Spring 5-2001

Modeling of a Two-Dimensional Airfoil Using Boundary Element Method

Brent Derek Weinberg
University of Tennessee-Knoxville

UNIVERSITY HONORS PROGRAM

SENIOR PROJECT - APPROVAL

Name: Brent Weinberg

College: Engineering Department: Engineering Science (Biomed)

Faculty Mentor: Dr. Joe Iannelli

PROJECT TITLE: Modeling of a Two-Dimensional Airfoil using Boundary Element Method

I have reviewed this completed senior honors thesis with this student and certify that it is a project commensurate with honors level undergraduate research in this field.

Signed: Joe Iannelli, Faculty Mentor

Date: 5/1/2001

Comments (Optional):

Modeling of a Two-Dimensional Airfoil using Boundary Element Method

Brent D. Weinberg

Mentor: Dr. Joe Iannelli

Senior Honors Project

May 7, 2001

Table of Contents

Abstract

Most external flow situations, specifically that of air around a thin airfoil with a low mach number, can be specified as irrotational and inviscid flow fields. Since for all such fields Laplace's equation is always satisfied, a system of partial differential equations describing the flow can be solved for a solution that is both useful and unique. Simulating the desired cross section as a series of nodes and boundary elements and then manipulating the results into a desired form elucidates the pressure distribution across the airfoil as well as the coefficent of lift. However, solution of the system is nonunique unless Kutta's condition is observed on the trailing edge of the airfoil; in other words, the velocity of the fluid at the trailing edge of the airfoil is equal to zero. By observing this condition the solution becomes unique. One method that can be used to evaluate this technique is by selecting specific Joukowski airfoil profiles that have been solved analytically and comparing the computational solution to the analytical result. Once the technique and differential equation solver have been proven to be accurate, it can then be applied to any variety of internal or external flows that are irrotational and inviscid. This project uses a boundary element differential equation solver written in C to solve for this unique solution and then determine the corresponding pressures and lift coefficient for the configuration and velocity field. Some difficulties in obtaining and evaluating correct results occurred, and they are presented here along with some the results of the project.

Introduction

In the solution of fluid mechanics problems, analytical solutions are very difficult and often impossible to obtain directly. For this reason, empirical analysis and numerical techniques allow for analysis of situations far more complex than would be otherwise achievable. In the numerical analysis of fluid situations, two main approaches exist: the finite element method or the boundary element method.

The finite element consists of dividing the entire flow field (or the portion which is to be analyzed) into small but finite elements over which the flow can be approximated. These finite elements can then be assembled into a mesh and the desired variables solved for by assembling a large number of simultaneous equations. The solution of these simultaneous equations then yields a numerically approximated value at each node of the generated mesh. This solution involves very intricate mesh generation and arrives at a solution in much the same method that would be used in a solid finite element modeling analysis. Finite element modeling of flow fields can be done in either two or three dimensions and tailored to suit the complexity of the problem.

The second analysis method is the boundary element method, in which only the boundary of the flow field is considered. In this technique, this boundary is broken up into small segments which each contribute some effect to the entire flow field. By selecting appropriate boundary conditions and solving for the values on each of the boundary elements, the entire state of the boundary is then known. To calculate values of the flow field at positions that are not along the boundary, each boundary element can be taken as contributing to the state of the flow depending on its position. In this method, the determination of the entire flow field is accomplished by solving for the values only along the flow boundary. In many situations, such as in the flow of air over an airfoil, flow along the boundary is the most important consideration (Munson 373). The use of the boundary element method is quite useful and is the procedure considered in this project.

In the use of the boundary element method, two main methods exist in solving for the flow field. One method involves considering "boundary sources, doublets and vortices of appropriate intensity (Iannelli 1)," while the method in this paper uses the value of the potential functions and the normal derivatives of these potential functions on these boundaries. This project investigates the use of this second method, or use of potential function values along the boundary of an airfoil in order to solve for the flow field around an airfoil. After solving the flow field in this manner, associated values of interest, such as the pressure distribution across the airfoil and the associated coefficient of lift can also be determined.

This paper uses two main elements, enforcement of Laplace's equation along the boundary in conjunction with Kutta's condition along the trailing edge of an airfoil to determine the velocity at each point along the airfoil. From the velocity, associated values for pressure and lift are then evaluated.

Background

Basic Considerations

In the consideration of the use of the boundary element method (BEM) to determine air flow around an airfoil, two major conditions are enforced in this method. First, the flow is considered to be a true "potential flow." This assumption is a fairly drastic one in that few fluid mechanics situations are entirely incompressible, irrotational, and inviscid. Second, Kutta's condition is taken to hold true for the flow as it is considered in this simulation. This condition specifies that the velocity of the flow along the trailing edge be zero, or stated otherwise, that the stagnation point along the airfoil coincides with the rear edge of the airfoil.

The assumption that the flow around an airfoil is a "potential flow" means that the flow can be considered incompressible, irrotational, and inviscid. Incompressibility states that the density of the fluid remains constant in all situations that are considered.

$$\rho = \text{constant}$$

Liquids and solids are most often considered incompressible because a change in density requires a very high change in pressure, and for most situations the pressure change is not significant to affect the density. While density changes in a gas require much smaller pressure changes than solids or liquids, gases can also be considered incompressible if changes for small changes in pressure. Generally, this assumption will hold true within a reasonable accuracy for any flow with a Mach number less than 0.3 (Munson 141). For flows with $Ma < 0.3$, the incompressible solutions most often differs from the compressible solution by $< 2\%$. Assuming that the fluid is inviscid is a similar simplification. Although no real fluids have a zero viscosity, fluids can often be considered to have zero viscosity because pressure effects so thoroughly dominate viscosity forces. In the consideration of an airfoil at low velocities, this assumption is quite valid. The last main assumption is that of irrotational flow. In order for a flow to be irrotational, the derivative of the y-component of the velocity in the x-direction must be equal to the derivative of the x-component of the velocity in the y-direction.

$$\frac{d}{dx}v = \frac{d}{dy}u$$

u	x component of velocity
v	y component of velocity

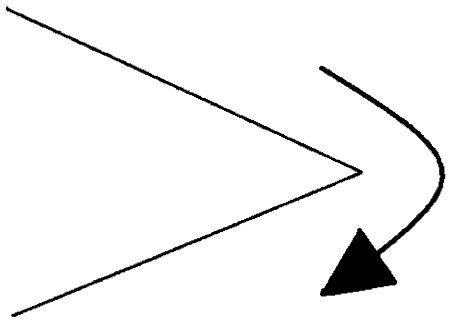
Again, while most flows are not irrotational, they can be considered irrotational when the rotation generated by viscous effects is small. For most flows, the flow is largely irrotational outside of some boundary layer in which viscous effects dominate. In this paper, all flows are considered to be ideal, i.e. incompressible, irrotational, and inviscid.

Application of Kutta's condition along the trailing edge completes the general conditions that make solution of the flow field using the BEM possible. Enforcement of Kutta's condition along the trailing edge of the boundary makes the solution both unique and meaningful, which is necessary for the successful solution of the flow field around an airfoil. Without successful implementation of this stipulation, the solution is neither. As was shown by Joukowski, a rounded airfoil with a pointed trailing edge has only one possible irrotational solution, and successful implementation of the Kutta condition ensures that the code arrives at that solution rather than a generally uninformative arbitrary solution. In his famous book, Theory of Flight, Richard von Mises explains the essential elements of Kutta's condition in terms of the surface vorticity:

> It can easily be proved, also, that when the cross section of the sheet is curved and if the vorticity is not uniformly distributed, the velocity at the ends

> becomes infinite unless the vortex density here is zero.... Kutta's condition stipulating a finite velocity at the trailing end therefore implies that the vortex density Γ' is zero here (von Mises 210).

While Kutta's condition directly states that the velocity at the trailing edge must be finite, in order for the vorticity to be equal to zero at this point the velocity must be equal to zero. This statement completes the condition that the velocity is equal to zero at the trailing edge and is shown in the following figure.



$$a = \frac{V^2}{r}$$

$$as\ \mathrm{r} \rightarrow 0, \mathrm{a} \rightarrow \infty$$

$$\text{unless V} = 0$$

Figure 1 Graphical representation of Kutta's condition

Since the radius of curvature goes to zero at the sharp point on the trailing edge, then the velocity of the fluid at the trailing edge goes to zero. While this brief explanation only touches the subject, the main idea that the velocity at the trailing edge must be zero is essential.

In the use of potential functions to determine the flow field around an airfoil, the function specifically referred to in this formulation is the stream function, ψ (x,y). This function is specifically defined in terms of the velocity components such that

$$\frac{\partial}{\partial y}\psi = u \qquad \frac{\partial}{\partial x}\psi = -v$$

The boundary element formulation described in later portions of this paper applies, either directly or indirectly, all of the stipulations specified in this section as an integral part of the formulation of the solution.

Specific Considerations

The specific formulation of the method used to solve for the velocities at the boundary nodes is as exactly as shown by Iannelli, Grillo, and Tulumello. The treatment of the solution of the potential function through the boundary element model in this introduction is by no means complete and is intended to serve only as a general overview, and it is of note that the following description is neither novel nor the original work of the author.

As described in the introduction, "potential flows" are flows that can be described as inviscid, irrotational, and incompressible. When these three characteristics are met, the main governing equation is determined by the Laplacian operator ∇^2 taken on the potential function. When expressed in two dimensions, this relationship is seen as

$$\frac{\partial^2}{\partial x^2}\psi + \frac{\partial^2}{\partial y^2}\psi = 0$$

It follows from this relationship that if the potential function is known, then the velocity can be determined by taking the partial derivatives of ψ (Munson 337). One additional characteristic of the stream function ψ is that it is constant along a streamline, and since the boundary of an airfoil forms a streamline, the value of ψ along the boundary is equal to a constant. By dividing the stream function into two components, a component that exists due to the free-stream flow and a component due to the perturbation generated by the airfoil, superposition allows that

$$\psi_{tot} = \psi_{pert} + \psi_{freestream}$$

Additionally, by integrating the values of the velocity, the value of $\psi_{\text{freestream}}$ in a uniform velocity field at an angle of attack α is

$$\psi_{\text{freestream}} = V_{\text{inf}} \cdot (y \cdot \cos(\alpha) - x \cdot \sin(\alpha))$$

Substitution in the previous equation gives a general equation for ψ_{pert} in terms of the free stream velocity and the constant value of ψ across the airfoil surface, which is taken to be ψ_{TE}. This value of the stream function on the boundary of the airfoil is not allowed to have an arbitrary value, but is solved for

$$\psi_{\text{tot}} = \psi_{TE}$$

$$\psi_{\text{pert}} = \psi_{\text{tot}} - \psi_{\text{freestream}}$$

$$\psi_{\text{pert}} \rightarrow \psi_{TE} - V_{\text{inf}} \cdot (y \cdot \cos(\alpha) - x \cdot \sin(\alpha))$$

This slight alteration of the total stream function allows for the setup of the numerical system to solve for the perturbation velocity at each point on the boundary. Since it is not the value of ψ specifically that is of interest, the derivative of the stream function is taken with respect to **n**, an outward pointing unit vector. Because the derivative of the stream function is velocity, the derivative of ψ with respect to n give the velocity of the flow along the surface of the airfoil.

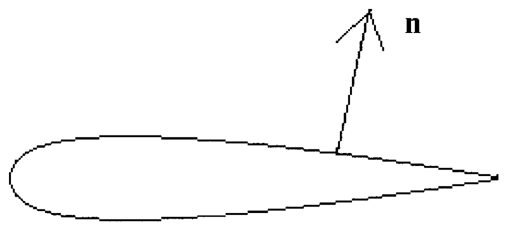


Figure 2 Schematic of the outward normal vector, n

Application of the Kutta condition at the trailing edge yields two equations that are solved simultaneously along with the rest of the linear system, the derivative of ψ with respect to the positive outward derivative, and the derivative of ψ with respect to the negative outward derivative.

