \quad \{ \\
&\quad \quad \quad \quad \text{if } \text{col} < \text{elemcol} \\
&\quad \quad \quad \quad \quad *(\text{det} + \text{row} - 1) * (\text{MatSize}-1) + \text{col} = \\
&\quad \quad \quad \quad \quad \text{row} \times \text{MatSize} + \text{col}; \\
&\quad \quad \quad \quad \quad \text{if } \text{col} > \text{elemcol} \\
&\quad \quad \quad \quad \quad \quad *(\text{det} + \text{row} - 1) * (\text{MatSize}-1) + \text{col} - 1 = \\
&\quad \quad \quad \quad \quad \quad \text{row} \times \text{MatSize} + \text{col}); \\
&\quad \quad \quad \quad \} \\
&\quad \} \\
&\} \\
&m + \text{elemrow} \times \text{MatSize} + \text{elemcol} = \text{CheckEven}(\text{elemrow, elemcol}) \times \\
\text{Determinant}(\text{det, (MatSize} - 1)); \\
\} \\
&\} \\
&\text{for } \text{calcdet}=0; \text{calcdet}<\text{MatSize}; \text{calcdet}++ \\
&\quad \text{DetermInp} = \text{DetermInp} + (*\text{inp} + \text{calcdet}) \times (*m + \text{calcdet}); \\
&\} \\
&\text{if } (\text{DetermInp} == 0) \\
&\quad \{ \\
&\quad \quad \text{for } \text{output}=0; \text{output}<\text{(MatSize} \times \text{MatSize}); \text{output}++ \\
&\quad \quad \quad *(\text{outp} + \text{output}) = 0; \\
&\quad \} \\
&\text{else} \\
&\quad \{ \\
&\quad \quad \text{for } \text{output}=0; \text{output}<\text{(MatSize} \times \text{MatSize}); \text{output}++ \\
&\quad \quad \quad *(\text{outp} + \text{output}) = (*\text{m} + \text{output}) / \text{DetermInp}; \\
&\quad \} \\
&\} \\
&\text{delete m;} \\
&\text{delete det;} \\
\}
\]

//
void ProvideCount(CString location, int *t, int *b)
{ 
    int botcount=0, topcount=0;
    int countflag=0;
    double d;
    
    ifstream infile(location);
    
    while (infile>>d) 
    { 
        if (d == -1) 
            countflag=1;
        else 
        { 
            infile>>d;
            if(countflag == 0) 
                topcount++;
            if(countflag == 1) 
                botcount++;
        } 
    } 
    (*b) = botcount;
    (*t) = topcount;
    infile.close();
}

// void GetData(CString location, double (*t)[2], double (*b)[2], int topnum, int botnum)
{ 
    int botcount=0, topcount=0;
    double temp l=0;
    
    ifstream infile(location);
    
    for(topcount=0; topcount<topnum; topcount++) 
    { 
        infile >> t[topcount][0];
        infile >> t[topcount][1];
    } 
    infile >> temp l;
    
    for (botcount=0; botcount<botnum; botcount++) 
    { 
        infile >> b[botcount][0];
        infile >> b[botcount][1];
    } 
    infile.close();
}
int CheckCoordValue(int *coord)
{
    int ret = 0;

    if ((*coord)>(Num_Coordinates-1))
    {
        *coord = Num_Coordinates - 1;
        ret = 1;
    }

    return(ret);
}

int matmult(double *mat1, double *mat2, double *outmat, int Mat1RowSize, int Mat1ColSize, int Mat2RowSize, int Mat2ColSize)
{
    //matmult multiplies two matrices of arbitrary size.
    //for error checking, matmult returns 0 for success
    //1 for failure (if matrices cannot be multiplied)

    int row=0, col=0;
    int mult=0;
    double total=0;

    if (Mat1ColSize == Mat2RowSize)
    {
        for (row=0; row<Mat1RowSize; row++)
        {
            for (col=0; col<Mat2ColSize; col++)
            {
                for (mult=0; mult<Mat1ColSize; mult++)
                    total = total + (*(mat1 + row * Mat1ColSize + mult)) * (*(mat2 + mult * Mat2ColSize + col));

                *(outmat + row * Mat2ColSize + col) = total;
                total = 0;
            }
        }
        return(0);
    }
    else
        return(1);
}

void transpose (double *inmat, double *outmat, int inMatRowSize, int inMatColSize)
{
    int a=0, b=0;
for (a=0; a<inMatRowSize; a++)
{
    for (b=0; b<inMatColSize; b++)
        *(outmat + b * inMatRowSize + a) = *(immat + a * inMatColSize + b);
}

//
void GaussJordan(double *A, double *b, int unknowns)
{
    int row=0, col=0;
    int n=0;
    double factor=0.0;

    for (row=0; row<unknowns; row++)
    {
        for (col=(row+l); col<unknowns; col++)
        {
            *(A+unknowns * row + col) = *(A+unknowns * row + col) / *(A+unknowns * row + row);
        }
        *(b+row) = *(b+row) / *(A+unknowns * row + row);
        *(A+unknowns * row + row) = 1;

        for (n=(row+l); n<unknowns; n++)
        {
            if (*(A+unknowns * n + row) != 0)
            {
                factor = *(A+unknowns * n + row) / *(A+unknowns * n + row + row);

                for (col=(row); col<unknowns; col++)
                {
                    *(A+unknowns * n + col) = *(A+unknowns * n + col) - factor * *(A+unknowns * n + row + col));
                }
                *(b+n) = *(b+n) - factor * *(b+row));
            }
        }
    }
    for (n=0; n<row; n++)
    {
        if (*(A+unknowns * n + row) != 0)
        {
            factor = *(A+unknowns * n + row) / *(A+unknowns * n + row + row);

            for (col=(row); col<unknowns; col++)
            {
                *(A+unknowns * n + col) = *(A+unknowns * n + col) - factor * *(A+unknowns * n + row + col));
            }
            *(b+n) = *(b+n) - factor * *(b+row));
        }
    }
Spline.h file:

#include <string.h>
#include <fstream.h>
#include <iomanip.h>
#include <afxwin.h>
#include "resource.h"
#include "matrix.h"
#include "constant.h"

#include "spline.h"

#ifndef _SPLINE_H

#define _SPLINE_H

/*
 *****
 ** Begin class definition calls. **
 *****
*/

/*
 Bsplines have 2 parameters: n & k;
it must be supplied a knot vector with n+k+1 terms
and it must be given 3D polygon vertices numbering n+1.
*/

//This comment should be disregarded by any programmer using this code.
// It is meant for Lodor, who wished that it be placed here. To all those who look
// for this, call me for a dot.
//Phi Delta Theta is a precious thing. Treasure it always, and good luck.
//153
//

//defines Base_spline class
//
class Base_spline
public:
    void Set_xi (int i, double Xi);
    double Get_xi (int i) const;
Base_spline();
Base_spline(int init_n, int init_k, int init_num_points);
~Base_spline();

protected:
    //private data members
    int n;
    int k;
    int num_points;
    int flag;
    double *knot;
    double (*N_Mat)[Max_N+2];

    //private functions used internally to the class
    virtual void CalcN_Mat ()
    void CalcStep_N (int index, double tindex);
    double first (int i, double t);
    double formula (int i, int j, double (*Step_N)[2], double t);
};

//defines Bspline class - inherited from Base_spline
//
class Bspline : public Base_spline
{
    public:
    Bspline();
    Bspline(int init_n, int init_k, int init_num_points);
    Bspline(int init_n, int init_k, int init_num_points, double *init_knot);
    ~Bspline();
    double ReturnData(int Coord, int index);
    void Set_Poly (int i, double *M);
    double Get_Poly (int i, int Coord) const;
    int getn();
    int getk();
    int getnumpoints();

    protected:
    double (*Poly_Poirrts)[Num_Coordinates];
};

//defines OnlyGetN_spline class - used only for Airfoil curve fitting
//
class OnlyGetNspline: public Basespline {
    
    public:
    
    double GetNvalue(int nindex, int point);
    void CalcNnow();
    OnlyGetNspline();
    OnlyGetNspline(int init_n, int init_k, int init_num_points);
    OnlyGetNspline(int init_n, int init_k, int init_num_points, double *init_knot);
    ~OnlyGetNspline();
};

//
// defines Airfoil class
//

class Airfoil {
    
    public:
    
    Airfoil();
    Airfoil(int init_n, int init_k, int init_num_points, double *init_knot, double z_coord, CString location, BOOL method = TRUE);
    Airfoil(int init_n, int init_k, int init_num_points, double *init_knot, double z_coord, CString location, double (*init_P), BOOL method = TRUE);
    Airfoil(int init_n, int init_k, int init_num_points, double *init_knot, double z_coord, double *polytop, double *polybot);
    ~Airfoil();
    double GetDataPoint(int Coord, int index, int toporbottom);
    void WriteOutData(CString location);
    void SetName(CString name);
    CString GetName() const;
    void SetPolygon(double *Values, int index, int toporbottom);
    double GetPolygon(int Coord, int index, int toporbottom);

    //
    // private member functions
    //
    private:
    
    Bspline *topspline;
    Bspline *botspline;
    CString Name;
    double Z;

            void Curvefit(double *data_top, double *data_bot, int init_n, int init_k, double *init_knot, int toppoints, int botpoints, int top_or_bottom);
            void Calculation(double *Ncurvefit, double *curvefitdata, int numofpoints, int numofpolygons);
            void SetPolyPoints(double *curvefitdata, double *data, int numofpoints, int numofpolygons, int top_or_bottom);
            void MixedCurvefit(double *data_top, double *data_bot, int init_n, int init_k, double *init_knot, int toppoints, int botpoints, double *init_P, int top_or_bottom);
            void SetMixedPolyPoints(double *init_P, double *curvefitdata, double *data, int numofpoints, int numofpolygons, int top_or_bottom);


Spline.cpp file:

#include "spline.h"

////////////////////////////////////////////////////////////////////////////////
// spline.cpp
// compiled July 15, 1999
// Matthew MacLean
// RIT master's thesis project
// Department of Mechanical Engineering
//
// defines the class members and related functions for
// airfoil and spline classes
//
////////////////////////////////////////////////////////////////////////////////

/**
*****
** Begin class function calls. **
*****
*/

// Base_spline definitions

// general constructor - uses maximum values for inputs
Base_spline::Base_spline ()
{
    //initializes dynamic arrays for use.
    knot = new double [Max_N + Max_K + 2];
    N_Mat = new double [Max_Num_Points + 1][Max_N + 2];
    num_points = Max_Num_Points;
    n = Max_N;
    k = Max_K;
}

// constructor with specified bspline inputs
Base_spline::Base_spline (int init_n, int init_k, int init_num_points)
{
    //initializes dynamic arrays for use.
    if (init_k >= 2)
k = init_k;
else
    k = 2;
if (init_n >= (k-1))
    n = init_n;
else
    n = (k-1);
if (init_num_points >= (n+1))
    num_points = init_num_points;
else
    num_points = (n+1);
knot = new double [n + k + 2];
N_Mat = new double [num_points + 1][Max_N + 2];
}

// destructor returns dynamically allocated array memory
Base_spline::~Base_spline()
{
    delete knot;
    delete [] N_Mat;
}

// double Base_spline::formula (int i, int j, double (*Step_N)[2], double t)
{
    double Ret=0;
    double P1=0, P2=0;
    if (((Step_N[i][1] != 0) && ((knot[i+j] - knot[i]) != 0))
        P1 = ((t - knot[i]) * Step_N[i][1] / (knot[i+j] - knot[i]));
    else
        P1 = 0;
    if (((Step_N[i+1][1] != 0) && ((knot[i+j] - knot[i+1]) != 0))
        P2 = ((knot[i+j] - t) * Step_N[i+1][1] / (knot[i+j] - knot[i+1]));
    else
        P2 = 0;
    Ret = P1 + P2;
    return(Ret);
}