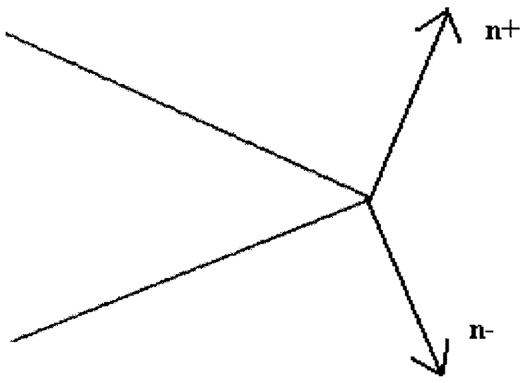


Figure 3 The two outward pointing normal vectors at the trailing edge

Since the derivative of ψ at the trailing edge must equal zero, this yields two equations:

$$-\left(\frac{\partial \psi_{tot}}{\partial y}\right)_{TE}^{-} \cdot n_y^{-} - \left(\frac{\partial \psi_{tot}}{\partial x}\right)_{TE}^{-} \cdot n_x^{-} = -\left(\frac{\partial \psi_{tot}}{\partial n}\right)_{TE}^{-} = 0$$

$$-\left(\frac{\partial \psi_{tot}}{\partial y}\right)_{TE}^{+} \cdot n_y^{+} - \left(\frac{\partial \psi_{tot}}{\partial x}\right)_{TE}^{+} \cdot n_x^{+} = -\left(\frac{\partial \psi_{tot}}{\partial n}\right)_{TE}^{+} = 0$$

After some mathematical manipulations and the use of Green's Identity (Iannelli 4), the system of partial differential equations can be converted into a boundary integral along the boundary of the airfoil, which is denoted by the Greek letter, Ω. This boundary integral indicates that the integrals to be taken are line integrals evaluated across the entire surface of the airfoil.

$$\int_{\Omega} \psi \cdot \nabla^2 \cdot \phi \cdot d\Omega = \oint_{d\Omega} \psi \cdot \frac{\partial \phi}{\partial n} \cdot d\Gamma - \oint \phi \cdot \frac{\partial \psi}{\partial n} \cdot d\Gamma$$

where

$\phi(a,b) = \ln|a - b|$

and

$a = (x_1, y_1)$

$b = (x_2, y_2)$

After the formulation of this integral that is continuous along the boundary of the airfoil, the system can be discretized by defining a number of boundary nodes at which each of the specific values can be solved. Various other mathematical manipulations and the summation of all the simultaneous equations yields a single system of linear equations with N unknowns. While the final steps of the derivation are not shown here, the system is shown in summation notation:

$$\sum_{j=1}^{N-1} G_{i,j} \left(\frac{\partial \psi}{\partial n} \right)^{j} - \psi_{TOT}^{TE} \sum_{j=1}^{N} H_{i,j} = G_{i,N}^{-} \left(\frac{\partial \psi_{\text{inf}}}{\partial n} \right)^{N-} + G_{i,N}^{+} \left(\frac{\partial \psi_{\text{inf}}}{\partial n} \right)^{N+} - \sum_{j=1}^{N} H_{i,j} \psi_{\text{inf}}^{j}$$

The coefficients denoted by the capital letters G and H represent the terms of the integral formulation for solution in a linear system. All of the coefficients denoted by these letters can be determined numerically using known information. This system yields N equations, where N is the number of discrete nodes placed on the boundary of the airfoil, with N unknowns. The unknown values to be calculated in the system are

ψ_{TE} and $\frac{\partial}{\partial n} \psi_{\text{pert}}{}^{1}$ thru $\frac{\partial}{\partial n} \psi_{\text{pert}}{}^{N-1}$

The additional desired values, velocity, pressure, and lift, are all determined solely by the values of the stream function at the discrete locations, a process which is described further in the procedure section. Additionally, values at locations other than the locations chosen is possible by combining the contributions of all of the different nodes. Further information about the mathematics behind this discretization process is available in the references found at the end of this paper.

Procedure

Since the numerical techniques of the system described above were provided above in a C program graciously provided by Dr. Joe Iannelli, University of Tennessee, Department of Engineering Science and Mechanics, for use in this project, additional steps were taken to apply these numerical methods to the solution of a flow field about an airfoil. This problem consisted of altering and adapting the code provided in such a way as to implement the correct boundary conditions for the airfoil and then extracting the desired velocity, pressure, and ultimately, lift information about a given airfoil profile. Using the software provided, the additional programming and analysis was done using Metrowerks Codewarrior version 3.0 on a Toshiba 475MHz PC. The actual code used is presented in

Appendix A: Code, with the software as provided printed in small Courier font and the adaptations and additions made by the author in larger, bold Times Roman font to more emphasize the changes that were necessary to implement the code in this circumstance.

"C" Code

The C code used in this project was written using a modular approach, where each particular function performed a specific task and could be called from any location in the program. A typical progression of the program is listed in Figure 4, and the purpose of each function is listed in Table 1. The processes that are highlighted are those that were either written entirely or changed significantly for this project. Specifically, the added function *funcdef()* used the Joukowski transformation to determine the boundary nodes along a Joukowski airfoil, and *calculate()* determined the values of pressure and lift associated with each airfoil profile. Additionally, the code was designed such that it could easily be adapted to use any number of nodes, but values less than or around 1000 were typically used to limit computation time.

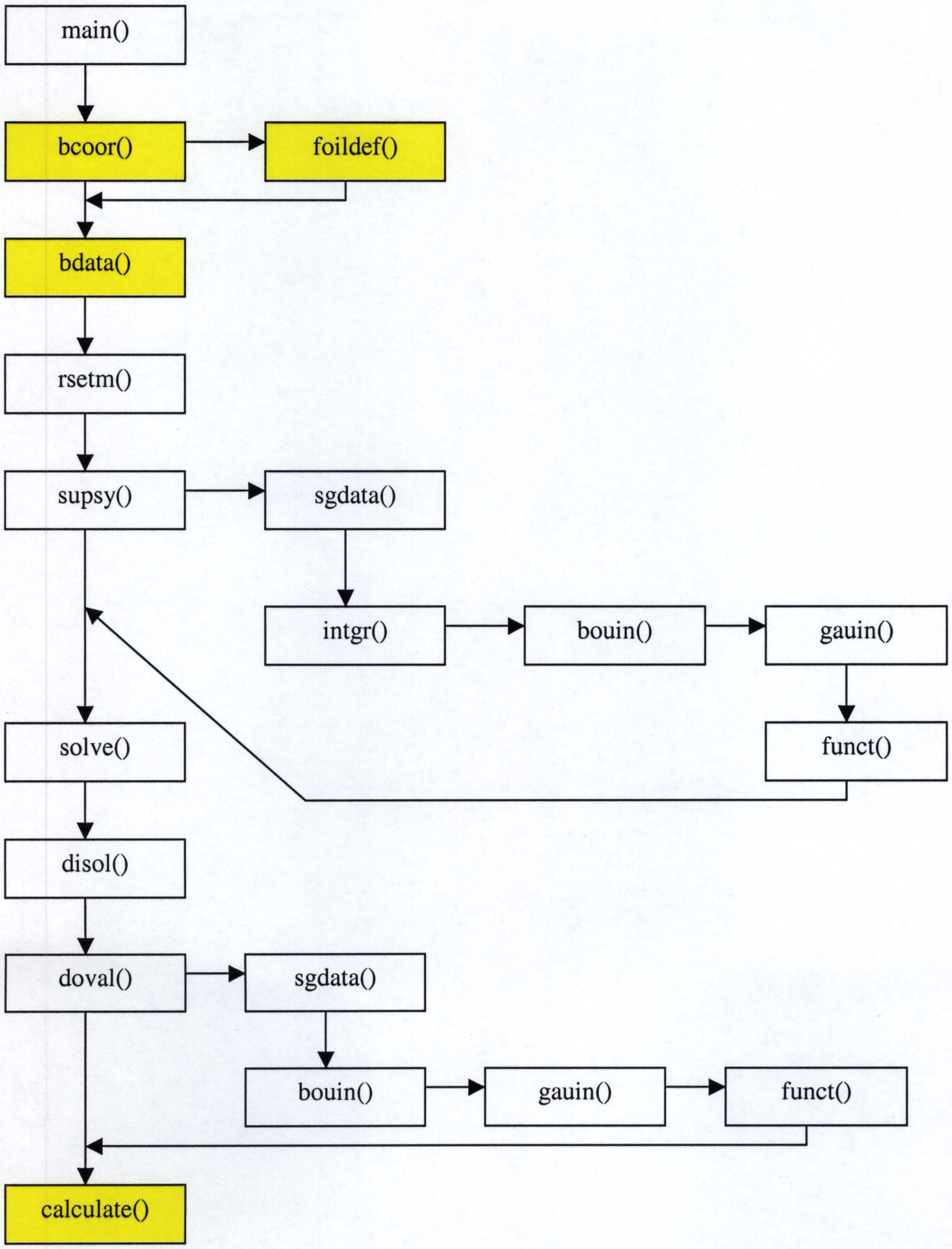


Figure 4 Flowchart of the processes in a typical run of the BEM code

Table 1 List of the programs functions and their designated task

Function	***Task***
bcoor()	defines the location of the boundary coordinates
bdata()	assigns boundary conditions along the boundary
bouin()	integrates all along the boundary at each node
calculate()	determines velocities, pressures, and lift from the output data
disol()	sorts the solved values and outputs them
dngeo()	checks boundary conditions and sets up solution method
doval()	determines the values of velocity at points not on the boundary
foildef()	defines the airfoil using the Joukowski transformation
funct()	determines the values h and g that are summed for H and G
gauin()	performs the Gaussian integral function
intgr()	determines the values of the integral coefficients H and G
rsetm()	resets the linear system a*b=c
sgdata()	defines the weights for the Gaussian integration function
solve()	solves the linear system with Gaussian elimination
supsy()	sets up linear system for solution

After computation, the results were output to MATLAB 5.0 to generate all visual presentation of data.

Joukowski Transformation

To generate an airfoil for the desired analysis, the widely understood Joukowski profile was chosen because it has analytical solutions available and provides an excellent method of generating any arbitrary number of points on an airfoil by simply transforming the coordinates of a circle. For this transformation, a circle of radius, R = 1.0 was chosen, and for the transformation the circle was offset from the origin by $x_c = 0.1$. Using the transformation

$$z' = z + \frac{\lambda^2}{z}$$

where z is the coordinates of the circle in complex form and λ is a value depending on the offset value of the center of the circle, the new airfoil z' was generated. A variety of different airfoil profiles can be generated based on the displacement of the center of the circle. As the x displacement of the center increases, the thickness of the foil increases; a similar increase in the y coordinate of the center of the circle increases the camber of the airfoil ("Conformational Mapping"). Each airfoil was generated using points evenly spaced around the original circle, which does not place them evenly around the airfoil. Some sample circles and their corresponding airfoils are displayed in **Figure 5** as an example of how the cross sections are

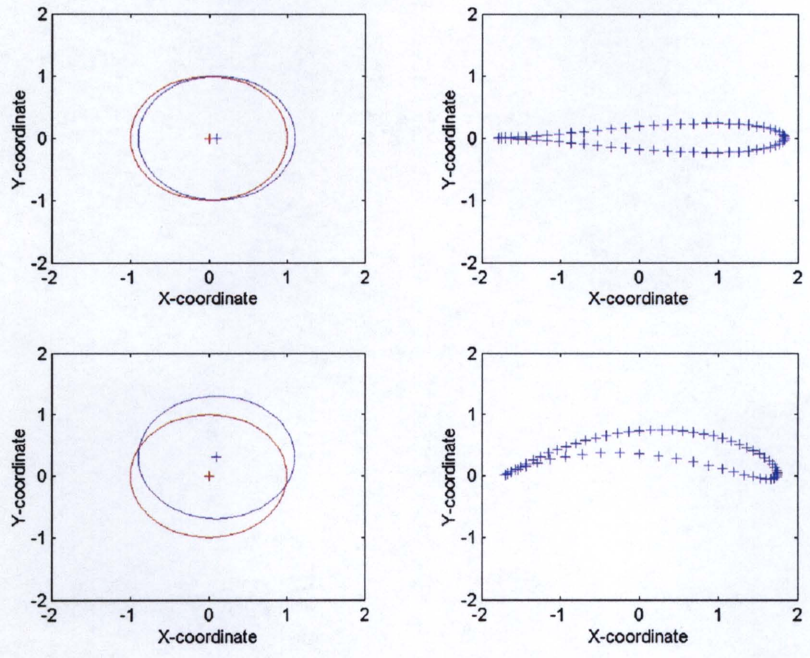


Figure 5 Sample Joukowski profiles generated from the circles in complex coordinates on the left

generated. The referenced web page from the University Genoa contains an excellent JavaScript tool which allows interaction with the profiles in an instructive way. Another valuable advantage of using a complex transform to generate the airfoil is that rotation is accomplished quite easily by merely multiplying the complex variable by the associated complex variable in exponential form.

$$z_a = z_0 \cdot e^{a \cdot j}$$

In this case, z_0 is an unrotated airfoil in complex form, and the angle of rotation is a. The corresponding values z_a represent an airfoil that has been rotated to have an angle of attack of a radians. Figure 6 displays four different Joukowski airfoils at varying angles of attack, the smallest being 0° and the largest being 15°. These figures merely act as an example of the advantages of using complex transforms to generate the coordinates of the airfoil. One disadvantage of using the Joukowski airfoil is that it is typically thicker than airfoils that are actually used. However, the lift values have been theoretically determined for these profiles, which is advantageous when investigating computational fluid dynamics.

All of the airfoils investigated in this project were generated using a x_c of 0.1 and a y_c of 0.0. This transformation generates an airfoil that is symmetric about the x-axis and relatively thin, both of which are useful for the simulation. The symmetry of the airfoil proves especially useful because at a zero angle of attack, the computed lift coefficient should be very nearly zero. This check provides an excellent method of determining whether lift coefficient calculations are valid. A sample of this airfoil is shown in Figure 7.

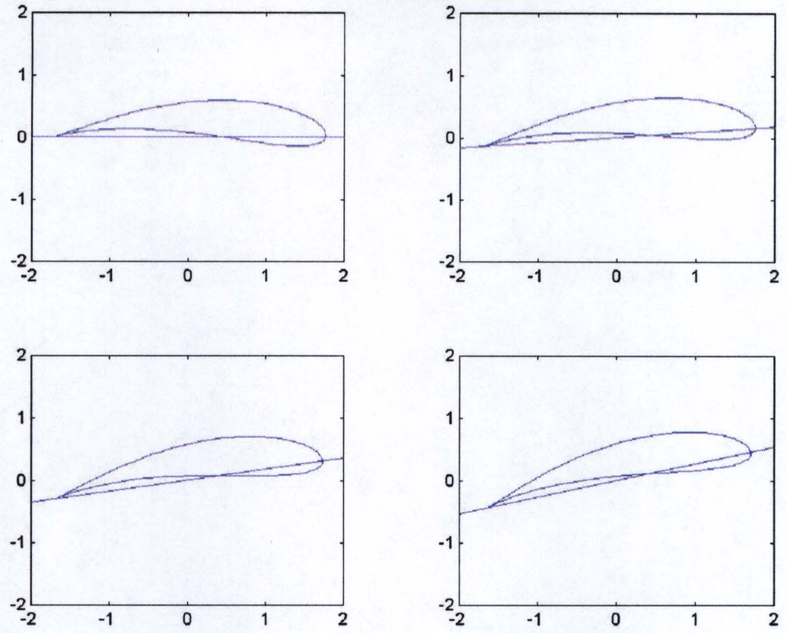


Figure 6 A Joukowski airfoil at various angles of attack: UL (0 deg), UR (5 deg), LL (10 deg), LR (15 deg)

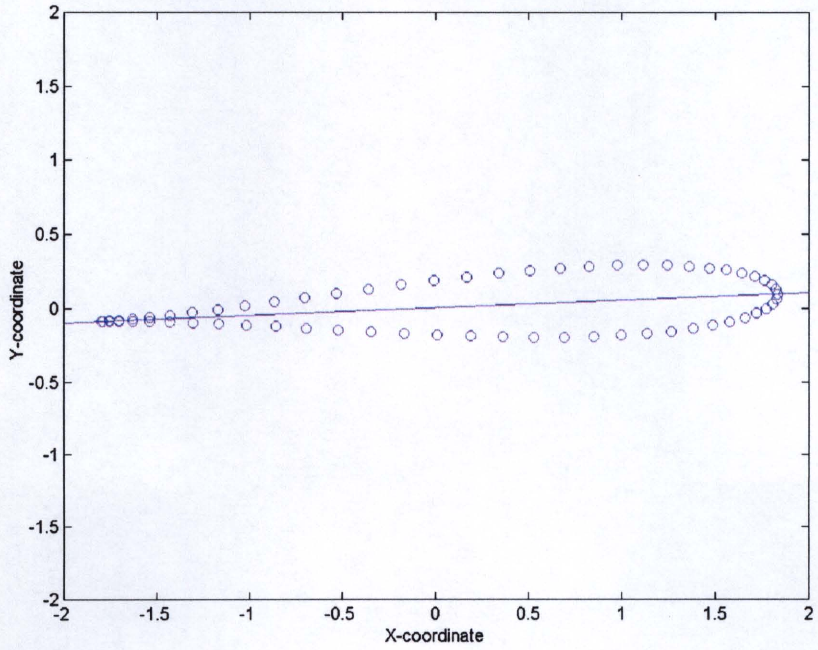


Figure 7 The profile of the Joukowski profile used in this project, shown with a 2.8 degree angle of attack

Pressure and Lift Calculations

After the computations were run, pressure and lift values were calculated to determine the effect of changing angle of attack on the airfoil. While the program output the perturbation velocities at each of the points, addition of the tangential components of the free stream velocity allowed determination of the total velocity at all of the boundary elements. From this velocity, pressure was determined through the use of Bernoulli's equation in the form

$$\Delta P = -\left(\frac{1}{2}\right)\rho \cdot U^2$$

After pressure was determined using this equation, pressures were summed to get the total force on the airfoil, and the dimensionless coefficient of lift was determined to be

$$C_l = \frac{L}{\frac{1}{2}\rho \cdot V_{inf}^{\ 2} \cdot A}$$

with the density of air taken to be 1.23 kg/m^3, which is the approximate standard value at atmospheric pressure. To check the calculated values for C_l they were recalculated by using the values for the pressure coefficient C_p. In addition to the values of the lift coefficient, the values of ψ_{TE} were tabulated to determine if the solution converged to the correct value. Additionally, pressures and velocities were recorded for comparison values. The BEM results are presented in the following section.

Results

Boundary Data

In the first set of results, the data presented compares the coefficient of lift for angles of attack varying from 0° to almost 15°. The data was tabulated using a simplistic model with only 16 nodes.

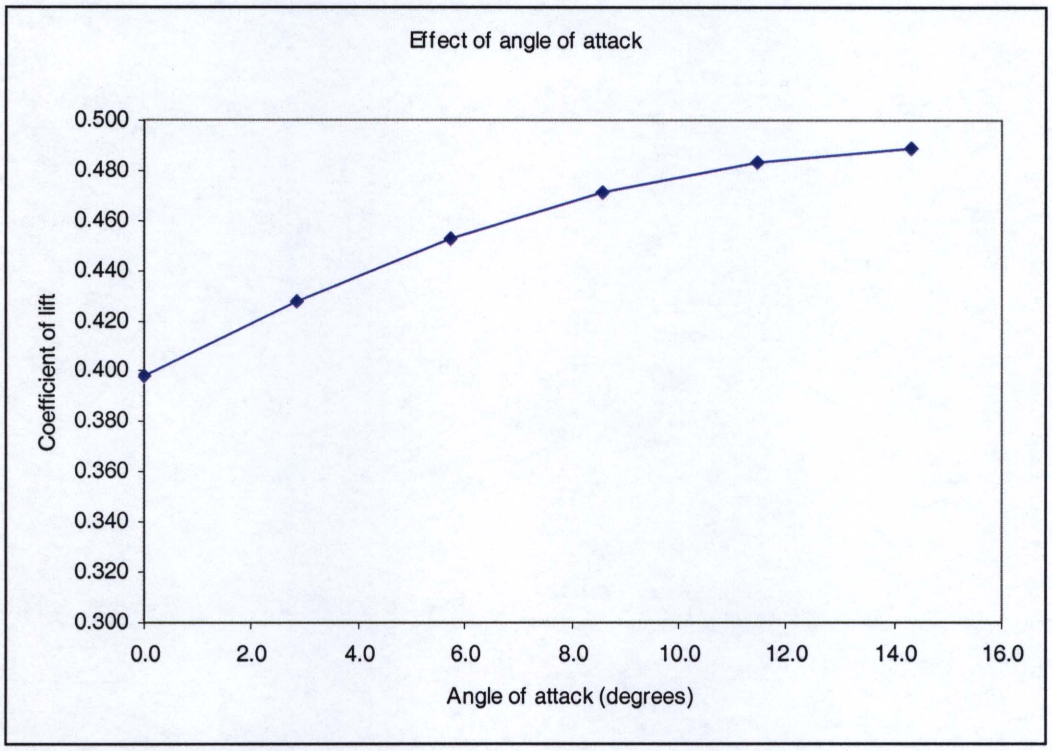


Figure 8 Graph presenting angle of attack's effect on lift

Angle of attack	*Lift coefficient*
0.0	0.398
2.9	0.428
5.7	0.452
8.6	0.471
11.5	0.483
14.3	0.489

As would be expected, the lift coefficient values increase as the angle of attack increases, but a couple of problems appear from within this data, the main one being that at angle 0° the lift coefficient is not zero. Since the airfoil is symmetrical on the x-axis, the theoretical lift is zero. This result suggests that these values are largely invalid.

To check the validity of the BEM technique, the values for C_l were calculated using a variety of different numbers of nodes. This technique was changed slightly by changing the position of the initial point $\mathbf{X_i}$, but unfortunately the lift coefficient climbs steadily as the node number increases.

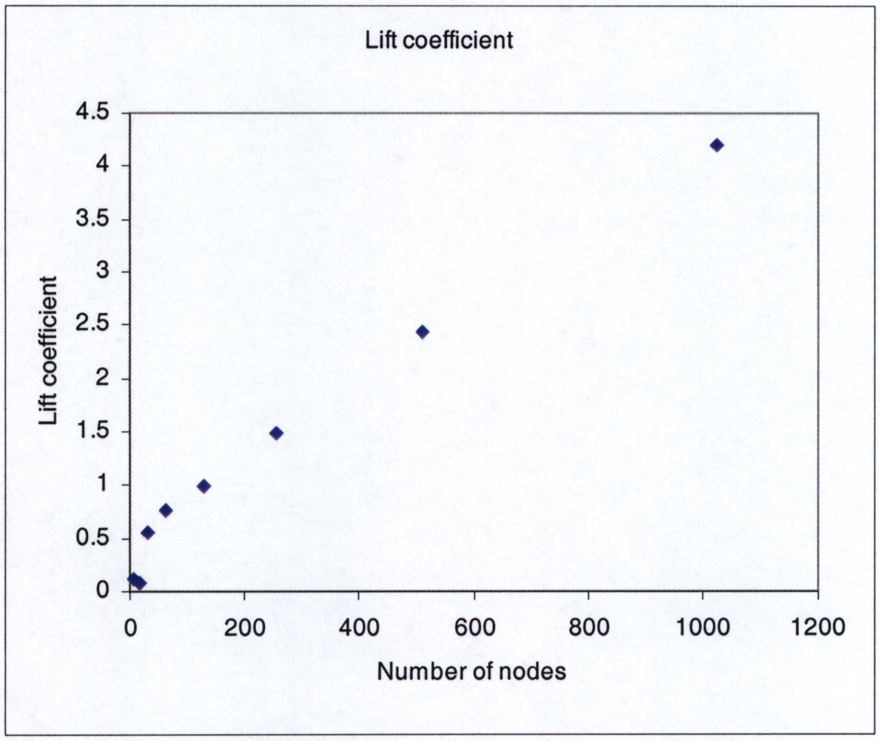


Figure 9 Graph showing instability of the lift coefficient

Nodes	*Lift coefficient*
8	0.107
16	0.084
32	0.554
64	0.766
128	0.979
256	1.475
512	2.439
1024	4.190

Since lift coefficient depending so strongly on the number of nodes suggested that the solution mechanism being used was unstable and had errors, the next results present the values of ψ_{TE} as the number of nodes in the BEM model changes. Contradicting strongly with the results taken from the lift coefficient, the value of the stream function along the boundary of the airfoil converges nicely with the increase in nodes, as is seen in the following graph and table.