// double Base_spline::first (int i, double t)
{
    double Ret=0;
    int c=0;
    if ((t >= knot[i]) && (t < knot[i+1]))
        Ret = 1;
else
    Ret = 0;

if ((t == knot[n+k+1]) && (knot[i]<knot[i+1]))
{
    Ret=1;
    for (c=i+1; c<=(n+k+1); c++)
    {
        if(knot[c]!=knot[i+1])
            Ret=0;
            break;
    }
}

return(Ret);

//
void Base_spline::CalcN_Mat()
{
    double dt = ((knot[n+k+1] - knot[1])/(num_points - 1));
    double tindex = knot[1];
    int z=0;

    for (z=1; z<=num_points; z++)
    {
        if (z == num_points)
            tindex = knot[n+k+1];

        CalcStep_N(z, tindex);
        tindex = tindex + dt;
    }

    flag = 1;
}
//
void Base_spline::CalcStep_N(int index, double tindex)
{
    int i=0, j=0;
    double (*Step_N)[2];
    int z=0;
    int Nindex=0;

    Step_N = new double [n+k+2][2];

    for (j=1; j<=k; j++)
    {
        for (i=1; i<=(n+k+1-j); i++)
{  
    if (j == 1)
        Step_N[i][2] = first(i, tindex);
    else
        Step_N[i][2] = formula(i, j, Step_N, tindex);
}

for (z=0; z <= (n+k+1); z++)
    Step_N[z][1] = Step_N[z][2];

for (Nindex=0; Nindex<=(n+1); Nindex++)
    N_Mat[index][Nindex] = Step_N[Nindex][1];

delete [] Step_N;

// void Base_spline::Set_xi (int i, double Xi)  
// {
//    flag = 0;
//    if (Xi >= knot[i-1])
//        knot[i] = Xi;
//    else
//        knot[i] = knot[i-1];
//}  
//
// double Base_spline::Get_xi (int i) const  
// {
//    return(knot[i]);
//}  
//
// Bspline definitions  
//
//general constructor - uses maximum values for inputs
B spline::B spline () : Base_spline()
{  
    //initializes dynamic arrays for use.
    Poly_Points = new double [Max_N + 2][Num_Coordinates];
}

//constructor with specified bspline inputs
B spline::B spline (int init_n, int init_k, int init_num_points) : Base_spline (init_n, init_k, init_num_points)
{  
    //initializes dynamic arrays for use.
    Poly_Points = new double [init_n + 2][Num_Coordinates];
}
// Bspline::Bspline (int init_n, int init_k, int init_num_points, double *init_knot) : Base_spline (init_n, init_k, init_num_points)
{
    int temp1=0;

    //initializes dynamic arrays for use.
    Poly_Points = new double [init_n + 2][Num_Coordinates];

    for (temp1=1; temp1<=(n+k+1); temp1++)
        knot[temp1] = *(init_knot + temp1 - 1);
}

//destructor returns dynamically allocated array memory
Bspline::~Bspline()
{
    delete [] Poly_Points;
}

// double Bspline::ReturnData(int Coord, int index)
{
    double Ret=0;
    int z=0;

    if (flag == 0)
        CalcN_Mat();

    for (z=1; z<= (n+l); z++)
        Ret = Ret + N_Mat[index][z] * Poly_Points[z][(Coord-1)];

    return (Ret);
}

// int Bspline::getn()
{
    return(n);
}

// int Bspline::getk()
{
    return(k);
}

// int Bspline::getnumpoints()
{
    return(num_points);
}
// void Bspline::Set_Poly (int i, double *M)
{  
    int z=0;
    flag = 0;
    for (z=0; z < Num_Coordinates; z++)
        Poly_Points[i][z] = *(M + z);
}

// double Bspline::Get_Poly (int i, int Coord) const
{  
    return(Poly_Points[i][Coord-1]);
}

// OnlyGetN_spline definitions

void OnlyGetN_spline::CalcNnow()
{  
    CalcN_Mat();
}

double OnlyGetN_spline::GetNvalue(int nindex, int point)
{  
    return(N_Mat[point][nindex]);
}

// OnlyGetN_spline::OnlyGetN_spline() : Base_spline()
{  
}

// OnlyGetN_spline::OnlyGetN_spline(int init_n, int init_k, int init_num_points) : Base_spline (init_n, init_k, init_num_points)
{  
}

// OnlyGetN_spline::OnlyGetN_spline(int init_n, int init_k, int init_num_points, double *init_knot) : Base_spline (init_n, init_k, init_num_points)
{  
    int temp1=0;
}
for (temp1=1; temp1<=(n+k+1); temp1++)
    knot[temp1] = *(init_knot + temp1 - 1);

    CalcNnow();
}

//
OnlyGetN_spline::OnlyGetN_spline()
{
}

//
//Airfoil definitions
//

Airfoil::Airfoil()
{
    topspline = new Bspline();
    botspline = new Bspline();
    Z = 0;
}

//
Airfoil::Airfoil(int init_n, int init_k, int init_num_points, double *init_knot, double z_coord, CString location, BOOL method)
{
    double (*topdata)[2];
    double (*botdata)[2];
    int top=0, bot=0;

    ProvideCount(location, &top, &bot);

    topdata = new double [top][2];
    botdata = new double [bot][2];

    GetData(location, topdata, botdata, top, bot);

    topspline = new Bspline(init_n, init_k, init_num_points, init_knot);
    botspline = new Bspline(init_n, init_k, init_num_points, init_knot);

    if (method == TRUE)
    {
        //performs calculations by using matrix inversion method
        Curvefit((&(topdata[0][0])), (&(botdata[0][0])), init_n, init_k, init_knot, top, bot, 1);

        if (top != bot)
            Curvefit((&(topdata[0][0])), (&(botdata[0][0])), init_n, init_k, init_knot, top, bot, 2);
    }

    Z = z_coord;

    page A -19
delete [] topdata;
delete [] botdata;
}

// Airfoil::Airfoil(int init_n, int init_k, int init_num_points, double *init_knot, double z_coord, double *polytop, double *polybot)
{
    topspline = new Bspline(init_n, init_k, init_num_points, init_knot);
    botspline = new Bspline(init_n, init_k, init_num_points, init_knot);
    Z = 0;

    int temp1=0;
    for (temp1=1; temp1<=(topspline->getnO + 1); temp1++)
        topspline->Set_Poly(temp1, (polytop + 2 * (temp1 -1)));
    for (temp1=1; temp1<=(botspline->getnO + 1); temp1++)
        botspline->Set_Poly(temp1, (polybot + 2 * (temp1 -1)));
}

// Airfoil::Airfoil(int init_n, int init_k, int init_num_points, double *init_knot, double z_coord, CString location, double (*init_P), BOOL method)
{
    double (*topdata)[2];
    double (*botdata)[2];
    int top=0, bot=0;

    ProvideCount(location, &top, &bot);

    todata = new double [top][2];
    botdata = new double [bot][2];

    GetData(location, todata, botdata, top, bot);

    topspline = new Bspline(init_n, init_k, init_num_points, init_knot);
    botspline = new Bspline(init_n, init_k, init_num_points, init_knot);

    if (method == TRUE)
    {
        //performs calculations by using matrix inversion method
        MixedCurvefit((&(topdata[0][0])), (&(botdata[0][0])), init_n, init_k, init_knot, top, bot, init_P, 1);

        if (top != bot)
            MixedCurvefit((&(topdata[0][0])), (&(botdata[0][0])), init_n, init_k, init_knot, top, bot, init_P, 2);
    }

    Z = z_coord;
delete [] topdata;
delete [] botdata;
}

//
Airfoil::~Airfoil()
{
    delete topspline;
    delete botspline;
}

//
void Airfoil::SetName(CString name)
{
    Name = name;
}

//
CString Airfoil::GetName() const
{
    return (Name);
}

//
void Airfoil::SetPolygon(double *Values, int index, int toporbottom)
{
    switch(toporbottom)
    {
    case 1:
        topspline->Set_Poly(index, Values);
        break;
    case 2:
        botspline->Set_Poly(index, Values);
        break;
    default:
        break;
    }
}

//
double Airfoil::GetPolygon(int Coord, int index, int toporbottom)
{
    double ret=0;
    switch(toporbottom)
    {
    case 1:
        ret=(topspline->Get_Poly(index, Coord));
        break;
    case 2:
        ret=(botspline->Get_Poly(index, Coord));
        break;
    default:
        break;
    }
break;

default:
    break;
}

return(ret);
}

//
void Airfoil::WriteOutData(CString location)
{
    ofstream outfile(location);
    int temp1=0;

    for(temp1=(topspline->getnumpoints()); temp1>=1; temp1--)
    {
        outfile<<((topspline->ReturnData(1, temp1)));
        outfile<<"\n";
        outfile<<((topspline->ReturnData(2, temp1)));
        outfile<<"\n";
    }

    for(temp1=1; temp1<=(botspline->getnumpoints()); temp1++)
    {
        outfile<<((botspline->ReturnData(1, temp1)));
        outfile<<"\n";
        outfile<<((botspline->ReturnData(2, temp1)));
        outfile<<"\n";
    }

    outfile.close();
}

//
double Airfoil::GetDataPoint(int Coord, int index, int toporbottom)
{
    double ret=0;

    switch(toporbottom)
    {
    case 1:
        ret = topspline->ReturnData(Coord, index);
        break;
    case 2:
        ret = botspline->ReturnData(Coord, index);
        break;
    default:
        break;
    }

    return (ret);
void Airfoil::Curvefit(double *data_top, double *data_bot, int init_n, int init_k, double *init_knot, int toppoints, int botpoints, int top_or_bottom)
{
    OnlyGetN_spline (*TempForN);
    double (*Ncurvefit);
    double (*curvefitdata);

    int temp1=0, temp2=0;

    switch (top_or_bottom)
    {
        case 1:
            TempForN = new OnlyGetN_spline(init_n, init_k, toppoints, init_knot);
            Ncurvefit = new double [toppoints * (init_n+1)];
            curvefitdata = new double [(init_n+1) * toppoints];

            for (temp1=1; temp1<=toppoints; temp1++)
            {
                for (temp2=1; temp2<=(init_n+1); temp2++)
                    Ncurvefit[(temp1-1) * (init_n +1) + (temp2 - 1)] =
                    (TempForN->GetNvalue(temp2, temp1));
            }
            Calculation (Ncurvefit, curvefitdata, toppoints, (init_n +1));
            SetPolyPoints (curvefitdata, data_top, toppoints, (init_n+1), 1);

            if (toppoints == botpoints)
                SetPolyPoints (curvefitdata, data_bot, botpoints, (init_n+1), 2);

            break;
        case 2:
            TempForN = new OnlyGetN_spline(init_n, init_k, botpoints, init_knot);
            Ncurvefit = new double [(init_n+1) * botpoints];
            curvefitdata = new double [(init_n+1) * botpoints];

            for (temp1=1; temp1<=botpoints; temp1++)
            {
                for (temp2=1; temp2<=(init_n+1); temp2++)
                    Ncurvefit[(temp1-1) * (init_n +1) + (temp2 - 1)] =
                    (TempForN->GetNvalue(temp2, temp1));
            }
            Calculation (Ncurvefit, curvefitdata, botpoints, (init_n +1));
            SetPolyPoints (curvefitdata, data_bot, botpoints, (init_n+1), 2);

            break;
        default:
            break;
    }
}
delete TempForN;
delete Ncurvefit;
delete curvefitdata;

//
void Airfoil::Calculation(double *Ncurvefit, double *curvefitdata, int numofpoints, int numofpolygons)
{
    double (*Ntrans);
    double (*curvestep1);
    double (*curvestep2);

    curvestep1 = new double [numofpolygons * numofpolygons]; // n x n matrix
    curvestep2 = new double [numofpolygons * numofpolygons]; // n x n matrix
    Ntrans = new double [numofpolygons * numofpoints]; // n x p matrix

    transpose(Ncurvefit, Ntrans, numofpoints, numofpolygons);
    matmult(Ntrans, Ncurvefit, curvestep1, numofpolygons, numofpoints, numofpolygons);
    InvertMatrix(curvestep1, curvestep2, numofpolygons);
    matmult(curvestep2, Ntrans, curvefitdata, numofpolygons, numofpolygons, numofpolygons);
    delete curvestep1;
    delete curvestep2;
    delete Ntrans;
}

//
void Airfoil::MixedCurvefit(double *data_top, double *data_bot, int init_n, int init_k, double *init_knot, int toppoints, int botpoints, double *init_P, int top_or_bottom)
{
    OnlyGetN_spline (*TempForN);
    double (*Ncurvefit);
    double (*curvefitdata);
    double (*calctopdata);
    double (*calcbotdata);

    int temp1=0, temp2=0;

    calctopdata = new double [ttoppoints * 2];
    calcbotdata = new double [botpoints * 2];

    for (temp1=0; temp1<(ttoppoints * 2); temp1++)
        *(calctopdata+temp1) = *(data_top + temp1);
for (temp1=0; temp1<(botpoints * 2); temp1++)
    *(calcbotdata+temp1) = *(data_bot + temp1);

switch (top_or_bottom)
{
    case 1:
        TempForN = new OnlyGetN_spline(init_n, init_k, toppoints, init_knot);
        Ncurvefit = new double [(init_n-l) * toppoints];
        curvefitdata = new double [(init_n-l) * toppoints];

        for (temp1=1; temp1<=toppoints; temp1++)
        {
            for (temp2=1; temp2<=(init_n+1); temp2++)
            {
                if (temp2 == 1)
                {
                    *(calctopdata + 2 * (temp1-1)) = *(data_top + 2 * (temp1-1)) - (*(init_P)) * (TempForN->GetNvalue(temp2, temp1));
                    *(calctopdata + 2 * (temp1-1) + 1) = *(data_top + 2 * (temp1-1) + 1) - (*(init_P + 1)) * (TempForN->GetNvalue(temp2, temp1));
                }

                if (temp2 == (init_n+1))
                {
                    *(calctopdata + 2 * (temp1-1)) = *(data_top + 2 * (temp1-1)) - (*(init_P + 2)) * (TempForN->GetNvalue(temp2, temp1));
                    *(calctopdata + 2 * (temp1-1) + 1) = *(data_top + 2 * (temp1-1) + 1) - (*(init_P + 3)) * (TempForN->GetNvalue(temp2, temp1));
                }

                if ((temp2 != 1) && (temp2 != (init_n+1)) && (temp2 != (init_n+1)))
                    Ncurvefit[(temp1 - 1) * (init_n - 1) + (temp2 - 2)] = (TempForN->GetNvalue(temp2, temp1));
            }
        }

        Calculation (Ncurvefit, curvefitdata, toppoints, (init_n - 1));
        SetMixedPolyPoints (init_P, curvefitdata, calctopdata, toppoints, (init_n-1), 1);

        if (toppoints == botpoints)
        {
            for (temp1=1; temp1<=botpoints; temp1++)
            {
                *(calcbotdata + 2 * (temp1-1)) = *(data_bot + 2 * (temp1-1)) - (*(init_P + 4)) * (TempForN->GetNvalue(1, temp1));
                *(calcbotdata + 2 * (temp1-1) + 1) = *(data_bot + 2 * (temp1-1) + 1) - (*(init_P + 5)) * (TempForN->GetNvalue(1, temp1));
                *(calcbotdata + 2 * (temp1-1)) = *(data_bot + 2 * (temp1-1)) - (*(init_P + 6)) * (TempForN->GetNvalue((init_n+1), temp1));

            }
        }
SetMixedPolyPoints (init_P, curvefitdata, calcbotdata, botpoints, (init_n-1), 2);
void Airfoil::SetPolyPoints(double *curvefitdata, double *data, int numofpoints, int numofpolygons, int top_or_bottom)
{
    double (*output)[2];
    int temp1=0;
    output = new double [numofpolygons][2];
    matmult(curvefitdata, data, (&(output[0][0])), numofpolygons, numofpoints, numofpoints, 2);
    switch (top_or_bottom)
    {
        case 1:
        for (temp1=1; temp1<=numofpolygons; temp1++)
            topspline->Set_Poly(temp1, (&(output[temp1-1][0])));
            break;
        case 2:
        for (temp1=1; temp1<=numofpolygons; temp1++)
            botspline->Set_Poly(temp1, (&(output[temp1-1][0])));
            break;
        default:
        break;
    }
    delete [] output;
}

void Airfoil::SetMixedPolyPoints(double *init_P, double *curvefitdata, double *data, int numofpoints, int numofpolygons, int top_or_bottom)
{
    double (*output)[2];
    int temp1=0;
    output = new double [numofpolygons][2];
    matmult(curvefitdata, data, (&(output[0][0])), numofpolygons, numofpoints, numofpoints, 2);
    switch (top_or_bottom)
    {
        case 1:
            topspline->Set_Poly(1, init_P);
            topspline->Set_Poly(numofpolygons+2, (init_P+2));
            for (temp1=1; temp1<=numofpolygons; temp1++)
            

```c

topspline->Set_Poly(temp1+1, (&(output[temp1 - 1][0])));
break;
case 2:
botspline->Set_Poly(1, (init_P+4));
botspline->Set_Poly(numofpolygons+2, (init_P+6));
for (temp1=1; temp1<=numofpolygons; temp1++)
    botspline->Set_Poly(temp1+1, (&(output[temp1 - 1][0])));
break;
default:
    break;
}
delete [] output;

Resource.h file:

//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Spline.rc

#define IDM_FOUR 100
#define IDM_ICON1 101
#define IDM_EXIT 110
#define IDD_TERMS 200
#define IDD_TITLE 201
#define IDM_ABOUT 300
#define IDC_EDIT0 1000
#define IDC_EDIT1 1001
#define IDC_EDIT2 1002
#define IDC_EDIT3 1003
#define IDC_EDIT4 1004
#define IDC_EDIT5 1005
#define IDC_EDIT6 1006
#define IDC_EDIT7 1007
#define IDC_EDIT8 1008
#define IDC_EDIT9 1009
#define IDC_EDIT10 1010
#define IDC_EDIT11 1011
#define IDC_EDIT12 1012
#define IDC_EDIT13 1013
#define IDC_EDIT14 1014
#define IDC_EDIT15 1015
#define IDC_EDIT16 1016
#define IDC_EDIT17 1017
#define IDC_EDIT18 1018
#define IDC_EDIT19 1019
#define IDC_EDIT20 1020
#define IDC_EDIT21 1021
```
// Next default values for new objects

#if defined APSTUDIO_INVOKED
#if defined APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 110
#define _APS_NEXT_COMMAND_VALUE 40015
#define _APS_NEXT_CONTROL_VALUE 1020
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
#endif
Spline.rc file:

//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
 strstr一日一八=印刷八会
// Generated from the TEXTINCLUDE 2 resource.
//
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS
#include "afxres.h"

// English (U.S.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32
#endif

AIRFOILMENU MENU DISCARDABLE
BEGIN
 POPUP "File"
 BEGIN
 MENUITEM "New...", IDM_NEW
 MENUITEM "Plot Scale...", IDM_SCALE
 MENUITEM "Export Data...", IDM_EXPORT
 MENUITEM "Export Vertices...", IDM_VERTEX
 MENUITEM "Exit", IDM_EXIT
 END
 POPUP "Data"
 BEGIN
 MENUITEM "Knot...", IDM_KNOT
 MENUITEM "Top Polygon...", IDM_POLYGON1
 MENUITEM "Bottom Polygon...", IDM_POLYGON2
 END
 POPUP "Calculations"

BEGIN
MENUITEM "Current Airfoil Information...", IDM_CURINFO
MENUITEM "Set Aerodynamic Parameters...", IDM_AERODYN
MENUITEM "Call X-FOIL", IDM_XFOIL
MENUITEM "Retrieve Last XFOIL Call...", IDM_RETRIEVE
END
POPUP "Help"
BEGIN
MENUITEM "About...", IDM_ABOUT
END
END

/////]

// Dialog

ABOUTBOX DIALOG DISCARDABLE 14, 22, 200, 77
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
CAPTION "About"
FONT 8, "MS Sans Serif"
BEGIN
CTEXT "Airfoil Spline Program",-1,28,5,144,8
CTEXT "RIT - Master's Thesis",-1,28,15,144,8
CTEXT "By Matthew MacLean",-1,28,26,144,8
CTEXT "version 3.0",201,59,37,83,8
DEFPUSHBUTTON "OK",IDOK,84,55,32,WS_GROUP
END

CONSTANTS DIALOG DISCARDABLE 0, 0, 187, 109
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Global Constants"
FONT 8, "MS Sans Serif"
BEGIN
LTEXT "Polygon Points (n):",IDC_STATIC,18,14,75,8
LTEXT "Order (k):",IDC_STATIC,46,34,29,8
LTEXT "Data Points: ",IDC_STATIC,38,54,55,8
EDITTEXT IDC_EDIT1,104,14,40,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT2,104,34,40,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT3,104,54,40,14,ES_AUTOHSCROLL
DEFPUSHBUTTON "OK",IDOK,23,88,50,14
PUSHBUTTON "Cancel",IDCANCEL,103,88,50,14
END

NEWAIRFOIL DIALOG DISCARDABLE 0, 0, 187, 98
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "New Airfoil"
FONT 8, "MS Sans Serif"
BEGIN
DEFPUSHBUTTON "OK",IDOK,26,77,50,14
END
PUSHBUTTON "Cancel", IDCANCEL, 101, 77, 50, 14
CONTROL "Import from File", IDC_RADIO1, "Button",
    BS_AUTORADIOBUTTON | WS_GROUP, 55, 15, 62, 10
CONTROL "Input Polygon Points", IDC_RADIO2, "Button",
    BS_AUTORADIOBUTTON, 55, 33, 77, 10
CONTROL "Specify End Points", IDC_RADIO3, "Button",
    BS_AUTORADIOBUTTON, 55, 51, 76, 10
END

POLYGON DIALOG DISCARDABLE 0, 0, 249, 163
STYLE DS_MODALFRAME | WS_POPUP | WS.Caption | WS_SYSMENU
CAPTION "Polygon Points"
FONT 8, "MS Sans Serif"
BEGIN
LTEXT "x", IDC_STATIC, 46, 5, 8, 8
LTEXT "y", IDC_STATIC, 94, 5, 8, 8
LTEXT "x", IDC_STATIC, 164, 5, 8, 8
LTEXT "y", IDC_STATIC, 212, 5, 8, 8
EDITTTEXT IDC_EDIT1, 30, 17, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT2, 75, 17, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT5, 30, 36, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT6, 75, 36, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT10, 30, 55, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT8, 75, 55, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT9, 30, 74, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT11, 75, 74, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT16, 30, 93, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT14, 75, 93, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT15, 30, 112, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT17, 75, 112, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT13, 145, 18, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT18, 190, 18, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT3, 145, 37, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT4, 190, 37, 40, 14, ES_AUTOHSCROLL
LTEXT "#1", IDC_STATIC, 16, 19, 9, 8
LTEXT "#2", IDC_STATIC, 16, 38, 9, 8
LTEXT "#3", IDC_STATIC, 16, 57, 9, 8
LTEXT "#4", IDC_STATIC, 16, 76, 9, 8
LTEXT "#5", IDC_STATIC, 16, 95, 9, 8
LTEXT "#6", IDC_STATIC, 16, 114, 9, 8
LTEXT "#7", IDC_STATIC, 125, 20, 9, 8
LTEXT "#8", IDC_STATIC, 125, 39, 9, 8
DEFPUSHBUTTON "OK", IDC_OK, 54, 142, 50, 14
PUSHBUTTON "Cancel", IDCANCEL, 155, 142, 50, 14
EDITTTEXT IDC_EDIT7, 145, 56, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT12, 190, 56, 40, 14, ES_AUTOHSCROLL
LTEXT "#9", IDC_STATIC, 125, 58, 9, 8
EDITTTEXT IDC_EDIT19, 145, 75, 40, 14, ES_AUTOHSCROLL
EDITTTEXT IDC_EDIT20, 190, 75, 40, 14, ES_AUTOHSCROLL
LTEXT "#10", IDC_STATIC, 125, 77, 18, 8
EDITTTEXT IDC_EDIT21, 145, 94, 40, 14, ES_AUTOHSCROLL
END