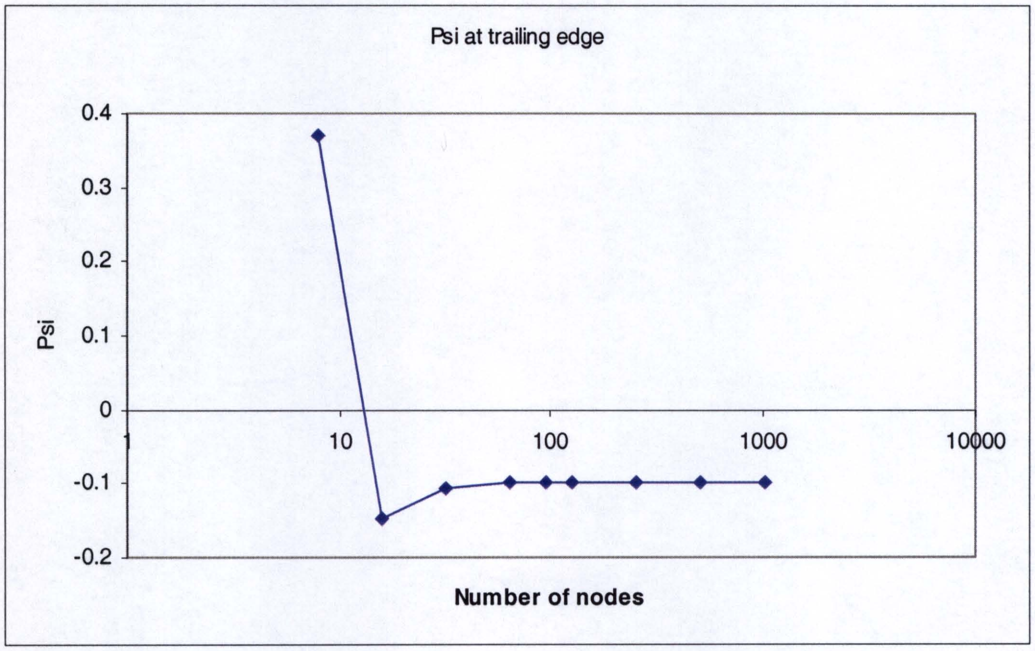


Figure 10 Fluctuation of psi along the trailing edge with the number of nodes

Nodes	*Psi at trailing edge*
8	0.36878883
16	-0.14705403
32	-0.10760685
64	-0.09931629
96	-0.09817919
128	-0.09794198
256	-0.09810054
512	-0.09851623
1024	-0.098746

These two somewhat contradictory results suggest errors in the boundary element formulation or in the application of the boundary conditions. The following results shed some light on this subject by displaying the pressure profile across the top of the airfoil as a percentage of the total chord length. On this graph, the leading edge of the airfoil would lie at zero on the x-axis and the trailing edge at one.

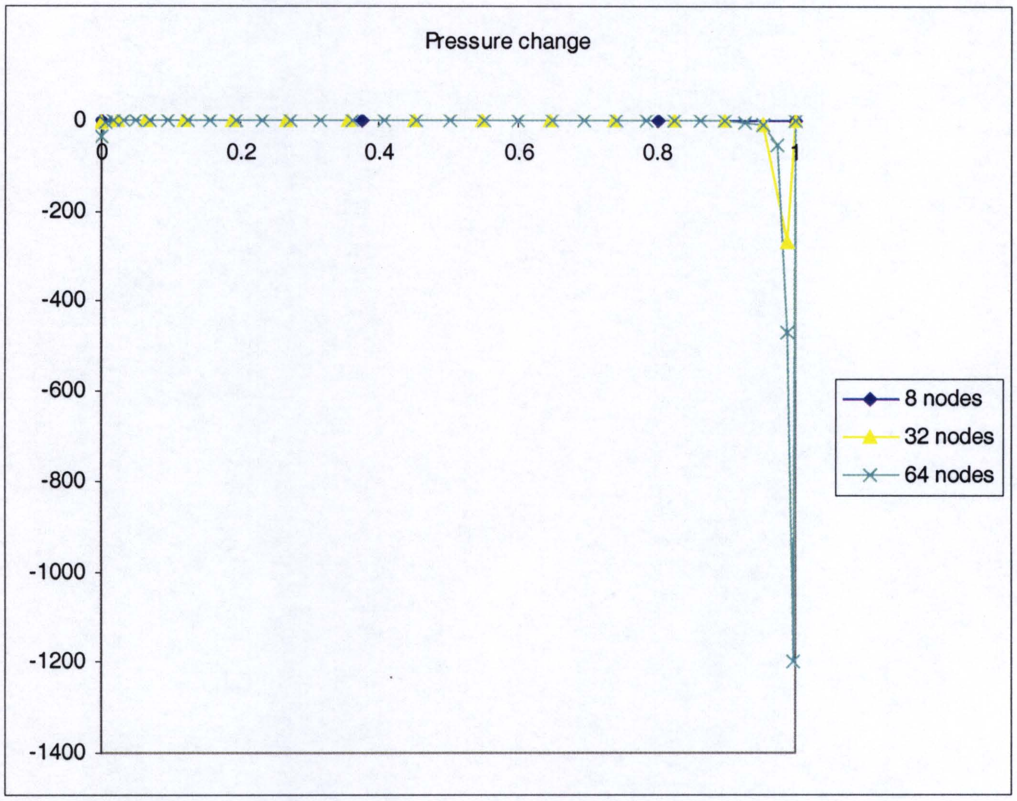


Figure 11 Pressure spike seen close to the trailing edge

From the data in Figure 11, the pressure experiences a very high drop at the nodes immediately adjacent to the trailing edge. This drop indicates an extremely large velocity very near the trailing edge, where the velocity has been necessarily set to equal zero. Such an error is indicative of computational errors or problems with the solution method.

External Data

To completely utilize the abilities of the computational code, the velocity values at a variety of points external to the airfoil were generated to determine the pressure field external to the airfoil. A matrix of values was generated at the points plotted in Figure 12.

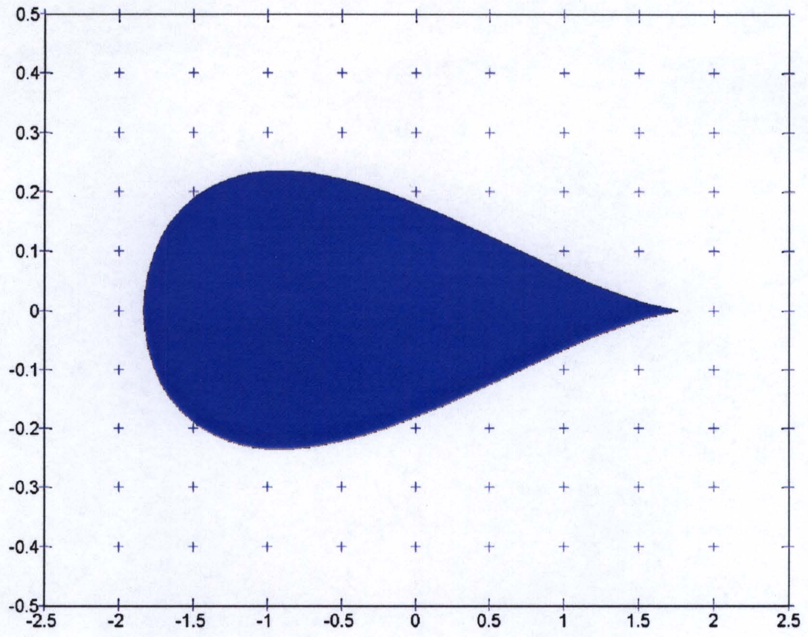


Figure 12 External points at which pressure was determined

Then, when the pressures were generated at each point they were plotted in a variety of ways. On the linear scale of the contour plot in Figure 13, seeing pressure differences is somewhat difficult because the differences are quite small. Changing the color scale would most likely make the task somewhat simpler, while an increase in the number of points plotted would also increase the accuracy of the plots. Figure 14 and Figure 15 present the same data, with the first being a contour plot in two dimensions and the latter being a surface plot showing higher pressure decreases with raised surfaces in three dimensions. Both of these graphs make the data more visible as well as illuminate the low pressure regions that are found over the rear portion of the airfoil. After solving for the boundary conditions, these external pressures were easily determined by the BEM program.

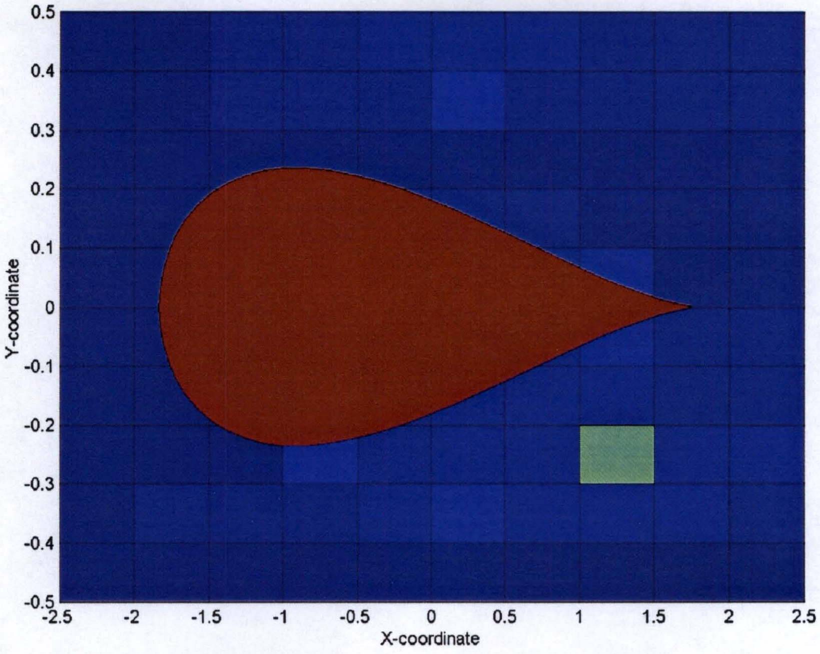


Figure 13 Color plot of the pressure drop around the airfoil.

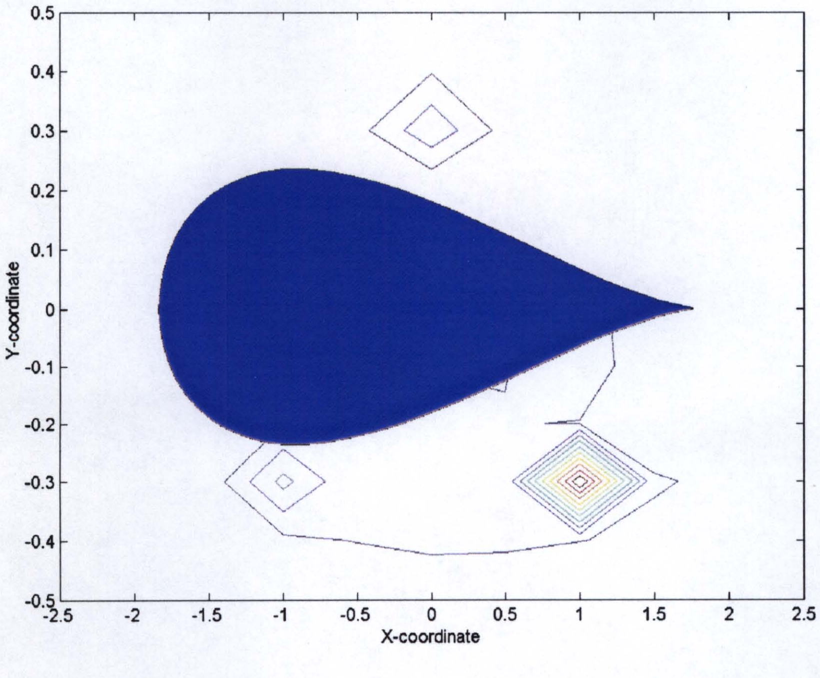


Figure 14 Contour plot of the same pressures found in Figure 13

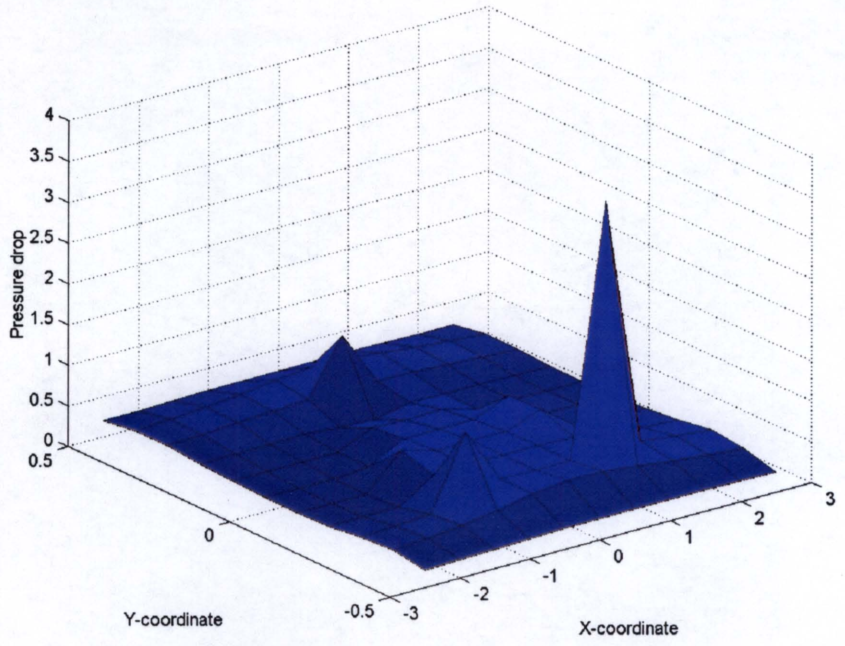


Figure 15 Surface plot of pressure drop around the airfoil surface

Conclusions

In conclusion, several main items stand out as important in the results of this project. While the boundary element method (BEM) is a fast and effective method for computationally determining the flow field around an airfoil, some problems exist that are evident in this paper. The instability of the lift coefficient as the number of nodes changes brings out a very subtle problem. What problems exist that could have generated this problem? Three possible solutions exist.