KNOT DIALOG DISCARDABLE  0, 0, 288, 108
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Knot Vector"
FONT 8, "MS Sans Serif"
BEGIN
EDITTEXT IDC_EDIT0,9,14,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT1,44,14,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT2,79,14,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT3,114,14,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT4,149,14,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT5,184,14,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT6,219,14,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT7,254,14,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT8,9,37,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT9,44,37,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT10,79,37,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT11,114,37,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT12,149,37,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT13,184,37,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT14,219,37,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT15,254,37,27,14,ES_AUTOHSCROLL
DEFPUSHBUTTON "OK",IDOK,78,87,50,14
PUSHBUTTON "Cancel",IDCANCEL,160,87,50,14
EDITTEXT IDC_EDIT16,9,60,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT17,44,60,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT18,79,60,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT19,114,60,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT20,149,60,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT21,184,60,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT22,219,60,27,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT23,254,60,27,14,ES_AUTOHSCROLL
END

SCALEBOX DIALOG DISCARDABLE  0, 0, 187, 64
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Plotting Scale"
FONT 8, "MS Sans Serif"
BEGIN
LTEXT "Y/X scale factor:",IDC_STATIC,33,17,50,8
EDITTEXT IDC_EDIT1,101,15,40,14,ES_AUTOHSCROLL
DEFPUSHBUTTON "OK",IDOK,21,43,50,14
PUSHBUTTON "Cancel",IDCANCEL,104,43,50,14
END
AEROPARAMETER DIALOG DISCARDBABLE 0, 0, 266, 149
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Aerodynamic Parameters"
FONT 8, "MS Sans Serif"
BEGIN

DEFPUSHBUTTON "OK",IDOK,51,128,50,14
PUSHBUTTON    "Cancel",(IDCANCEL,142,128,50,14
LTEXT         "Angle of Attack:" ,IDC_STATIC,25,20,52,8
LTEXT         "Reynold's Number:" ,IDC_STATIC,25,58,60,8
LTEXT         "Mach Number:" ,IDC_STATIC,25,99,48,8
EDITTEXT     IDC_EDIT1,121,18,95,14,ES_AUTOHSCROLL
EDITTEXT     IDC_EDIT2,121,56,95,14,ES_AUTOHSCROLL
EDITTEXT     IDC_EDIT3,121,97,95,14,ES_AUTOHSCROLL

ENDPT DIALOG DISCARDBABLE 0, 0, 241, 207
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Specified Polygon Points"
FONT 8, "MS Sans Serif"
BEGIN

DEFPUSHBUTTON "OK",IDOK,41,186,50,14
PUSHBUTTON    "Cancel",(IDCANCEL,149,186,50,14
GROUPBOX     "Top Spline",IDC_STATIC,36,15,169,74
GROUPBOX     "Bottom Spline",IDC_STATIC,36,92,169,74
EDITTEXT     IDC_EDIT1,101,32,40,14,ES_CENTER | ES_AUTOHSCROLL
EDITTEXT     IDC_EDIT2,155,32,40,14,ES_CENTER | ES_AUTOHSCROLL
EDITTEXT     IDC_EDIT3,101,60,40,14,ES_CENTER | ES_AUTOHSCROLL
EDITTEXT     IDC_EDIT4,155,60,40,14,ES_CENTER | ES_AUTOHSCROLL
EDITTEXT     IDC_EDIT5,101,112,40,14,ES_CENTER | ES_AUTOHSCROLL
EDITTEXT     IDC_EDIT6,155,112,40,14,ES_CENTER | ES_AUTOHSCROLL
EDITTEXT     IDC_EDIT7,101,141,40,14,ES_CENTER | ES_AUTOHSCROLL
EDITTEXT     IDC_EDIT8,155,141,40,14,ES_CENTER | ES_AUTOHSCROLL
LTEXT         "First Point:" ,IDC_STATIC,41,34,34,8
LTEXT         "Last Point:" ,IDC_STATIC,41,63,34,8
LTEXT         "First Point:" ,IDC_STATIC,46,118,34,8
LTEXT         "Last Point:" ,IDC_STATIC,46,147,34,8

END

#define APSTUDIO_INVOKED

//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDBABLE
BEGIN
"resource.h"0"
END

2 TEXTINCLUDE DISCARDBABLE
BEGIN
"#define APSTUDIO_HIDDEN_SYMBOLS\n" "#include "windows.h"\n" "#undef APSTUDIO_HIDDEN_SYMBOLS\n" "#include "afxres.h"\n"

END

3 TEXTINCLUDE DISCARDABLE
BEGIN
 "\n"
 "0"
END

#endif // APSTUDIO_INVOKED


// Icon

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_ICON1 ICON DISCARDABLE "icon1.ico"


// DESIGNINFO


#define APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
 "CONSTANTS", DIALOG
 BEGIN
   LEFTMARGIN, 7
   RIGHTMARGIN, 180
   TOPMARGIN, 7
   BOTTOMMARGIN, 102
 END

 "NEWAIRFOIL", DIALOG
 BEGIN
   LEFTMARGIN, 7
   RIGHTMARGIN, 180
   TOPMARGIN, 7
   BOTTOMMARGIN, 91
 END

 "POLYGON", DIALOG
 BEGIN
   LEFTMARGIN, 7
WinDlg.h file:

#include <string.h>
#include <iomanip.h>
#include <afxwin.h>
#include <afxcmn.h>
#include <afxdlgs.h>
#include <math.h>
#include "resource.h"
#include "constant.h"
#include "matrix.h"
#include "spline.h"

#ifndef _WinDlg_H
#define _WinDlg_H

extern CString mytitle;
extern int radio;
extern int method;
extern int m_cXClient;
extern int m_cyClient;
extern int globalflag;
extern double scalefactor;

extern CString globalimport;
extern CString globalexport;
extern CString globalCurDir;

extern Airfoil *currentairfoil;
extern double globalknot[Max_N + Max_K + 1];
extern int globalN;
extern int globalK;
extern int globalDataP;

extern double alpha;
extern double reynolds;
extern double machnum;

extern float cl;

#endif  // not APSTUDIO_INVOKED
extern float cdrg;
extern float cm;

//XY Scale Dialog Box
class CScaleDataDialog : public CDialog
{
public:
    CScaleDataDialog(CWnd* pParentWnd=NULL) : CDialog("Scalebox",pParentWnd)
    {
    }

    virtual void DoDataExchange(CDataExchange* pDX);
};

//Constants Dialog Box
class CConstDataDialog : public CDialog
{
public:
    CConstDataDialog(CWnd* pParentWnd=NULL) : CDialog("Constants",pParentWnd)
    {
    }

    virtual void DoDataExchange(CDataExchange* pDX);
};

//Knot Vector Box
class CKnotDataDialog : public CDialog
{
public:
    CKnotDataDialog(CWnd* pParentWnd=NULL) : CDialog("Knot",pParentWnd)
    {
    }

    virtual void DoDataExchange(CDataExchange* pDX);
};

//New Airfoil Dialog Box
class CNewDataDialog : public CDialog
{
public:
    CNewDataDialog(CWnd* pParentWnd=NULL) : CDialog("NewAirfoil",pParentWnd)
    {
    }

    virtual void DoDataExchange(CDataExchange* pDX);
};

//specified polygons points dialog box
class CEndPtDialog : public CDialog
{
public:
    CEndPtDialog(CWnd* pParentWnd=NULL);
    virtual void DoDataExchange(CDataExchange* pDX);
    virtual double *ReturnPolyPt();

private:
    double PolyPt[4][2];
};

//Top B spline Polygon Box
class CTopSplineDataDialog : public CDialog
{
    public:
        CTopSplineDataDialog(CWnd* pParentWnd=NULL) : CDialog("Polygon", pParentWnd) {
        
        virtual void DoDataExchange(CDataExchange* pDX);
    
};

//Bottom B spline Polygon Box
class CBotSplineDataDialog : public CDialog
{
    public:
        CBotSplineDataDialog(CWnd* pParentWnd=NULL) : CDialog("Polygon", pParentWnd) {
        
        virtual void DoDataExchange(CDataExchange* pDX);
    
};

// B spline Polygon Box for a new airfoil
class CNewSplineDataDialog : public CDialog
{
    public:
        CNewSplineDataDialog(int N, CWnd* pParentWnd=NULL) : CDialog("Polygon", pParentWnd) {
            Values = new double [N][2];
            int i = 0;
            for (i = 0; i < N; i++)
            {
                Values[i][0] = 0.0;
                Values[i][1] = 0.0;
            }
        
        ~CNewSplineDataDialog()
        {
            delete [] Values;
        }
virtual double *ReturnAddress();
virtual void DoDataExchange(CDataExchange* pDX);

private:
    double (*Values)[2];
};

//Aerodynamic Parameters Dialog Box
class CAeroDataDialog : public CDialog
{
public:
    CAeroDataDialog(CWnd* pParentWnd=NULL) : CDialog("AeroParameter", pParentWnd)
    {
    }

    virtual void DoDataExchange(CDataExchange* pDX);
};
#endif //_WinDlg_H

WinDlg.cpp file:

#include "WinDlg.h"


#include "WinDlg.h"

// defines the various dialog box members in support
// of the WinFunc.cpp file

void CScaleDataDialog::DoDataExchange(CDataExchange* pDX)
{
    CDIalog::DoDataExchange(pDX);

    DDX_Text(pDX, IDC_EDIT1, scaleFactor);
}

void CNewDataDialog::DoDataExchange(CDataExchange* pDX)
// CConstDataDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_EDIT1, globalN);
    DDX_Text(pDX, IDC_EDIT2, globalK);
    DDX_Text(pDX, IDC_EDIT3, globalDataP);
    DDX_Radio(pDX, IDC_RADIO1, method);
}

// CEndPtDialog::CEndPtDialog(CWnd* pParentWnd) : CDialog("Endpt", pParentWnd)
{ 
    PolyPt[0][0] = 0;
    PolyPt[0][1] = 0;
    PolyPt[1][0] = 1;
    PolyPt[1][1] = 0;
    PolyPt[2][0] = 0;
    PolyPt[2][1] = 0;
    PolyPt[3][0] = 1;
    PolyPt[3][1] = 0;
}

// CEndPtDialog::DoDataExchange(CDataExchange* pDX)
{
    CDlgDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_EDIT1, PolyPt[0][0]);
    DDX_Text(pDX, IDC_EDIT2, PolyPt[0][1]);
    DDX_Text(pDX, IDC_EDIT3, PolyPt[1][0]);
    DDX_Text(pDX, IDC_EDIT4, PolyPt[1][1]);
    DDX_Text(pDX, IDC_EDIT5, PolyPt[2][0]);
    DDX_Text(pDX, IDC_EDIT6, PolyPt[2][1]);
    DDX_Text(pDX, IDC_EDIT7, PolyPt[3][0]);
    DDX_Text(pDX, IDC_EDIT8, PolyPt[3][1]);
}

// double *CEndPtDialog::ReturnPolyPt()
{
    return (&PolyPt[0][0]);
}
double *CNewSplineDataDialog::ReturnAddress()
{
    return(&Values[0][0]);
}

void CNewSplineDataDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);