First, the boundary element solution method could be fundamentally corrupted with an error that was not found. In the amount of code involved in this type analysis, a simple mistake can create large errors in the results and lead to this type of instability. The fact that the value of ψ at the trailing edge stably approaches a single value as the number of nodes increases speaks strongly against this explanation. Second, the lift calculations could be incorrect. While each coefficient of lift was calculated twice, a simple error in both of the calculations could have corrupted the results. This explanation seems to be the most likely. Last, enforcement of the Kutta condition very close to the trailing edge could lead to a glitch in the results. von Mises states that when enforcing Kutta's condition, unless the vortex density along the entire surface of the airfoil equals zero, then a singularity must exist in the profile for the condition to hold true. Since the BEM artificially holds the trailing edge velocity equal to zero, this explanation, although unlikely, could explain the large increase in pressure drop at the trailing edge of the airfoil.

Despite difficulties with the results, however, the robust nature and ability of the BEM is quite evident in the wide variety of results presented. Lift coefficients, flow velocities, and pressure distributions are all easily within the scope of boundary element modeling. With some adjustments, the modeling tool used in this project could quite effectively predict the lift coefficients for airfoils, and then be extended to include other computations, such as internal flows and external flows around objects other than an airfoil.

Works Cited

Codewarrior IDE. Vers. 3.0. Computer software. Metrowerks, 1998. Microsoft Windows 2000, 398 MB.

"Conformal Mapping." University of Genoa, Hydraulic Institute. Online. 20 April 2001. Available URL http://www.diam.unige.it/~irro/conformi_e.html

Grillo, C., G. Iannelli, and L. Tulumello. "An Alternative Boundary Element Method Approach to the 2D Potential Problem around Airfoils." European Journal of Mechanics: Vol. B, Fluids. 1990, Vol. 9, no. 6. 527-543.

Iannelli, G. Boundary Element Method System Solver. Vers. 1.0. Computer software. 2000. Microsoft Windows 2000 Professional, C, 118 KB.

Iannelli, G. Professor of Engineering Science and Mechanics, University of Tennessee Knoxville. Personal Communication. January to March 2001.

Iannelli, G., C. Grillo, and L. Tulumello. "A Kutta Condition Enforcing BEM Technology for Airfoil Aerodynamics." Unpublished results.

Munson, Bruce R., Donald Young, and Theodore Okiishi. Fundamentals of Fluid Mechanics. New York: Wiley, 1998.

von Mises, Richard. Theory of Flight. New York: Dover, 1959.

Appendix A: Code

In this appendix, all the code was written by Dr. Joe Iannelli, Department of Engineering Science and Mechanics, University of Tennessee-Knoxville, with the exception of those portions in bold and larger typeface.

main.c

```
        # include <stdio.h>
        # include <stdlib.h>
        # include <time.h>
        # include <math.h>


        # include      "ParamDef.h"
        # include      "Funct.h"

/* ---------------------------------------------------------------------------- */

        void main( void ) {
                int  i,j,n, np ;
                int bc[ N ] ;
                int extpr ;

                long double x[ N ], y[ N ] ;
                long double h[ N ][ 3 ], g[ N ][ 3 ] ;
                static long double a[ N ][ N ], b[ N ] ;
                long double ps[ N ], dpsdn[ N ][ 3 ]  ;
                int dnd, pdnd[ N ] ;
                long double psi[ 4 ] ;

                long double xi, yi ;

/* defines the number of nodes */
                n  = 64 ;
                np = n ;

                extpr = 0 ;

/* defines the coordinates of the boundary */
                bcoor( n, x, y ) ;

/* sets up the boundary conditions for each node */
                bdata( n, x, y, bc, ps, dpsdn ) ;

/* determines conditions necessary for solving the system */
                dngeo( n, bc, pdnd, &dnd, &np ) ;

/* defines matrices a and b with all 0 values */
                rsetm( np, a, b ) ;

/* (external pressure, number of elements, x-coor, y-coor, h?,g? */
/* bc, ps, dpsidn, b, a*/
/* within sys.c */
                supsy( extpr, n, x, y, h, g, bc, ps, dpsdn, b, a ) ;

/* executes the function only if bc>4 for any elements */

            if ( dnd > 0 ) {

                doneq( n, dnd, pdnd, x, y, h, g, bc, ps, dpsdn, b, a ) ;

            }
/* Gaussian elimination linear solver */
```

```
            solve( np, a, b );

/* returns the desired values of dpsdn and psi */
               disol( extpr, n, bc, b, ps, dpsdn );

/* Generates a matrix over which the pressure drop is then determined */

       for (i=-5; i<=5; i++){
       for (j=-5; j<=5; j++){

               xi=i/2.0;
               yi=j/10.0;

               /* evaluates the pressure values at these points */

               doval( n, xi, yi, x, y, h, g, ps, dpsdn, psi ) ;

               printf("\n %2.8f  %2.8f  %2.8f  %2.8f",xi,yi,psi[2],psi[3]);
               }}
               printf("\n\n");

               /* calculates the lift forces based on predetermined pressures */

               calculate( n, x, y, dpsdn, ps);

       return ;
       }

/* ------------------------------------------------------------------------ */
```

boundata.c

```
# include <stdio.h>
       # include <stdlib.h>
       # include <time.h>
       # include <math.h>


       # include      "ParamDef.h"
       # include      "Funct.h"

/* ---------------------------------------------------------------------------- */
       void bcoor( int n, long double x[], long double y[] ){

       /* calls the function that defines the airfoil based on the Joukowski profile */
       foildef(n, x, y );

       return ; }

/* ---------------------------------------------------------------------------- */

/* ---------------------------------------------------------------------------- */
       void bdata( int n, long double x[],  long double y[], int bc[],     \
                                   long double ps[], long double dpsdn[][ 3 ] ) {


              int i ;
              long double nx,ny,dx,dy;
       /* ------------------------------------------------------------ */

              for ( i = 1 ; i <= n ; i = i + 1 ){

                     bc[ i ] = 1 ;

                     if ( i > 1 ) {

                       bc[ i ] = 1 ; }

                     /* defines a function for psi (free stream) */
                     ps[ i ] = cos( angle ) * y[i] -  sin ( angle ) * x[i] ;

                     dpsdn[ i ][ 1 ] =  -1.0 ;

              }

              for ( i = 2 ; i < n ; i = i + 1 ){

              if (x[ i ] > x[ i+1 ]) {

                     if (x[ i] > x[i-1]){

               bc[ i ] = 2 ;

                     nx=0;
                     ny=0;
                     dx=x[i]-x[i-1];
                     dy=y[i]-y[i-1];
                     nx=-dy/sqrt(dx*dx +dy *dy);
                     ny=dx/sqrt(dx*dx +dy *dy);
```

```
            dpsdn[ i ][ 1 ] =  - (nx * cos(angle) - ny * sin(angle));

                    nx=0;
                    ny=0;
                    dx=x[i+1]-x[i];
                    dy=y[i+1]-y[i];
                    nx=-dy/sqrt(dx*dx +dy *dy);
                    ny=dx/sqrt(dx*dx +dy *dy);

            dpsdn[ i ][ 2 ] =  - (nx * cos(angle) - ny * sin(angle)) ;
            }
            }
            }
        /* ------------------------------------------------------------ */

            printf( "Boundary conditions.\n");
            printf( "x          y           ps          bc\n");


            for ( i = 1 ; i <= n ; i = i + 1 ){

                    printf( "%2.5f    %2.5f   %.8Lf     %.d\n", x[ i], y[ i ], ps[ i ], bc[ i
] ) ;

            }

            printf( "\n\n" );

            return ; }

/* ---------------------------------------------------------------------------- */
/* ---------------------------------------------------------------------------- */

        void dngeo( int n, int bc[], int pdnd[], int *dnd, int *np ) {

            int i ;

        /* ------------------------------------------------------------ */

            *dnd = 0 ;

            for ( i = 1 ; i <= n ; i = i + 1 ) {

                    if ( bc[ i ] > 4 ) {

                            *dnd = *dnd + 1 ;

                            pdnd[ *dnd ] = i ;

                    }
            }

            *np = n +   *dnd ;

        /* ------------------------------------------------------------ */
```

```
          return ; }
/* ------------------------------------------------------------------------ */
```

airfoil.c

```
        # include <stdio.h>
        # include <stdlib.h>
        # include <time.h>
        # include <math.h>


        # include      "ParamDef.h"
        # include      "Funct.h"
        # define pi 3.14159265359


void foildef(int n, long double x[],long double y[]) {

int i;
long double xc,yc,r,a,theta;
long double x1[N],y1[N];

/* sets up the Joukowski profile parameters */

xc=0.1;
yc=0.0;
r=1;
a=0;

/* determines necessary values to use the Joukowski profile */

if (yc==0){
        if (xc==0){
                a = r;}
        else {a=r-xc;}
                }

if (yc !=0){
        if (xc==0){
                a=sqrt(r*r-yc*yc);}
        else {
                a=-xc+sqrt(r*r-yc*yc);}
                }

/* Joukowski transform (without rotation) */

for (i=1 ; i<=n; i++){
        theta=2*pi/n*i;
        x[i]=r*cos(theta)+xc;
```

```
          y[i]=r*sin(theta)+yc;
          x1[i]=x[i]+a*a*x[i] / (x[i]*x[i]+y[i]*y[i]);
          y1[i]=y[i]-a*a*y[i] / (x[i]*x[i]+y[i]*y[i]);
          x[i]=- x1[i];
          y[i]=y1[i];

                    }

return;
}

/*-------------------------------------------------------*/

void calculate(int n,long double x[], long double y[], \
          long double dpsdn[][ 3 ], long double ps[]){

          int i,j;
          long double
pressure[N],chord,psiconstant,cl,cl2,cp[N],force,rhoair,vpert,vinf,vtot[N],dx,dy,dx2,dy2,nx,ny;

          /* sets the density of air for use in pressure determinations */

          rhoair=1.23;
          force=0;
          cl=0;

          /* calculates the chord length of the airfoil */

          chord=x[n/2]-x[n];

          /* determines the contribution of the free stream velocity */

          for ( i=1; i<=n; i++){

                    vpert=dpsdn[i][1];
                    dx=0;
                    dy=0;
                    dx2=0;
                    dy2=0;

                              if (i==1) {
                              nx=0;
                              ny=0;
                              dx=x[i+1]-x[i];
                              dy=y[i+1]-y[i];
                              nx=-dy/sqrt(dx*dx +dy *dy);
```

```
                ny=dx/sqrt(dx*dx +dy *dy);
                dx2=x[i]-x[n];
                dy2=y[i]-y[n];
                nx=(nx-dy2/sqrt(dx2*dx2 +dy2 *dy2))/2;
                ny=(ny+dx2/sqrt(dx2*dx2 +dy2 *dy2))/2;
                }

                else if(i==n) {
                nx=0;
                ny=0;
                dx=x[1]-x[i];
                dy=y[1]-y[i];
                nx=-dy/sqrt(dx*dx +dy *dy);
                ny=dx/sqrt(dx*dx +dy *dy);
                dx2=x[i]-x[i-1];
                dy2=y[i]-y[i-1];
                nx=(nx-dy2/sqrt(dx2*dx2 +dy2 *dy2))/2;
                ny=(ny+dx2/sqrt(dx2*dx2 +dy2 *dy2))/2;
                }

                else  {
                nx=0;
                ny=0;
                dx=x[i+1]-x[i];
                dy=y[i+1]-y[i];
                nx=-dy/sqrt(dx*dx +dy *dy);
                ny=dx/sqrt(dx*dx +dy *dy);
                dx2=x[i]-x[i-1];
                dy2=y[i]-y[i-1];
                nx=(nx-dy2/sqrt(dx2*dx2 +dy2 *dy2))/2;
                ny=(ny+dx2/sqrt(dx2*dx2 +dy2 *dy2))/2;
                }

        vinf= cos (angle) * ny - sin (angle) * nx;

/* adds the perturbation and free stream velocities */

        vtot[i]=vinf + vpert;

/* sets the velocity equal to zero at the trailing edge */

        if (i == n/2){
        vtot[i]=0.0;
        psiconstant=ps[i];
        }
```

```
/* uses two different methods to determine the lift coefficient */

        pressure[i]=-.5*rhoair*vtot[i]*vtot[i];
        cp[i]=1-vtot[i]*vtot[i];
        }

for (i=1; i<=(n/2-2); i++){

        j=i-1;
        if (i==1){
        j=n;}

        cl=cl+(cp[j]+cp[i])*0.5*(x[i]-x[j]);

        force=force-(pressure[j]+pressure[i])*0.5*(x[i]-x[j]);

        printf("%1.8f  %1.8f  %1.8f  %1.8f\n",x[i],y[i],pressure[i],force);

        }

for (i=n/2+2; i<=n; i++){

        j=i-1;
        if (i==1){
        j=n;}

        cl=cl+(cp[j]+cp[i])*0.5*(x[i]-x[j]);

        force=force-(pressure[j]+pressure[i])*0.5*(x[i]-x[j]);

        printf("%1.8f  %1.8f  %1.8f  %1.8f\n",x[i],y[i],pressure[i],force);

        }

        cl=cl/chord;
        cl2=force/(0.5*rhoair*chord);

        printf("\n The coefficient of lift is %1.8f ",cl);
        printf("\n The recalculated value is %1.8f ",cl2);
        printf("\n The value of psi at the trailing edge is %2.8f ",psiconstant);
        printf("\n The number of nodes is %4.0d ",n);
}
```