    DDX_Text(pDX, IDC_EDIT1, Values[0][0]);
    DDX_Text(pDX, IDC_EDIT2, Values[0][1]);
    DDX_Text(pDX, IDC_EDIT5, Values[1][0]);
    DDX_Text(pDX, IDC_EDIT6, Values[1][1]);

    if(globalN >= 2)
    {
        DDX_Text(pDX, IDC_EDIT10, Values[2][0]);
        DDX_Text(pDX, IDC_EDIT8, Values[2][1]);
    }

    if(globalN >= 3)
    {
        DDX_Text(pDX, IDC_EDIT9, Values[3][0]);
        DDX_Text(pDX, IDC_EDIT11, Values[3][1]);
    }

    if(globalN >= 4)
    {
        DDX_Text(pDX, IDC_EDIT16, Values[4][0]);
        DDX_Text(pDX, IDC_EDIT14, Values[4][1]);
    }

    if(globalN >= 5)
    {
        DDX_Text(pDX, IDC_EDIT15, Values[5][0]);
        DDX_Text(pDX, IDC_EDIT17, Values[5][1]);
    }

    if(globalN >= 6)
    {
        DDX_Text(pDX, IDC_EDIT13, Values[6][0]);
        DDX_Text(pDX, IDC_EDIT18, Values[6][1]);
    }

    if(globalN >= 7)
    {
        DDX_Text(pDX, IDC_EDIT3, Values[7][0]);
    }
DDX_Text(pDX, IDC_EDIT4, Values[7][1]);

if (globalN >= 8)
{
    DDX_Text(pDX, IDC_EDIT7, Values[8][0]);
    DDX_Text(pDX, IDC_EDIT12, Values[8][1]);
}

if (globalN >= 9)
{
    DDX_Text(pDX, IDC_EDIT19, Values[9][0]);
    DDX_Text(pDX, IDC_EDIT20, Values[9][1]);
}

if (globalN >= 10)
{
    DDX_Text(pDX, IDC_EDIT21, Values[10][0]);
    DDX_Text(pDX, IDC_EDIT22, Values[10][1]);
}

if (globalN >= 11)
{
    DDX_Text(pDX, IDC_EDIT23, Values[10][0]);
    DDX_Text(pDX, IDC_EDIT24, Values[10][1]);
}

void CTopSplineDataDialog::DoDataExchange(CDataExchange* pDX)
{
    CDlg::DoDataExchange(pDX);
    double tempvalue[2];

    if (globalflag == 1)
    {
        tempvalue[0] = currentairfoil->GetPolygon(1, 1, 1);
        tempvalue[1] = currentairfoil->GetPolygon(2, 1, 1);
        DDX_Text(pDX, IDC_EDIT1, tempvalue[0]);
        DDX_Text(pDX, IDC_EDIT2, tempvalue[1]);
        currentairfoil->SetPolygon(tempvalue, 1, 1);

        tempvalue[0] = currentairfoil->GetPolygon(1, 2, 1);
        tempvalue[1] = currentairfoil->GetPolygon(2, 2, 1);
        DDX_Text(pDX, IDC_EDIT5, tempvalue[0]);
        DDX_Text(pDX, IDC_EDIT6, tempvalue[1]);
        currentairfoil->SetPolygon(tempvalue, 2, 1);

        if (globalN >= 2)
        {
            tempvalue[0] = currentairfoil->GetPolygon(1, 3, 1);
tempvalue[1] = currentairfoil->GetPolygon(2, 3, 1);
DDX_Text(pDX, IDC_EDIT10, tempvalue[0]);
DDX_Text(pDX, IDC_EDIT8, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 3, 1);
}

if (globalN >= 3)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 4, 1);
tempvalue[1] = currentairfoil->GetPolygon(2, 4, 1);
    DDX_Text(pDX, IDC_EDIT9, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT11, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 4, 1);
}

if (globalN >= 4)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 5, 1);
tempvalue[1] = currentairfoil->GetPolygon(2, 5, 1);
    DDX_Text(pDX, IDC_EDIT16, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT14, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 5, 1);
}

if (globalN >= 5)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 6, 1);
tempvalue[1] = currentairfoil->GetPolygon(2, 6, 1);
    DDX_Text(pDX, IDC_EDIT15, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT17, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 6, 1);
}

if (globalN >= 6)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 7, 1);
tempvalue[1] = currentairfoil->GetPolygon(2, 7, 1);
    DDX_Text(pDX, IDC_EDIT13, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT18, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 7, 1);
}

if (globalN >= 7)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 8, 1);
tempvalue[1] = currentairfoil->GetPolygon(2, 8, 1);
    DDX_Text(pDX, IDC_EDIT3, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT4, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 8, 1);
}
if (globalN >= 8) {
    tempvalue[0] = currentairfoil->GetPolygon(1, 9, 1);
    tempvalue[1] = currentairfoil->GetPolygon(2, 9, 1);
    DDX_Text(pDX, IDC_EDIT7, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT12, tempvalue[1]);
    currentairfoil->SetPolygon(tempvalue, 9, 1);
}

if (globalN >= 9) {
    tempvalue[0] = currentairfoil->GetPolygon(1, 10, 1);
    tempvalue[1] = currentairfoil->GetPolygon(2, 10, 1);
    DDX_Text(pDX, IDC_EDIT19, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT20, tempvalue[1]);
    currentairfoil->SetPolygon(tempvalue, 10, 1);
}

if (globalN >= 10) {
    tempvalue[0] = currentairfoil->GetPolygon(1, 11, 1);
    tempvalue[1] = currentairfoil->GetPolygon(2, 11, 1);
    DDX_Text(pDX, IDC_EDIT21, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT22, tempvalue[1]);
    currentairfoil->SetPolygon(tempvalue, 11, 1);
}

if (globalN >= 11) {
    tempvalue[0] = currentairfoil->GetPolygon(1, 12, 1);
    tempvalue[1] = currentairfoil->GetPolygon(2, 12, 1);
    DDX_Text(pDX, IDC_EDIT23, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT24, tempvalue[1]);
    currentairfoil->SetPolygon(tempvalue, 12, 1);
}

//
void CBotSplineDataDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    if (globalflag==1) {
        double tempvalue[2];
        tempvalue[0] = currentairfoil->GetPolygon(1, 1, 2);
        tempvalue[1] = currentairfoil->GetPolygon(2, 1, 2);
        DDX_Text(pDX, IDC_EDIT1, tempvalue[0]);
        DDX_Text(pDX, IDC_EDIT2, tempvalue[1]);
        currentairfoil->SetPolygon(tempvalue, 1, 2);
    }
}
tempvalue[0] = currentairfoil->GetPolygon(1, 2, 2);
tempvalue[1] = currentairfoil->GetPolygon(2, 2, 2);
DDX_Text(pDX, IDC_EDIT5, tempvalue[0]);
DDX_Text(pDX, IDC_EDIT6, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 2, 2);

if (globalN >= 2)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 3, 2);
tempvalue[1] = currentairfoil->GetPolygon(2, 3, 2);
    DDX_Text(pDX, IDC_EDIT10, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT8, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 3, 2);
}

if (globalN >= 3)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 4, 2);
tempvalue[1] = currentairfoil->GetPolygon(2, 4, 2);
    DDX_Text(pDX, IDC_EDIT9, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT11, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 4, 2);
}

if (globalN >= 4)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 5, 2);
tempvalue[1] = currentairfoil->GetPolygon(2, 5, 2);
    DDX_Text(pDX, IDC_EDIT16, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT17, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 5, 2);
}

if (globalN >= 5)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 6, 2);
tempvalue[1] = currentairfoil->GetPolygon(2, 6, 2);
    DDX_Text(pDX, IDC_EDIT15, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT18, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 6, 2);
}

if (globalN >= 6)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 7, 2);
tempvalue[1] = currentairfoil->GetPolygon(2, 7, 2);
    DDX_Text(pDX, IDC_EDIT13, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT18, tempvalue[1]);
currentairfoil->SetPolygon(tempvalue, 7, 2);
}
if (globalN >= 7)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 8, 2);
    tempvalue[1] = currentairfoil->GetPolygon(2, 8, 2);
    DDX_Text(pDX, IDC_EDIT3, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT4, tempvalue[1]);
    currentairfoil->SetPolygon(tempvalue, 8, 2);
}

if (globalN >= 8)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 9, 2);
    tempvalue[1] = currentairfoil->GetPolygon(2, 9, 2);
    DDX_Text(pDX, IDC_EDIT7, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT12, tempvalue[1]);
    currentairfoil->SetPolygon(tempvalue, 9, 2);
}

if (globalN >= 9)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 10, 2);
    tempvalue[1] = currentairfoil->GetPolygon(2, 10, 2);
    DDX_Text(pDX, IDC_EDIT19, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT20, tempvalue[1]);
    currentairfoil->SetPolygon(tempvalue, 10, 2);
}

if (globalN >= 10)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 11, 2);
    tempvalue[1] = currentairfoil->GetPolygon(2, 11, 2);
    DDX_Text(pDX, IDC_EDIT21, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT22, tempvalue[1]);
    currentairfoil->SetPolygon(tempvalue, 11, 2);
}

if (globalN >= 11)
{
    tempvalue[0] = currentairfoil->GetPolygon(1, 12, 2);
    tempvalue[1] = currentairfoil->GetPolygon(2, 12, 2);
    DDX_Text(pDX, IDC_EDIT23, tempvalue[0]);
    DDX_Text(pDX, IDC_EDIT24, tempvalue[1]);
    currentairfoil->SetPolygon(tempvalue, 12, 2);
}

//
void CKnotDataDialog::DoDataExchange(CDataExchange* pDX)
{

CDialog::DoDataExchange(pDX);

DDX_Text(pDX, IDC_EDIT0, globalknot[0]);
DDX_Text(pDX, IDC_EDIT1, globalknot[1]);
DDX_Text(pDX, IDC_EDIT2, globalknot[2]);
DDX_Text(pDX, IDC_EDIT3, globalknot[3]);
DDX_Text(pDX, IDC_EDIT4, globalknot[4]);
DDX_Text(pDX, IDC_EDIT5, globalknot[5]);
DDX_Text(pDX, IDC_EDIT6, globalknot[6]);
DDX_Text(pDX, IDC_EDIT7, globalknot[7]);
DDX_Text(pDX, IDC_EDIT8, globalknot[8]);
DDX_Text(pDX, IDC_EDIT9, globalknot[9]);
DDX_Text(pDX, IDC_EDIT10, globalknot[10]);
DDX_Text(pDX, IDC_EDIT11, globalknot[11]);
DDX_Text(pDX, IDC_EDIT12, globalknot[12]);
DDX_Text(pDX, IDC_EDIT13, globalknot[13]);
DDX_Text(pDX, IDC_EDIT14, globalknot[14]);
DDX_Text(pDX, IDC_EDIT15, globalknot[15]);
DDX_Text(pDX, IDC_EDIT16, globalknot[16]);
DDX_Text(pDX, IDC_EDIT17, globalknot[17]);
DDX_Text(pDX, IDC_EDIT18, globalknot[18]);
DDX_Text(pDX, IDC_EDIT19, globalknot[19]);
DDX_Text(pDX, IDC_EDIT20, globalknot[20]);
DDX_Text(pDX, IDC_EDIT21, globalknot[21]);
DDX_Text(pDX, IDC_EDIT22, globalknot[22]);
DDX_Text(pDX, IDC_EDIT23, globalknot[23]);

//
void CAeroDataDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);

    DDX_Text(pDX, IDC_EDIT1, alpha);
    DDX_Text(pDX, IDC_EDIT2, reynolds);
    DDX_Text(pDX, IDC_EDIT3, machnum);
}

WinFunc.h file:

#include <string.h>
#include <iomanip.h>
#include <a&Awin.h>
#include <afxcmn.h>
#include <afxdlgs.h>
#include <math.h>
#include "resource.h"
#include "constant.h"
#include "matrix.h"
#include "spline.h"
#include "WinDlg.h"
#include "global.h"

////////////////////////////////////////////
//
// WinFunc.h
//
////////////////////////////////////////////