sys.c

```
        # include <stdio.h>
        # include <stdlib.h>
        # include <time.h>
        # include <math.h>


        # include       "ParamDef.h"
        # include       "Funct.h"



/* ------------------------------------------------------------------------- */

        void rsetm( int n, long double a[][ N ], long double b[] ) {

                int i, j ;

                for ( i = 1 ; i <= n ; i = i + 1 ) {

                        for ( j = 1 ; j <= n ; j = j + 1 ) {

                                a[ i ][ j ] = 0.0 ;

                        }

                                b[ i ] = 0.0 ;

                }


        return ; }
/* ------------------------------------------------------------------------- */


/* ------------------------------------------------------------------------- */
        void supsy( int extpr, int n, long double x[], long double y[], long double h[][3],\
                                long double g[][3], int bc[], long double ps[], long double
dpsdn[][3],\
                                long double b[], long double a[][ N ] ){

                long double xg[ 6 ], wg[ 6 ]   ;

                int i, j1, j2, j, i1, i2 ;
                int idnd ;
                long double xi, yi ;
                long double x1, y1, x2, y2 ;
                long double dx, dy, dl ;
                long double h1, h2, g1, g2 ;
                long double ci ;

        /* ------------------------------------------------------------ */

                sgdta( wg, xg ) ;

        /* ------------------------------------------------------------ */

                for ( i = 1 ; i <= n ; i = i + 1 ) {
                        idnd = 0 ;

/* defines the point xi and yi from which you want to integrate */
                        xi = x[ i ] ;
                        yi = y[ i ] ;

                        j2 = i ;
                        j1 = i - 1 ;
                        if ( j1 < 1 ) {
                                j1 = n ;
                        }
        /* ------------------------------------------------------------ */

                for ( j = 1 ; j <= n ; j = j + 1 ) {

                        i1 = j ;
                        i2 = j + 1 ;
                        if              ( i2 > n ) {
                                i2 = 1 ;
                        }

                        x1 = x[ i1 ] ;
                        y1 = y[ i1 ] ;
                        x2 = x[ i2 ] ;
                        y2 = y[ i2 ] ;
                        dx = x2 - x1 ;
                        dy = y2 - y1 ;
                        dl = sqrt( dx * dx + dy * dy ) ;

                        intgr( xg, wg, j, j1, j2, xi, yi, x1, y1, x2, y2, \
                                   dx, dy, dl, &h1, &h2, &g1, &g2) ;

                        h[ j ][ 1 ] = h1 ;
                        h[ j ][ 2 ] = h2 ;
                        g[ j ][ 1 ] = g1 ;
                        g[ j ][ 2 ] = g2 ;



                }

        /* ------------------------------------------------------------ */

        /* ------------------------------------------------------------ */
                ci = 0.0 ;

                for ( j = 1 ; j <= n ; j = j + 1 ) {

                        ci = ci + h[ j ][ 1 ] + h[ j ][ 2 ] ;

                }

                if ( extpr > 0 ) {
                        ci = 1.0 + ci ;
                }

        /* ------------------------------------------------------------ */

        /* ------------------------------------------------------------ */

        if ( i == 1 ) {
                if              ( bc[ i ] == 1 )  {

                        b[ i ] = b[ i ] - ( ci * ps[ 1 ] + ci * ps[ 2 ] ) / 2.0  ; }

                else if ( bc[ i ] == 2 ) {

                        a[ i ][ 1 ] = a[ i ][ 1 ] + ci / 2.0 ;

                        a[ i ][ 2 ] = a[ i ][ 2 ] + ci / 2.0 ;}
                else {

                b[ i ] = b[ i ] - ( ci * ps[ 1 ] + ci * ps[ 2 ] ) / 2.0 ;

                 } }

else {

                if               ( bc[ i ] == 1 )  {
                        b[ i ] = b[ i ] - ci * ps[ i ] ; }
                else if ( bc[ i ] == 2 ) {
                        a[ i ][ i ] = a[ i ][ i ] + ci ;  }
                else {
                b[ i ] = b[ i ] - ci * ps[ i ] ;
                 }

}
                for ( j = 1 ; j <= n ; j = j + 1 ) {
                        i1 = j ;
                        if ( bc[ i1 ] == 1 ) {
                                a[ i ][ i1 ] = a[ i ][ i1 ] + g[ j ][ 1 ] ;
                                b[ i ] = b[ i ] + ps[ i1 ] * h[ j ][ 1 ] ; }
                        else if ( bc[ i1 ] == 2 ) {
                                a[ i ][ i1 ] = a[ i ][ i1 ] - h[ j ][ 1 ] ;
                                b[ i ] = b[ i ] - dpsdn[ i1 ][ 2 ] * g[ j ][ 1 ] ; }
                        else if ( bc[ i1 ] == 3 ) {
                                b[ i ] = b[ i ] - dpsdn[ i1 ][ 2 ] * g[ j ][ 1 ] ;
                                b[ i ] = b[ i ] + ps[ i1 ] * h[ j ][ 1 ] ; }
                        else if ( bc[ i1 ] == 4 ) {
                                a[ i ][ i1 ] = a[ i ][ i1 ] + g[ j ][ 1 ] ;
                                b[ i ] = b[ i ] + ps[ i1 ] * h[ j ][ 1 ] ;  }
                        else if ( bc[ i1 ] == 5 ) {
                                idnd = idnd + 1 ;
                                a[ i ][ n + idnd ] = a[ i ][ n + idnd ] + g[ j ][ 1 ] ;
                    `           b[ i ] = b[ i ] + ps[ i1 ] * h[ j ][ 1 ] ;
                        }

                        i2 = j + 1 ;
                        if ( i2 > n ) {
                                i2 = 1 ;
                        }
                        if ( bc[ i2 ] == 1 ) {
                                a[ i ][ i2 ] = a[ i ][ i2 ] + g[ j ][ 2 ] ;
                                b[ i ] = b[ i ] + ps[ i2 ] * h[ j ][ 2 ] ; }
                        else if ( bc[ i2 ] == 2 ) {
                                a[ i ][ i2 ] = a[ i ][ i2 ] - h[ j ][ 2 ] ;
                                b[ i ] = b[ i ] - dpsdn[ i2 ][ 1 ] * g[ j ][ 2 ] ; }
                        else if ( bc[ i2 ] == 3 ) {
                                a[ i ][ i2 ] = a[ i ][ i2 ] + g[ j ][ 2 ] ;
                                b[ i ] = b[ i ] + ps[ i2 ] * h[ j ][ 2 ] ; }
                        else if ( bc[ i2 ] == 4 ) {

                                b[ i ] = b[ i ] - dpsdn[ i2 ][ 1 ] * g[ j ][ 2 ] ;
                                b[ i ] = b[ i ] + ps[ i2 ] * h[ j ][ 2 ] ; }

                        else if ( bc[ i2 ] == 5 ) {

                                a[ i ][ i2 ] = a[ i ][ i2 ] + g[ j ][ 2 ] ;

                                b[ i ] = b[ i ] + ps[ i2 ] * h[ j ][ 2 ] ;

                        }


                    }

        /* -------------------------------------------------------------- */

                }


                if ( extpr == 2 ) {

                        for ( i = 1 ; i <= n ; i = i + 1 ) {

                                b[ i ] = b[ i ] + a[ i ][ 1 ] ;
                                a[ i ][ 1 ] = 1.0 ;

                        }

                }

        /* -------------------------------------------------------------- */


        return ; }
/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */
        void doneq( int n, int dnd, int pdnd[], long double x[], long double y[], \
                        long double h[][3], long double g[][3], int bc[], long double ps[],
\
                        long double dpsdn[][3], long double b[], long double a[][ N ] ){

                long double xg[ 6 ], wg[ 6 ]  ;

                int i, j1, j2, j, i1, i2 ;
                int ii, iint, idnd ;
                long double xi, yi ;
                long double x1, y1, x2, y2 ;
                long double dx, dy, dl ;
                long double h1, h2, g1, g2 ;

        /* -------------------------------------------------------------- */
                sgdta( wg, xg ) ;

                iint = 1 ;
        /* -------------------------------------------------------------- */

        /* -------------------------------------------------------------- */


                for ( i = 1 ; i <= dnd ; i = i + 1 ) {

                        idnd = 0 ;

                        ii = pdnd[ i ] ;

                        xi = x[ ii ] ;
                    yi = y[ ii ] ;

                    j1 = ii - 1 ;
                    if ( j1 < 1 ) {
                            j1 = n ;
                    }

                            i1 = j1 ;
                            i2 = j1 + 1 ;
                            if              ( i2 > n ) {
                                    i2 = 1 ;
                            }

                            x1 = x[ i1 ] ;
                            y1 = y[ i1 ] ;

                            x2 = x[ i2 ] ;
                            y2 = y[ i2 ] ;

                            dx = x2 - x1 ;
                            dy = y2 - y1 ;
                            dl = sqrt( dx * dx + dy * dy ) ;

                            xi = xi + 0.5 * ( y2 - y1 ) / ( 2.0 * dl ) ;
                            yi = yi + 0.5 * ( x1 - x2 ) / ( 2.0 * dl ) ;

                     j2 = ii ;

                            i1 = j2 ;
                            i2 = j2 + 1 ;
                            if              ( i2 > n ) {
                                    i2 = 1 ;
                            }

                            x1 = x[ i1 ] ;
                            y1 = y[ i1 ] ;

                            x2 = x[ i2 ] ;
                            y2 = y[ i2 ] ;

                            dx = x2 - x1 ;
                            dy = y2 - y1 ;
                            dl = sqrt( dx * dx + dy * dy ) ;

                            xi = xi + 0.5 * ( y2 - y1 ) / ( 2.0 * dl ) ;
                            yi = yi + 0.5 * ( x1 - x2 ) / ( 2.0 * dl ) ;

            /* ------------------------------------------------------------ */
                    for ( j = 1 ; j <= n ; j = j + 1 ) {


                            i1 = j ;
                            i2 = j + 1 ;
                            if              ( i2 > n ) {
                                    i2 = 1 ;
                            }

                            x1 = x[ i1 ] ;
                            y1 = y[ i1 ] ;

                            x2 = x[ i2 ] ;
                            y2 = y[ i2 ] ;

                            dx = x2 - x1 ;
                            dy = y2 - y1 ;
                            dl = sqrt( dx * dx + dy * dy ) ;

                            bouin( xg, wg, iint, xi, yi, x1, y1, x2, y2, dx, dy, dl, &h1, &h2,
        &g1, &g2 ) ;

                            h[ j ][ 1 ] = h1 ;
```

```
                        h[ j ][ 2 ] = h2 ;
                        g[ j ][ 1 ] = g1 ;
                        g[ j ][ 2 ] = g2 ;


                }

        /* ---------------------------------------------------------- */

        /* ---------------------------------------------------------- */

                for ( j = 1 ; j <= n ; j = j + 1 ) {

                        i1 = j ;

                        if ( bc[ i1 ] == 1 ) {

                                a[ n + i ][ i1 ] = a[ n + i ][ i1 ] + g[ j ][ 1 ] ;

                                b[ n + i ] = b[ n + i ] + ps[ i1 ] * h[ j ][ 1 ] ; }

                        else if ( bc[ i1 ] == 2 ) {

                                a[ n + i ][ i1 ] = a[ n + i ][ i1 ] - h[ j ][ 1 ] ;

                                b[ n + i ] = b[ n + i ] - dpsdn[ i1 ][ 2 ] * g[ j ][ 1 ] ;
}

                        else if ( bc[ i1 ] == 3 ) {

                                b[ n + i ] = b[ n + i ] - dpsdn[ i1 ][ 2 ] * g[ j ][ 1 ] ;
                                b[ n + i ] = b[ n + i ] + ps[ i1 ] * h[ j ][ 1 ] ; }

                        else if ( bc[ i1 ] == 4 ) {

                                a[ n + i ][ i1 ] = a[ n + i ][ i1 ] + g[ j ][ 1 ] ;
                                b[ n + i ] = b[ n + i ] + ps[ i1 ] * h[ j ][ 1 ] ;  }

                        else if ( bc[ i1 ] == 5 ) {
                                idnd = idnd + 1 ;
                                a[ n + i ][ n + idnd ] = a[ n + i ][ n + idnd ] + g[ j ][ 1
] ;

                                b[ n + i ] = b[ n + i ] + ps[ i1 ] * h[ j ][ 1 ] ;

                        }

                        i2 = j + 1 ;
                        if ( i2 > n ) {
                                i2 = 1 ;
                        }

                        if ( bc[ i2 ] == 1 ) {

                                a[ n + i ][ i2 ] = a[ n + i ][ i2 ] + g[ j ][ 2 ] ;

                                b[ n + i ] = b[ n + i ] + ps[ i2 ] * h[ j ][ 2 ] ; }

                        else if ( bc[ i2 ] == 2 ) {

                                a[ n + i ][ i2 ] = a[ n + i ][ i2 ] - h[ j ][ 2 ] ;

                                b[ n + i ] = b[ n + i ] - dpsdn[ i2 ][ 1 ] * g[ j ][ 2 ] ;
}

                        else if ( bc[ i2 ] == 3 ) {

                                a[ n + i ][ i2 ] = a[ n + i ][ i2 ] + g[ j ][ 2 ] ;
                                b[ n + i ] = b[ n + i ] + ps[ i2 ] * h[ j ][ 2 ] ; }
                        else if ( bc[ i2 ] == 4 ) {

                                b[ n + i ] = b[ n + i ] - dpsdn[ i2 ][ 1 ] * g[ j ][ 2 ] ;
                                b[ n + i ] = b[ n + i ] + ps[ i2 ] * h[ j ][ 2 ] ; }

                        else if ( bc[ i2 ] == 5 ) {

                                a[ n + i ][ i2 ] = a[ n + i ][ i2 ] + g[ j ][ 2 ] ;

                                b[ n + i ] = b[ n + i ] + ps[ i2 ] * h[ j ][ 2 ] ;

                        }


                }

        /* ------------------------------------------------------------ */

                }


        /* ------------------------------------------------------------ */


        return ; }
/* ---------------------------------------------------------------------------- */


/* ---------------------------------------------------------------------------- */
        void disol( int extpr, int n, int bc[], long double b[], long double ps[], long double
dpsdn[][3] ){


                int i, idnd ;
                long double psito, psite ;
        /* ------------------------------------------------------------ */

                idnd = 0 ;

                printf( "The value of extpr is %d.",extpr);

                for ( i = 1 ; i <= n ; i = i + 1 ) {

                        if              ( bc[ i ] == 1 ) {

                                dpsdn[ i ][ 1 ] = b[ i ] ;
                                dpsdn[ i ][ 2 ] = b[ i ] ; }

                        else if ( bc[ i ] == 2 ) {
                                        ps[ i ] = b[ i ] ; }

                        else if ( bc[ i ] == 3 ) {

                                dpsdn[ i ][ 1 ] = b[ i ] ; }

                        else if ( bc[ i ] == 4 ) {

                                dpsdn[ i ][ 2 ] = b[ i ] ; }


                        else if ( bc[ i ] == 5 ) {

                                idnd = idnd + 1 ;
                                dpsdn[ i ][ 1 ] = b[ i ] ;
                                dpsdn[ i ][ 2 ] = b[ n + idnd ] ; }
                }
                printf( "\n" );

                if ( extpr == 2 ) {

                        dpsdn[ 1 ][ 1 ] = -1.0 ;

                        psito = b[ 1 ] ;

                        for ( i = 1 ; i <= n ; i = i + 1 ) {

                                ps[ i ] = psito + ps[ i ] ;

                        }

                        psite = 2.0 * 1.0 * log( 1.0 );
                        printf( "\n" ) ;

                        printf( "%.8Lf    %.8Lf\n", psito, psite  ) ;

                        printf( "\n" ) ;
                        printf( "\n" ) ;

                        for ( i = 1 ; i <= n ; i = i + 1 ){

                                psite = - 2.0 + ( ps[ i ] - psito ) / 1.0 ;

                                printf( "%.8Lf  %.8Lf  %.8Lf\n",  ps[ i ],  dpsdn[ i ][ 1 ],
psite ) ;

                        } }
                else {
                        printf( "\n" ) ;
                        printf( "\n" ) ;
                        printf( "ps          dpsdn          dpsdn2\n");
                        for ( i = 1 ; i <= n ; i = i + 1 ){

                                printf( "%.8Lf  %.8Lf  %.8Lf\n",  ps[ i ],  dpsdn[ i ][ 1 ],
dpsdn[ i ][ 2 ] ) ;

                        }
                }



                printf( "\n" ) ;
                printf( "\n" ) ;

        /* ------------------------------------------------------------ */


        return ; }
/* ---------------------------------------------------------------------------- */


/* ---------------------------------------------------------------------------- */
        void doval( int n, long double xi, long double yi, long double x[], long double y[], \
        long double h[][3], long double g[][3], long double ps[], long double dpsdn[][3], long
double psi[] ){


                long double xg[ 6 ], wg[ 6 ]  ;


                int ii, j, i1, i2 ;
                long double x1, y1, x2, y2 ;
                long double dx, dy, dl ;
                long double h1, h2, g1, g2 ;
                sgdta( wg, xg ) ;

                for ( ii = 1 ; ii <= 3 ; ii = ii + 1 ) {

        /* --------------------------------------------------------------- */


                for ( j = 1 ; j <= n ; j = j + 1 ) {


                        i1 = j ;
                        i2 = j + 1 ;
                        if              ( i2 > n ) {
                                i2 = 1 ;
                        }

                        x1 = x[ i1 ] ;
                        y1 = y[ i1 ] ;

                        x2 = x[ i2 ] ;
                        y2 = y[ i2 ] ;

                        dx = x2 - x1 ;
                        dy = y2 - y1 ;
                        dl = sqrt( dx * dx + dy * dy ) ;

                        bouin( xg, wg, ii, xi, yi, x1, y1, x2, y2, dx, dy, dl, &h1, &h2, &g1, &g2)
;

                        h[ j ][ 1 ] = h1 ;
                        h[ j ][ 2 ] = h2 ;
                        g[ j ][ 1 ] = g1 ;
                        g[ j ][ 2 ] = g2 ;



                }

                psi[ ii ] = 0.0 ;

                for ( j = 1 ; j <= n ; j = j + 1 ) {

                        i1 = j ;
                        i2 = j + 1 ;
                        if              ( i2 > n ) {
                                i2 = 1 ;
                        }

                        psi[ ii ] = psi[ ii ] +    ps[ i1 ]      * h[ j ][ 1 ] +    ps[ i2 ]
* h[ j ][ 2 ] ;
                        psi[ ii ] = psi[ ii ] - dpsdn[ i1 ][ 2 ] * g[ j ][ 1 ] - dpsdn[ i2 ][ 1 ]
* g[ j ][ 2 ] ;

                }

        /* --------------------------------------------------------------- */
                }

        return ; }
/* ----------------------------------------------------------------------- */
```

gaussint.c

```
        # include <stdio.h>
        # include <stdlib.h>
        # include <time.h>
        # include <math.h>


        # include       "ParamDef.h"
        # include       "Funct.h"

/* --------------------------------------------------------------------------- */
        void sgdta(long double wg[], long double xg[ ] ) {


/* sets the weight values for Gaussian integration */

                xg[ 1 ] = - 0.906179845938664 ;
                xg[ 2 ] = - 0.538469310105683 ;
                xg[ 3 ] = - 0.0 ;
                xg[ 4 ] =   0.538469310105683 ;
                xg[ 5 ] =   0.906179845938664 ;

                wg[ 1 ] =   0.236926885056189 ;
                wg[ 2 ] =   0.478628670499366 ;
                wg[ 3 ] =   0.568888888888889 ;
                wg[ 4 ] =   0.478628670499366 ;
                wg[ 5 ] =   0.236926885056189 ;




        return ; }

/* --------------------------------------------------------------------------- */

/* --------------------------------------------------------------------------- */
        void gauin( long double xg[], long double wg[], int ii, int ifu,                  \
                              long double xi,   long double yi, long double x1, long double y1, \
                              long double x2,   long double y2, long double dx, long double dy, \
                              long double dl,   long double *gi ) {

/* performs the Gaussian integration */

                long double xn, f, f1, f2, f3, f4, f5 ;

                        xn = xg[ 1 ] ;

                        funct( ii, ifu, xi, yi, x1, y1, x2, y2, dx, dy, dl, xn, &f ) ;

                        f1 = f ;

                        xn = xg[ 2 ] ;

                        funct( ii, ifu, xi, yi, x1, y1, x2, y2, dx, dy, dl, xn, &f ) ;

                        f2 = f ;


                        xn = xg[ 3 ] ;

                        funct( ii, ifu, xi, yi, x1, y1, x2, y2, dx, dy, dl, xn, &f ) ;

                        f3 = f ;


                        xn = xg[ 4 ] ;

                        funct( ii, ifu, xi, yi, x1, y1, x2, y2, dx, dy, dl, xn, &f ) ;
                        f4 = f ;

                        xn = xg[ 5 ] ;

                        funct( ii, ifu, xi, yi, x1, y1, x2, y2, dx, dy, dl, xn, &f ) ;

                        f5 = f ;

                        *gi = wg[ 1 ] * f1 + wg[ 2 ] * f2 + wg[ 3 ] * f3  + wg[ 2 ] * f4 + wg[ 5 ]
* f5 ;

                return ; }

/* ---------------------------------------------------------------------- */
```