#ifndef _WinFunc_H
#define _WinFunc_H

//defines main window class

class CMainWnd : public CFrameWnd
{
public:
    CMainWnd();
    afx_msg void OnPaint();
    afx_msg void OnSize(UINT,int,int);
    afx_msg int OnCreate(LPCREATESTRUCT cs);
    afx_msg void OnAbout();
    afx_msg void OnExportData();
    afx_msg void OnVertexExport();
    afx_msg void OnNewData();
    afx_msg void OnTopPolyData();
    afx_msg void OnBotPolyData();
    afx_msg void OnKnotData();
    afx_msg void OnInfoData();
    afx_msg void OnRetrieve();
    afx_msg void OnAeroData();
    afx_msg void OnXfoilData();
    afx_msg void OnExit();
    afx_msg void OnScaleData();
    DECLARE_MESSAGE_MAP()
};

//
//defines main application class
//

class CTheApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
    ~CTheApp();
};
Winfunc.cpp file:

#include "WinFunc.h"

/*
****** ******
** Begin message map.  **
****** ******
*/

BEGIN_MESSAGE_MAP(CMainWnd, CFrameWnd)
    ON_WM_PAINT()
    ON_WM_SIZE()
    ON_WM_CREATE()
    ON_COMMAND(IDM_ABOUT, OnAbout)
    ON_COMMAND(IDM_EXPORT, OnExportData)
    ON_COMMAND(IDM_VERTEX, OnVertexExport)
    ON_COMMAND(IDM_NEW, OnNewData)
    ON_COMMAND(IDM_SCALE, OnScaleData)
    ON_COMMAND(IDC_POLYGON1, OnTopPolyData)
    ON_COMMAND(IDC_POLYGON2, OnBotPolyData)
    ON_COMMAND(IDC_KNOT, OnKnotData)
    ON_COMMAND(IDC_CURINFO, OnInfoData)
    ON_COMMAND(IDC_RETRIEVE, OnRetrieve)
    ON_COMMAND(IDC_AERODYN, OnAeroData)
    ON_COMMAND(IDC_XFOIL, OnXfoilData)
    ON_COMMAND(IDC_EXIT, OnExit)
END_MESSAGE_MAP()

/*
****** ******
** Begin Main.  **
*/
CTheApp theApp;

/*/ Begin Functions. */

********

// CMainWnd::CMainWnd()
{
    Create((AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW,LoadCursor(NULL,IDC_CROSS), (HBRUSH) (GetStockObject(WHITE_BRUSH)),NULL)), "Spline Airfoil Application", WS_OVERLAPPEDWINDOW|WS_THICKFRAME, rectDefault, NULL, "AirfoilMenu");
}

// void CMainWnd::OnSize(UINT, int x, int y)
{
    m_cxClient = x;
    m_cyClient = y;
}

// void CMainWnd::OnPaint()
{
    CPaintDC dc(this);

    int i, j;
    double y, x,
    CPen newpen;
    CPen airfoilpen;

    //create a custom drawing surface
    dc.SetMapMode(MM_ISOTROPIC);
    dc.SetWindowExt(500,500);
    dc.SetViewportExt(m_cxClient, m_cyClient);
    dc.SetViewportOrg(m_cxClient/10, m_cyClient/2);

    newpen.CreatePen(BS_SOLID,2,RGB(0,0,0));
    dc.SelectObject(&newpen);

    //draw x & y coordinate axes
    dc.MoveTo(0,240);
    dc.LineTo(0,-240);
    dc.MoveTo(0,0);
    dc.LineTo(500,0);
    dc.MoveTo(0,0);
// draws airfoil
if (globalflag != 0)
{
    airfoilpen.CreatePen(BS_SOLID, 3, RGB(255,0,0));
dc.SelectObject(&airfoilpen);

    for (i=1; i<=globalDataP; i++)
    {
        x = currentairfoil->GetDataPoint(1, i, 1);
y = currentairfoil->GetDataPoint(2, i, 1);
dc.LineTo(x*500,(int)(y*500*scalefactor));
    }
dc.MoveTo(0, 0);
    for (j=1; j<=globalDataP; j++)
    {
        x = currentairfoil->GetDataPoint(1, j, 2);
y = currentairfoil->GetDataPoint(2, j, 2);
dc.LineTo(x*500,(int)(y*500*scalefactor));
    }
}
dc.SelectStockObject(BLACK_PEN);

// int CMainWnd::OnCreate(LPCREATESTRUCT cs)
{
    CFrameWnd::OnCreate(cs);

    UpdateWindowO;
    return(0);
}

// void CMainWnd::OnAbout()
{
    CModalDialog about("AboutBox",this);
    about.DoModalO;
}

// void CMainWnd::OnScaleData()
{
    CScaleDataDialog dlgScaleData(this);
    if (dlgScaleData.DoModalO==IDOK)
    {
        InvalidateRect(NULL,TRUE);
        UpdateWindowO;
    }
}
//
void CMainWnd::OnVertexExport()
{
    int i=0;
    CString FileFilter = "Text Files|*.txt|Dat Files |*.dat|All Files (*.*)|*.*||";

    CFileDialog dlgImportVertex(FALSE, ".txt", "vertex.txt",
    OFN_OVERWRITEPROMPT|OFN_HIDEREADONLY,(LPCTSTR)FileFilter, this);

    dlgImportVertex.m_ofn.lpstrInitialDir = globalCurDir;
    if (IDOK == dlgImportVertex.DoModal())
    {
        ofstream outfile(dlgImportVertex.GetPathName());
        for (i=1; i<=(globalN+1); i++)
        { outfile << currentairfoil->GetPolygon(1, i, 1) << "\t" << currentairfoil->GetPolygon(2, i, 1) << endl;
            for (i=1; i<=(globalN+1); i++)
            { outfile << currentairfoil->GetPolygon(1, i, 2) << "\t" << currentairfoil->GetPolygon(2, i, 2) << endl;
        outfile.close();
    }
}
} //

// void CMainWnd::OnExportData()
{
    CString FileFilter = "Text Files|*.txt|Dat Files |*.dat|All Files (*.*)|*.*||";

    CFileDialog dlgImportData(FALSE, ".txt", "airfoil.txt",
    OFN_OVERWRITEPROMPT|OFN_HIDEREADONLY,(LPCTSTR)FileFilter, this);

    dlgImportData.m_ofn.lpstrInitialDir = globalCurDir;
    if (IDOK == dlgImportData.DoModal())
    {
        globalexport = dlgImportData.GetPathName();
        currentairfoil->WriteOutData(globalexport);
    }
}

// void CMainWnd::OnNewData()
{ 
    CString msg;

    CNewDataDialog dlgNewData(this);
    if (dlgNewData.DoModal()==IDOK)
{ }
InvalidateRect(NULL,TRUE);
UpdateWindow();

CConstDataDialog dlgConstantData(this);
if (dlgConstantData.DoModal()==IDOK)
{
    InvalidateRect(NULL,TRUE);
    UpdateWindow();

    if (globalN > Max_N)
        globalN = Max_N;

    if (globalK > Max_K)
        globalK = Max_K;

    if (globalDataP > Max_Num_Points)
        globalDataP = Max_Num_Points;

    CKnotDataDialog dlgKnotData(this);

    msg.Format("The knot vector must have %d terms.",globalN + globalK + 1);
    MessageBox(msg,"Knot Vector Requirements",MB_OK | MB_APPLMODAL | MB_ICONASTERISK);

    if (dlgKnotData.DoModal()==IDOK)
    {
        if (radio==0)
        {
            CString FileFilter = "Text Files*.txt|Dat Files*.dat|All Files(*.*)|*.*||";

            CFileDialog dlgImportData(TRUE,0,0,
            OFN_FILEMUSTEXIST,(LPCTSTR)FileFilter,this);

            dlgImportData.m_ofn.lpstrInitialDir = globalCurDir;

            if (IDOK == dlgImportData.DoModal())
            {
                globalimport = dlgImportData.GetPathName();

                if (globalflag==1)
                    delete currentairfoil;

                currentairfoil = new Airfoil(globalN, globalK,
                globalDataP, &(globalknot[0]), 0.00, globalimport);

                globalflag=1;
                globalaerorun=0;
                InvalidateRect(NULL,TRUE);
                UpdateWindow();
            }
        }
    }
}
if (radio==0)
{
    CNewSplineDataDialog dlgNewTopPoly(globalN+1, this);
    if (dlgNewTopPoly.DoModal() == IDOK)
    {
        CNewSplineDataDialog dlgNewBotPoly(globalN+1, this);
        if (dlgNewBotPoly.DoModal() == IDOK)
        {
            if (globalflag==1)
                delete currentairfoil;
            globalflag=1;
            globalaerorun=0;
            currentairfoil = new Airfoil(globalN, 
                globalK, globalDataP, &(globalknot[0]), 0.0, dlgNewTopPoly.ReturnAddress(),
                dlgNewBotPoly.ReturnAddress());

            InvalidateRect(NULL, TRUE);
            UpdateWindow();
        } //if (dlgNewBotPoly.DoModal()==IDOK)
    } //if (dlgNewTopPoly.DoModal()==IDOK)

    InvalidateRect(NULL, TRUE);
    UpdateWindow();
} //if (radio==1)

if (radio == 2)
{
    CEndPtDialog pnts(this);

    if (pnts.DoModal() == IDOK)
    {
        CString FileFilter = "Text Files (*.txt)|Dat Files (*.dat)|All Files (*.*)|*.*||";

        CFileDialog dlgImportData(TRUE, 0, 0,
            OFN_FILEMUSTEXIST,(LPCTSTR)FileFilter, this);

        dlgImportData.m_ofn.lpstrInitialDir = globalCurDir;

        if (IDOK == dlgImportData.DoModal())
        {
            globalimport =
            dlgImportData.GetPathName();

            if (globalflag==1)
                delete currentairfoil;

            } //if (IDOK == dlgImportData.DoModal())
    } //if (pnts.DoModal()==IDOK)
} //if (radio==2)
currentairfoil = new Airfoil(globalN,
globalK, globalDataP, &(globalknot[0]), 0.00, globalimport, pnts.ReturnPolyPt());

globalflag=1;
globalaerorun=0;
InvalidateRect(NULL,TRUE);
UpdateWindow();
} //if (IDOK == dlgImportData.DoModal())
} //if (pnts.DoModal() == IDOK)
} //if (radio == 2)
} //if (dlgKnotData.DoModal() == IDOK)
} //if (dlgConstantData.DoModal()==IDOK)
} //if (dlgNewData.DoModal()==IDOK)

void CMainWnd::OnTopPolyData()
{
    CTopSplineDataDialog dlgPolygonData(this);
    if (dlgPolygonData.DoModal()==IDOK)
    {
        InvalidateRect(NULL,TRUE);
        UpdateWindow();
    }
}

void CMainWnd::OnBotPolyData()
{
    CBotSplineDataDialog dlgPolygonData(this);
    if (dlgPolygonData.DoModal()==IDOK)
    {
        InvalidateRect(NULL,TRUE);
        UpdateWindow();
    }
}

void CMainWnd::OnKnotData()
{
    CKnotDataDialog dlgKnotData(this);
    if (dlgKnotData.DoModal()==IDOK)
    {
        InvalidateRect(NULL,TRUE);
        UpdateWindow();
    }
}

void CMainWnd::OnInfoData()
CString msg;

if (globalflag == 1)
{
    msg.Format("top surface data points: %d\nbottom surface data points: %d\norder(k): %d\npolygon points(n+1): %d", globalDataP, globalDataP, globalK, globalN+1);
}
else
    msg.Format("Provides information on the current airfoil.");

MessageBox(msg, "Current Airfoil Information", MB_OK | MB_APPLMODAL | MB_ICONASTERISK);

//
void CMainWnd::OnRetrieve()
{
    CString msg;

    if(globalaerorun == 1)
        msg.Format("Results:\n\ncl = %f\ncd = %f\ncm = %f", cl, cdrag, cm);
    else
        msg.Format("Displays the results of an XFOIL call.");

    MessageBox(msg, "Aerodynamic Performance", MB_OK | MB_APPLMODAL | MB_ICONASTERISK);
}

//
void CMainWnd::OnAeroData()
{
    CAeroDataDialog dlgAeroData(this);
    if (dlgAeroData.DoModal() == IDOK)
    {
        InvalidateRect(NULL, TRUE);
        UpdateWindow();
    }
}

//
void CMainWnd::OnXfoilData()
{
    if (globalflag != 0)
    {
        float *x_points;
        float *y_points;
        int temp1 = 0;
        int tempcount = 0;

        int npoint = 0;

float tempreyno = (float)reynolds;
float tempalpha = (float)alpha;
float tempmach = (float)machnum;

x_points = new float [Max_XFOIL_Points];
y_points = new float [Max_XFOIL_Points];

//write top data to array
for (temp1=globalDataP; temp1>0; temp1--)
{
    x_points[tempcount] = currentairfoil->GetDataPoint(1, temp1, 1);
    y_points[tempcount] = currentairfoil->GetDataPoint(2, temp1, 1);
    tempcount++;
}

//write bottom data to array
for (temp1=1; temp1<=globalDataP; temp1++)
{
    x_points[tempcount] = currentairfoil->GetDataPoint(1, temp1, 2);
    y_points[tempcount] = currentairfoil->GetDataPoint(2, temp1, 2);
    if ((tempcount == globalDataP) && (x_points[globalDataP - 1] == x_points[globalDataP]) && (y_points[globalDataP] == y_points[globalDataP - 1]))
        continue;
    else
        tempcount++;
}

npoint = tempcount;

XFOIL(&npoint, x_points, y_points, &tempalpha, &tempreyno, &tempmach, &cl, &cdrag, &cm, &error);

CString msg;
msg.Format("Results:
  cl = \%\%n  cd = \%\%n  cm = \%\%n", cl, cdrag, cm);
MessageBox(msg,"Aerodynamic Performance", MB_OK | MB_APPLMODAL | MB_ICONASTERISK);

globalaerorun = 1;
delete x_points;
delete y_points;
}

// void CMainWnd::OnExit()
{
delete currentairfoil;
DestroyWindow();

//
BOOL CTheApp::InitInstance()
{
    #ifdef _AFXDLL
        Enable3dControls();
    #else
        Enable3dControlsStatic();
    #endif

    m_pMainWnd = new CMainWnd();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

//
CTheApp::~CTheApp()
{
}

Aerodynamic file:

(This file is only an excerpt from the already existing XFOIL aerodynamics code written in FORTRAN. Only this excerpt has been functionally modified for use with the C++ interface, so only this section is shown here.)

    subroutine xfoil(nact,xdata,ydata,alpha,reyno,amach,acl,acd,acm,
&   airor)

!DEC$ ATTEMPTED DLLEXPORT :: XFOIL

C     ******************************************************
C     * Interactive Design and Analysis Code             *
C     * for Subsonic Airfoils.                          *
C     *                                               *
C     * Linear-vorticity panel formulation.            *
C     * Karman-Tsien compressibility correction.        *
C     * Lag-dissipation BL formulation.                *
C     * Transpiration viscous/inviscid matching.       *
C     * Solution by global Newton-Raphson.            *
C     *                                               *
C     * by Mark Drela (MIT)                            *
C     *                                               *
C     * modified for use with OptdesX                  *
C     * by P.Venkataraman (RIT)                         *
C     *                                               *

page A - 59
INCLUDE 'XFOIL.INC'

real alpha, reyno, amach
real xdata(1), ydata(1)
real acl, acd, acm
real aior
integer nact

c call initialization routine "init"

call INIT

CONTINUE
cc GO TO 500

nactl = nact+1

NB = nact

do 101 i = 1, NB
cc read(10,*) xb(i), yb(i)
cc print*, i, xb(i), yb(i)

do 101 continue

do 101 i = 1, NB
cc (continued)
xb(i) = xdata(i)
yb(i) = ydata(i)
101 continue

c calculate area (borrowed from sub load) -
changed by Venkat

AREA = 0.
DO 10 I=1, NB-1
RX = XB(I+1) + XB(I)
RY = YB(I+1) + YB(I)
DX = XB(I+1) - XB(I)
DY = YB(I+1) - YB(I)
DA = 0.25*(RX*DY - RY*DX)
cc DA = 0.5*DX*RY
cc if(i.eq.59) go to 10
AREA = AREA + DA
10 CONTINUE

area = area

IF(AREA.GE.0.0) THEN
LCLOCK = .FALSE.
ELSE
if area is negative (clockwise order), reverse coordinate order
LCLOCK = .TRUE.
c
DO 15 I=1, NB
   W1(I) = XB(I)
   W2(I) = YB(I)
15  CONTINUE
   DO 18 I=1, NB
      IBACK = NB - I + 1
      XB(I) = W1(IBACK)
      YB(I) = W2(IBACK)
18  CONTINUE
endif
50 CALL PANEL
cc   call panell(nact)
cc   the interactive and plot facility are being
cc   suppressed
cc Subroutine OPER is being replaced by values
reinf = reyno
minf = amach
if(minf.eq.0.0) goto 1518
CALL COMSET
cc Subroutine CPCALC and CLCALC are run with new value of MINF
cc CALL CPCALC
CALL CLCALC
LVCONV = .FALSE.
1518 continue
cc alpha is picked up
cc ADEG = alpha
LALFA = .TRUE.
ARAD = DTOR*ADEG
QINF = 1.0
CALL SPECAL
c IF(ABS(ARAD-AVISC) .GT. 1.0E-5) LVCONV = .FALSE.
IF(LVISC) CALL VISCAL
cc capture aerodynamic information for transfer to OptdesX
cc acl = cl
acd = cd
acm = cm
airor = RMSBL
if(lvisc) then
aclcd = cl/cd
aclend = cl**(1.5)/cd
else
cd = 1
aclcd = cl/cd
aclend = cl**(1.5)/cd
endif
ccwrite(41,*alpha, reyno
ccwrite(41,*acl, acd, acm, aarea
cc
c  close (unit = 10)