integrals.c

```
        # include <stdio.h>
        # include <stdlib.h>
        # include <time.h>
        # include <math.h>


        # include       "ParamDef.h"
        # include       "Funct.h"

/* ----------------------------------------------------------------------------- */
        void intgr( long double xg[], long double wg[], int j, int j1, int j2,
        \
                               long double xi,   long double yi,  long double x1,  long double y1,
\
                               long double x2,   long double y2,  long double dx,  long double dy,
\
                               long double dl,   long double *h1, long double *h2, long double
*g1,\
                               long double *g2) {


                int ii ;

                        if ( j == j1 ) {

                               *h1 = 0.0 ;
                               *h2 = 0.0 ;
                               *g1 = dl * ( log( dl ) - 0.5 ) / 12.5663706144 ;
                               *g2 = dl * ( log( dl ) - 1.5 ) / 12.5663706144 ; }

                        else if ( j == j2 ) {

                               *h1 = 0.0 ;
                               *h2 = 0.0 ;
                               *g1 = dl * ( log( dl ) - 1.5 ) / 12.5663706144 ;
                               *g2 = dl * ( log( dl ) - 0.5 ) / 12.5663706144 ; }


                        else {

                                ii = 1 ;
                                bouin( xg, wg, ii, xi, yi, x1, y1, x2, y2, dx, dy, dl, h1, h2, g1,
g2 ) ;

                          }

                        if ( j2 == 1 ) {
                                 if ( j == j2 ) {

                               *h1 = 0.0 ;
                               *h2 = 0.0 ;
                               *g1 = dl * ( log( dl / 2.0 ) - 2.0 ) / 12.5663706144 ;
                               *g2 = dl * ( log( dl / 2.0 ) - 2.0 ) / 12.5663706144 ; }

                        else {
                                       ii = 1 ;
                                bouin( xg, wg, ii, xi, yi, x1, y1, x2, y2, dx, dy, dl, h1, h2, g1,
g2 ) ;  }

                          }

        return ; }

/* ----------------------------------------------------------------------------- */

/* ----------------------------------------------------------------------------- */
```

```
        void bouin( long double xg[], long double wg[], int ii,
                        \
                                long double xi,   long double yi,  long double x1,  long double y1,
\
                                long double x2,   long double y2,  long double dx,  long double dy,
\
                                long double dl,   long double *h1, long double *h2, long double
*g1,\
                                long double *g2) {

                int ifu ;
                long double gi ;

                        ifu = 1;
                        gauin( xg, wg, ii, ifu, xi, yi, x1, y1, x2, y2, dx, dy, dl, &gi );
                        *h1 = gi * dl / 2.0 ;

                        ifu = 2;
                        gauin( xg, wg, ii, ifu, xi, yi, x1, y1, x2, y2, dx, dy, dl, &gi );
                        *h2 = gi * dl / 2.0 ;

                        ifu = 3;
                        gauin( xg, wg, ii, ifu, xi, yi, x1, y1, x2, y2, dx, dy, dl, &gi );
                        *g1 = gi * dl / 2.0 ;

                        ifu = 4;
                        gauin( xg, wg, ii, ifu, xi, yi, x1, y1, x2, y2, dx, dy, dl, &gi );
                        *g2 = gi * dl / 2.0 ;


        return ; }
/* ---------------------------------------------------------------------------- */

/* ---------------------------------------------------------------------------- */

        void funct( int ii, int i,  long double xi, long double yi, long double x1,       \
                                long double y1, long double x2, long double y2, long double dx, \
                                long double dy, long double dl, long double s,  long double *f ) {


                long double x, y, r2, fi, dfidn, dfidx, dfidy, dfndx, dfndy ;

                if ( ii == 1 ) {

                        if ( i == 1 ) {

                                x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                                y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                          r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                          dfidn = ( ( x - xi ) * dy - ( y - yi ) * dx ) / ( 6.28318530718 * dl *
r2 ) ;

                          *f = ( 1 - s ) * dfidn / 2.0 ;  }

                        else if ( i == 2 ) {

                                x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                                y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                          r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                          dfidn = ( ( x - xi ) * dy - ( y - yi ) * dx ) / ( 6.28318530718 * dl *
r2 ) ;

                          *f = ( 1 + s ) * dfidn / 2.0 ;  }

                        else if ( i == 3 ) {

                                x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
```

```
                         y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                      r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                      fi = log( r2 ) / 12.5663706144 ;

                      *f = ( 1 - s ) * fi / 2.0 ;  }

                   else if ( i == 4 ) {

                         x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                         y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                      r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                      fi = log( r2 ) / 12.5663706144 ;

                      *f = ( 1 + s ) * fi / 2.0 ;  } }

              else if ( ii == 2 ) {

                   if ( i == 1 ) {

                         x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                         y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                      r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                      dfndx = ( 2.0 * ( x - xi ) * ( ( x - xi ) * dy - ( y - yi ) * dx ) / r2
  - dy ) / ( 6.28318530718 * dl * r2 ) ;

                      *f = ( 1 - s ) * dfndx / 2.0 ;  }

                   else if ( i == 2 ) {

                         x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                         y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                      r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                      dfndx = ( 2.0 * ( x - xi ) * ( ( x - xi ) * dy - ( y - yi ) * dx ) / r2
  - dy ) / ( 6.28318530718 * dl * r2 ) ;

                      *f = ( 1 + s ) * dfndx / 2.0 ;  }

                   else if ( i == 3 ) {

                         x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                         y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                      r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                      dfidx = -2.0 * ( x - xi ) / ( 12.5663706144 * r2 ) ;

                      *f = ( 1 - s ) * dfidx / 2.0 ;  }

                   else if ( i == 4 ) {

                         x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                         y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                      r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                      dfidx = -2.0 * ( x - xi ) / ( 12.5663706144 * r2 ) ;

                      *f = ( 1 + s ) * dfidx / 2.0 ;    } }

              else if ( ii == 3 ) {
                    if ( i == 1 ) {

                           x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                           y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                       r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                       dfndy = ( 2.0 * ( y - yi ) * ( ( x - xi ) * dy - ( y - yi ) * dx ) / r2
+ dx ) / ( 6.28318530718 * dl * r2 ) ;

                       *f = ( 1 - s ) * dfndy / 2.0 ;  }

                    else if ( i == 2 ) {

                           x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                           y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                       r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                       dfndy = ( 2.0 * ( y - yi ) * ( ( x - xi ) * dy - ( y - yi ) * dx ) / r2
+ dx ) / ( 6.28318530718 * dl * r2 ) ;

                       *f = ( 1 + s ) * dfndy / 2.0 ;  }

                    else if ( i == 3 ) {

                           x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                           y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                       r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                       dfidy = -2.0 * ( y - yi ) / ( 12.5663706144 * r2 ) ;

                       *f = ( 1 - s ) * dfidy / 2.0 ;  }

                    else if ( i == 4 ) {

                           x = ( ( 1.0 - s ) * x1 + ( 1.0 + s ) * x2 ) / 2.0 ;
                           y = ( ( 1.0 - s ) * y1 + ( 1.0 + s ) * y2 ) / 2.0 ;

                       r2 = ( x - xi ) * ( x - xi ) + ( y - yi ) * ( y - yi ) ;

                       dfidy = -2.0 * ( y - yi ) / ( 12.5663706144 * r2 ) ;

                       *f = ( 1 + s ) * dfidy / 2.0 ;    }


                     }

       return ; }
/* --------------------------------------------------------------------------- */
```

linsolv.c

```
        # include <stdio.h>
        # include <stdlib.h>
        # include <time.h>
        # include <math.h>

        # include       "ParamDef.h"
        # include       "Funct.h"

/* ------------------------------------------------------------------------ */
        void solve( int n, long double a[][ N ], long double b[] ){

                int i, j, k, nm, ip;

                double pi, c, eps;

                nm  = n - 1 ;
                eps = 0.00000001;

                /* Gaussian solver */

                /* ------------------------------------------------------------ */

                for ( i = 1 ; i <= nm ; i = i + 1 ) {

                        /* -------------------------------------------------------  */
                                pi = fabs( a[ i ][ i ] );
                                ip = 0 ;

                                for ( k = i + 1 ; k <= n ; k = k + 1 ) {

                                        c = fabs( a[ k ][ i ] ) ;

                                        if ( c > pi ) {
                                                ip = k ;
                                        }
                                }

                                if ( ip > 0 ) {

                                        for ( k = i ; k <= n ; k = k + 1 ) {

                                                c = a[ i ][ k ] ;
                                                a[ i ][ k ] = a[ ip ][ k ] ;
                                                a[ ip ][ k ] = c ;

                                        }

                                        c = b[ i ] ;
                                        b[ i ] = b[ ip ];
                                        b[ ip ] = c ; }

                                else if ( pi < eps ) {

                                        printf( "singular matrix\n" );

                                }

                        /* -------------------------------------------------------  */


                        pi = 1.0 / a[ i ][ i ] ;

                        for ( k = i + 1 ; k <= n ; k = k + 1 ) {

                                c = a[ k ][ i ] * pi ;
                                        a[ k ][ i ] = 0.0 ;

                                for ( j = i + 1 ; j <= n ; j = j + 1 ) {
```

```
                                a[ k ][ j ] = a[ k ][ j ] - a[ i ][ j ] * c;

                        }

                                b[ k ] = b[ k ] - b[ i ] * c ;

                }

        }

        /* ------------------------------------------------------------ */

        /* ------------------------------------------------------------ */

                        b[ n ] = b[ n ] / a[ n ][ n ] ;

        for ( i = n - 1 ; i >= 1 ; i = i - 1 ) {

                c = 0.0 ;

                for ( j = i + 1 ; j <= n ; j = j + 1 ) {

                        c = c + a[ i ][ j ] * b[ j ] ;

                }

                        b[ i ] = ( b[ i ] - c ) / a[ i ][ i ] ;

        }

        /* ------------------------------------------------------------ */

return ; }
```

Paramdef.h

```
/* appears to define the number of elements in the arrays x and y */

        # define                N       100
        # define angle 0.05
```

Funct.h

```
        void bcoor( int n, long double x[], long double y[] ) ;

        void bdata( int n, long double x[], long double y[], int bc[], long double ps[], long
double dpsdn[][3] ) ;

        void foildef(int n, long double x[],long double y[]);

        void calculate(int n,long double x[], long double y[], long double dpsdn[][ 3 ], long
double ps[]);

        void dngeo( int n, int bc[], int pdnd[], int *dnd, int *np ) ;

        void rsetm( int n, long double a[][ N ], long double b[] ) ;

        void supsy( int extpr, int n, long double x[], long double y[], long double h[][3],\
                           long double g[][3], int bc[], long double ps[], long double
dpsdn[][3],\
                           long double b[], long double a[][ N ] ) ;

        void doneq( int n, int dnd, int pdnd[], long double x[], long double y[], long double
h[][3],\
                           long double g[][3], int bc[], long double ps[], long double
dpsdn[][3],\
                           long double b[], long double a[][ N ] ) ;

        void sgdta( long double wg[], long double xg[ ] ) ;

        void gauin( long double xg[], long double wg[], int ii, int ifu,                 \
                           long double xi,   long double yi, long double x1, long double y1, \
                           long double x2,   long double y2, long double dx, long double dy, \
                           long double d1,   long double *gi ) ;

        void intgr( long double xg[], long double wg[], int j, int j1, int j2,
        \
                           long double xi,   long double yi,  long double x1,  long double y1,
\
                           long double x2,   long double y2,  long double dx,  long double dy,
\
                           long double d1,   long double *h1, long double *h2, long double
*g1, long double *g2) ;

        void bouin( long double xg[], long double wg[], int ii,
        \
                           long double xi,   long double yi,  long double x1,  long double y1,
\
                           long double x2,   long double y2,  long double dx,  long double dy,
\
                           long double d1,   long double *h1, long double *h2, long double
*g1,\
                           long double *g2) ;

        void funct( int ii, int i,  long double xi, long double yi, long double x1,       \
                           long double y1, long double x2, long double y2, long double dx, \
                              long double dy, long double dl, long double s,  long double *f ) ;

       void solve( int n, long double a[][ N ], long double b[] ) ;

       void disol( int extpr, int n, int bc[], long double b[], long double ps[], long double
dpsdn[][3] ) ;

       void doval( int n, long double xi, long double yi, long double x[], long double y[], \
                                         long double h[][3], long double g[][3],
                              \
                                     long double ps[],   long double dpsdn[][3], long double
psi[] ) ;
```