return
END
Appendix B: DataRead Program

datatdll.cpp file:

```cpp
#include <fstream.h>
#include <afxwin.h>

#define MaxDataPoints 200

text
_declspec(dllexport)

//function calls
int DataRead (CString infilenm, CString outfilenm)
{
    ifstream infile(infilenm);
    ofstream outfile(outfilenm);

    int i=0;
    int j=0;
    double (*datain)[2];
    int databreak=0;
    int totpoints=0;
    double tempval1=0, tempval2=0;
    double temp1[2];
    int flag=0;

    datain = new double [MaxDataPoints][2];

    //reading in data
    //a break is determined where the data switches directions on the x-coordinate
    while (infile >> datain[totpoints][0])
    {
        infile >> datain[totpoints][1];

        if (flag == 0)
        {
            if (totpoints >= 2)
            {
                if ((datain[totpoints][0] >= datain[totpoints-1][0]) &&
                    (datain[totpoints-1][0] < datain[totpoints-2][0]))
                {
                    databreak = totpoints;
                    flag = 1;
                }
            }
            if ((datain[totpoints][0] <= datain[totpoints-1][0]) &&
                (datain[totpoints-1][0] > datain[totpoints-2][0]))
            {
                databreak = totpoints;
            }
        }
    }
}
```

```
flag = 1;
}
}
totpoints++;

// calculates an average of y-coordinates
// this determines which data set is the top and which is the bottom surface
for (i=0; i< databreak; i++)
    tempval1 = tempval1 + datain[i][1];
tempval1 = tempval1 / databreak;

for (i=databreak; i< totpoints; i++)
    tempval2 = tempval2 + datain[i][1];
tempval2 = tempval2 / (totpoints - databreak);

if (tempval1 == tempval2)
    return(1);

// a set of if statements to sort the data into two ordered columns
// increasing from x=0 to x=1 for top and bottom
for (i=0, i<(databreak-1); i++)
{
    for (j=(i+1); j<databreak; j++)
    {
        if (datain[j][0] < datain[i][0])
        {
            temp1[0] = datain[j][0];
temp1[1] = datain[j][1];
            //
datain[j][0] = datain[i][0];
datain[j][1] = datain[i][1];
            //
datain[i][0] = temp1[0];
datain[i][1] = temp1[1];
        }
    }
}

for (i=databreak; i< (totpoints-1); i++)
{
    for (j=(i+1); j<totpoints; j++)
    {
        if (datain[j][0] < datain[i][0])
        {
            temp1[0] = datain[j][0];
temp1[1] = datain[j][1];
            //
datain[j][0] = datain[i][0];
datain[j][1] = datain[i][1];
        }
    }
}
//
datain[i][0] = temp1[0];
datain[i][1] = temp1[1];

//writes the data to an output file
if (tempval1 > tempval2)
{
    for (i=0; i < databreak; i++)
        outfile << datain[i][0] << "\t" << datain[i][1] << "\n";
    outfile << -1 << "\n";
    for (i=databreak; i < totpoints; i++)
        outfile << datain[i][0] << "\t" << datain[i][1] << "\n";
}
if (tempval1 < tempval2)
{
    for (i=databreak; i < totpoints; i++)
        outfile << datain[i][0] << "\t" << datain[i][1] << "\n";
    outfile << -1 << "\n";
    for (i=0; i < databreak; i++)
        outfile << datain[i][0] << "\t" << datain[i][1] << "\n";
}

//close files
infile.close();
outfile.close();
delete [] datain;
return(0);
}

dataread.h file:

// dataread.h : main header file for the DATAREAD application

#if
#define AFX_DATAREAD_H__95577FBF_0548_11D3_8522_0000B44914B5_INCLUDED_ 
#define AFX_DATAREAD_H__95577FBF_0548_11D3_8522_0000B44914B5_INCLUDED_ 
#endif
#if_MSC_VER > 1000

page B - 3
#pragma once
#error include 'stdafx.h' before including this file for PCH

#include "resource.h" // main symbols

class CDatareadApp : public CWinApp
{
public:
    CDatareadApp();

    // Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDatareadApp)
    public:
        virtual BOOL InitInstance();
    //}}AFX_VIRTUAL

    // Implementation
    //{{AFX_MSG(CDatareadApp)
        // NOTE - the ClassWizard will add and remove member functions here.
        // DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

 dataread.cpp file:

// dataread.cpp : Defines the class behaviors for the application.
//
#include "stdafx.h"
#include "dataread.h"
#include "datareadDlg.h"

#ifdef_DEBUG
#define new DEBUGNEW
#else
static char THIS_FILE[] = __FILE__;
#endif

extern __declspec(dllimport) DataRead(CString infilenm, CString outfilenm);

BEGIN_MESSAGE_MAP(CDatareadApp, CWinApp)
    //
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //    DO NOT EDIT what you see in these blocks of generated code!
    //}
    }AFX_MSG
    ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

CDatareadApp::CDatareadApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

CDatareadApp theApp;

BOOL CDatareadApp::InitInstance()
{
    // Standard initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

#ifndefAFXDLL
    Enable3dControls();
    // Call this when using MFC in a shared DLL
#else

Enable3dControlsStatic();  // Call this when linking to MFC statically
#endif

CDataReadDlg dlg;
m_pMainWnd = &dlg;
int nResponse = dlg.DoModal();
if (nResponse == IDOK)
{
    DataRead(dlg.Getinfile(), dlg.Getoutfile());
    // TODO: Place code here to handle when the dialog is
    // dismissed with OK
}
else if (nResponse == IDCANCEL)
{
    // TODO: Place code here to handle when the dialog is
    // dismissed with Cancel
}

// Since the dialog has been closed, return FALSE so that we exit the
// application, rather than start the application's message pump.
return FALSE;

stdafx.h file:

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//
#if !defined(AFX_STDAFX_H_95577FC3_0548_11D3_8522_0000B44914B5_INCLUDED_)
define AFX_STDAFX_H_95577FC3_0548_11D3_8522_0000B44914B5_INCLUDED_
#if_MSC_VER>1000
#pragma once
#endif // _MSC_VER > 1000
#define VC_EXTRALEAN    // Exclude rarely-used stuff from Windows headers
#include <afxwin.h>    // MFC core and standard components
#include <afxext.h>    // MFC extensions
#include <afxdtl.h>    // MFC support for Internet Explorer 4 Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>    // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.
stdafx.cpp

// stdafx.cpp : source file that includes just the standard includes
// dataread.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

datareadDlg.h file:

// datareadDlg.h : header file

#include <stdafx.h>

class CDatareadDlg : public CDialog
{
// Construction
public:
    CDatareadDlg(CWnd* pParent = NULL); // standard constructor
    CString Getinfile(); // accessor for infile field
    CString Getoutfile(); // accessor for outfile field

// Dialog Data
    enum { IDD = IDD_DATAREAD_DIALOG }; // NOTE: the ClassWizard will add data members here
};

// ClassWizard generated virtual function overrides
protected:

virtual void DoDataExchange(CDataExchange* pDX);  // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:
    HICON m_hIcon;

CString infile;
CString outfile;

// Generated message map functions
//{{AFX_MSG(CDatareadDlg)
virtual BOOL OnInitDialog();
afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
//}}AFX_MSG
afx_msg void HandleButton1();
afx_msg void HandleButton2();
DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.
#endif
//!defined(AFX_DATAREADDLG_H__95577FC1_0548_11D3_8522_0000B44914B5__INCLUDED
//UDEDJ )

**datareadDlg.cpp file:**

// datareadDlg.cpp : implementation file

#include <afxwin.h>
#include "stdafx.h"
#include "dataread.h"
#include "datareadDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// CAAboutDlg dialog used for App About

class CAAboutDlg : public CDialog
public:
    CAboutDlg();

// Dialog Data
    // {{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX }; // }}AFX_DATA

    // ClassWizard generated virtual function overrides
    // {{AFX_VIRTUAL(CAboutDlg)
    protected:
        virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    // }}AFX_VIRTUAL

// Implementation
protected:
    // {{AFX_MSG(CAboutDlg)
    DECLARE_MESSAGE_MAP()
    // }}AFX_MSG

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
    { // {{AFX_DATA_INIT(CAboutDlg)
        // }}AFX_DATA_INIT
    }

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
    { CDialog::DoDataExchange(pDX);
        // {{AFX_DATA_MAP(CAboutDlg)
        // }}AFX_DATA_MAP
    }

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    // {{AFX_MSG_MAP(CAboutDlg)
    // No message handlers
    // }}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
// CDatareadDlg dialog

CDatareadDlg::CDatareadDlg(CWnd* pParent /*=NULL*/)
    : CDIalog(CDatareadDlg::IDD, pParent)
    { // {{AFX_DATA_INIT(CDatareadDlg)
        // NOTE: the ClassWizard will add member initialization here
        // }}AFX_DATA_INIT
        // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

//accessor for infile field
CString CDatareadDlg::Getinfile()
{
    return(infile);
}

//accessor for outfile field
CString CDatareadDlg::Getoutfile()
{
    return(outfile);
}

void CDatareadDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_EDIT1, infile);
    DDX_Text(pDX, IDC_EDIT2, outfile);

    BEGIN_MESSAGE_MAP(CDatareadDlg, CDialog)
        ON_WM_SYSCOMMAND()
        ON_WM_PAINT()
        ON_WM_QUERYDRAGICON()
    END_MESSAGE_MAP()
}

BEGIN_MESSAGE_MAP(CDatareadDlg, CDialog)
    ON_WM_SYSCOMMAND0
    ON_WM_PAINT0
    ON_WM_QUERYDRAGICON0
    ON_BN_CLICKED(IDC_BUTTON1, HandleButton1)
    ON_BN_CLICKED(IDC_BUTTON2, HandleButton2)
END_MESSAGE_MAP()

BOOL CDatareadDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Add "About..." menu item to system menu.

    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0x0000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
if (pSysMenu != NULL)
{
    CString strAboutMenu;
    strAboutMenu.LoadString(IDS_ABOUTBOX);
    if (!strAboutMenu.IsEmpty())
    {
        pSysMenu->AppendMenu(MF_SEPARATOR);
        pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
    }
}

// Set the icon for this dialog. The framework does this automatically
// when the application's main window is not a dialog
SetIcon(m_hIcon, TRUE);           // Set big icon
SetIcon(m_hIcon, FALSE);          // Set small icon

// TODO: Add extra initialization here
return TRUE;  // return TRUE unless you set the focus to a control

void CDatareadDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}

// If you add a minimize button to your dialog, you will need the code below
// to draw the icon. For MFC applications using the document/view model,
// this is automatically done for you by the framework.

void CDatareadDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
GetClientRect(&rect);
int x = (rect.Width() - cxIcon + 1) / 2;
int y = (rect.Height() - cyIcon + 1) / 2;

// Draw the icon
dc.DrawIcon(x, y, m_hIcon);
}
else
{
    CDialog::OnPaint();
}

// The system calls this to obtain the cursor to display while the user drags
// the minimized window.
HCURSOR CDatareadDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}

//This function was added manually to handle the push button IDC_BUTTON1
void CDatareadDlg::HandleButton1()
{
    //sets the file types for file open dialog box
    CString FileFilter = "All Files (*.*)|*.*|Dat Files (*.dat)|*.dat|Text Files (*.txt)|*.txt||";
    //declares an open file dialog box
    CFileDialog dlgFileOpen(TRUE, 0, 0, OFN_FILEMUSTEXIST, (LPCTSTR)FileFilter, this);
    if(IDCANCEL == dlgFileOpen.DoModal())
        return;
    infile = dlgFileOpen.GetPathName(); //gets the path name from the open box
    UpdateData(FALSE); //updates the main dialog CDatareadDlg to reflect the
    //changes
    MessageBeep(-1);
}

//This function was added manually to handle the push button IDC_BUTTON2
void CDatareadDlg::HandleButton2()
{
    //sets the file types for file save dialog box
    CString FileFilter = "Text Files (*.txt)|*.txt|All Files (*.*)|*.*||";
    CString FileExt = ".txt";
    //declares a save file dialog box
    CFileDialog dlgFileSave(FALSE, FileExt, 0, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT, (LPCTSTR)FileFilter, this);
    if(IDCANCEL == dlgFileSave.DoModal())
        return;
outfile = dlgFileSave.GetPathName();  // gets the path name from the dialog box
UpdateData(FALSE);  // updates the main dialog CDatareadDlg to reflect the
                     // changes
MessageBeep(-1);
}

resource.h file:

//{{NO_DEPENDENCIES}}}
// Microsoft Developer Studio generated include file.
// Used by dataread.rc

#define IDM_ABOUTBOX 0x0010
#define IDD_ABOUTBOX 100
#define IDS_ABOUTBOX 101
#define IDD_DATAREAD_DIALOG 102
#define IDR_MAINFRAME 128
#define IDC_EDIT1 1000
#define IDC_EDIT2 1001
#define IDC_BUTTON1 1003
#define IDC_BUTTON2 1004

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS

#define _APS_NEXT_RESOURCE_VALUE 129
#define _APS_NEXT_COMMAND_VALUE 32771
#define _APS_NEXT_CONTROL_VALUE 1004
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif

dataread.rc file:

//Microsoft Developer Studio generated resource script.

#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS

// Generated from the TEXTINCLUDE 2 resource.

#include "afxres.h"
#ifndef APSTUDIO_READONLY_SYMBOLS

// English (U.S.) resources

#ifndef AFX_RESOURCE_DLL || defined(AFX_TARG_ENU)
    #ifdef _WIN32
        LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
        #pragma code_page(1252)
    #endif // _WIN32
#endif

#if !defined(APSTUDIO_INVOKED)
    #define _AFX_NO_SPLITTER_RESOURCES
    #define _AFX_NO_OLE_RESOURCES
    #define _AFX_NO_TRACKER_RESOURCES
    #define _AFX_NOPROPERTY_RESOURCES

    #if !defined(AFXRESOURCE_DLL) || defined(AFX_TARG_ENU)
        #ifdef _WIN32
            LANGUAGE 9, 1
            #pragma code_page(1252)
        #endif

        #include "res\dataread.rc2" // non-Microsoft Visual C++ edited resources
        #include "afxres.rc" // Standard components
    #endif
#endif

#endif // APSTUDIO_INVOKED


1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include "afxres.h"
    \0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "#define _AFX_RESOURCE_DLL
    "#define _WIN32"
    "LANGUAGE 9, 1"
    "#pragma code_page(1252)"
    "#ifdef _WIN32"
    "#include "res\dataread.rc2" // non-Microsoft Visual C++ edited resources"
    "#include "afxres.rc" // Standard components"
    "#endif"
    "\0"
END
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDR_MAINFRAME ICON DISCARDABLE "res\dataread.ico"


DOCUMENTATION

IDD_ABOUTBOX DIALOG DISCARDABLE 0, 0, 235, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About dataread"
FONT 8, "MS Sans Serif"
BEGIN
ICON IDR_MAINFRAME,IDC_STATIC,11,17,20,20
LTEXT "dataread Version 2.0",IDC_STATIC,40,10,119,8,
SS_NOPREFIX
LTEXT "Copyright (C) 1999",IDC_STATIC,40,25,119,8
DEFPUSHBUTTON "OK",IDOK,178,7,50,14,WS_GROUP
END

IDD_DATAREAD_DIALOG DIALOGEX 0, 0, 238, 121
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "Data Organization Utility"
FONT 8, "MS Sans Serif"
BEGIN
DEFPUSHBUTTON "OK",IDOK,62,100,50,14
PUSHBUTTON "Cancel",IDCANCEL,125,100,50,14
LTEXT "Input File:.",IDC_STATIC,6,19,29,8
LTEXT "Output File:.",IDC_STATIC,7,55,34,8
EDITTEXT IDC_EDIT1,6,32,169,14,ES_AUTOHSCROLL
EDITTEXT IDC_EDIT2,7,69,169,14,ES_AUTOHSCROLL
PUSHBUTTON "Browse...",IDC_BUTTON1,181,32,50,14
PUSHBUTTON "Browse...",IDC_BUTTON2,181,69,50,14
END

#ifndef_MAC

VS_VERSION_INFO VERSIONINFO
FILEVERSION 1,0,0,1
PRODUCTVERSION 1,0,0,1

#endif
FILEFLAGS 0x1L
#if _DEBUG
FILEFLAGS 0x1L
#else
FILEFLAGS 0x0L
#endif
FILEOS 0x4L
FILETYPE 0x1L
FILESUBTYPE 0x0L
BEGIN
BLOCK "StringFilelnfo"
BEGIN
BLOCK "040904B0"
BEGIN
VALUE "CompanyName", "\0"
VALUE "FileDescription", "dataread MFC Application\0"
VALUE "FileVersion", "1, 0, 0, 1\0"
VALUE "InternalName", "dataread\0"
VALUE "LegalCopyright", "Copyright (C) 1999\0"
VALUE "LegalTrademarks", "\0"
VALUE "OriginalFilename", "dataread.EXE\0"
VALUE "ProductName", "dataread Application\0"
VALUE "ProductVersion", "1, 0, 0, 1\0"
END
END
BLOCK "VarFilelnfo"
BEGIN
VALUE "Translation", 0x409, 1200
END
END
#endif // !_MAC

////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//
#endif APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
IDD_ABOUTBOX, DIALOG
BEGIN
LEFTMARGIN, 7
RIGHTMARGIN, 228
TOPMARGIN, 7
BOTTOMMARGIN, 48
END

IDD_DATAREAD_DIALOG, DIALOG
BEGIN
  LEFTMARGIN, 6
  RIGHTMARGIN, 231
  TOPMARGIN, 7
  BOTTOMMARGIN, 114
END
END
#include "res\dataread.rc2"
#include "afxres.rc"

END
END
END
END