Final Project:
Blasius Boundary Layer and Poisson's Equation

Ryan J. Krattiger*

Missouri University of Science and Technology, Rolla, MO, 65401, USA

The Blasius Boundary Layer equation and Poisson's equation were solved in this report. The Blasius was solved for the general form; however, the solution was not expanded on beyond the similarity solution. Poisson's equation was similarly solved, but for a set of solvable boundary conditions. The reason for this was to allow for a comparison of methods via error, which requires an exact solution. It should be clear from the examples provided that the functionality developed in these solvers is robust enough to apply the corresponding methods to a wide variety of scientific and engineering problems.

*Undergraduate Senior, Department of Mechanical and Aerospace Engineering.

Contents

Listings

Nomenclature

f	Function
$\bar{x}$	Variable value vector
ν	Dynamic viscosity
U	Velocity magnitude
h	Step size
x	x-position
δ	Boundary Layer thickness
$\bar{r}$	residual vector
Subscript	
i	Row index
j	Column index
e	Boundary layer edge

I. Introduction

For the final project it was perscribed to apply a minimum of six numerical methods from six unique topics covered in the Applied Computational Methods course. The problems choosen were to be from either relevent research or applications to engineering. The two problems choosen and presented in this report include a solution to the Blasius Boundary Layer equation and solution methods for Poisson's equation (commonly used in eletromagnitism, 2D Heat transfer, Subsonic flow fields, etc.). The Blasius equation used solved using the Runge-Kutta 4-step method coupled with the secant method to provide a smart 'shooting' method. Once this equation was solved, numerical integration was carried out to verify the solution. The Poisson equation was solved by first discretizing using central difference differentiation for a second derivative, then assembled into a system of linear equations and sovled using the Liebmann (Gauss-Seidel) Method and Steepest Decent Method.

II. Models

This section will highlight the models used in the final project and the respective solution methods implemented. The focus of this section is two parts. This first is to demonstrate the physical significance of the problem and the second is to show the general idea of what methods were implemented and why they were choosen.

II.A. Blasius Boundary Layer

The Blasiun equation is used to model boundary layer over a 2D flat plate as a similarity solution. The general form of this equation is given as Eq 1 where f is a function of η, the similarity parameter.

$$2f''' + ff'' = 0 \tag{1}$$

In order to do this accomplish this solution Blasius first converted (x,y) pairs to (ξ, η) using Eq 2.

$$\xi = x \quad \text{and} \quad \eta = y\sqrt{\frac{U}{\nu x}} \tag{2}$$

Using η it is then possible to define the function f(η) such that the stream function (ψ) can be expressed as Eq 3.

$$\psi = \sqrt{\nu U x} f(\eta) \tag{3}$$

From the stream function the x- and y-velocity components are then easily found by simply differentiating the stream function to obtain the results in Eq 4 and Eq 5 respectively.

$$u(x,\eta) = \frac{\partial \psi}{\partial y} = Uf' \tag{4}$$

$$v(x,\eta) = -\frac{\partial \psi}{\partial x} = \sqrt{\frac{\nu U}{x}}(\eta f' - f) \tag{5}$$

And further the shear stress at the wall, τ_{wall}, can be found using the gradient of the x-velocity with respect to η as seen in Eq 6.

$$\tau_{wall}(x) = \mu \frac{du}{d\eta}|_{\eta=0} = \mu U f''|_{\eta=0} \tag{6}$$

where μ is the dynamic viscosity.

Since it is clear that this ODE can be useful, it is now worth while to solve it. In order to solve this ODE using a Runge-Kutta 4-step scheme it is required to decompose the Eq 1 into a system of coupled first order ODEs. This decomposition can be seen in Eq. 7.

where

$$\bar{Z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} f \\ f' \\ f'' \end{bmatrix} \quad \text{and} \quad \bar{Z}' = \begin{bmatrix} z_1' \\ z_2' \\ z_3' \end{bmatrix} = \begin{bmatrix} f' \\ f'' \\ f''' \end{bmatrix} = \begin{bmatrix} z_2 \\ z_3 \\ -\frac{1}{2}z_1 z_2 \end{bmatrix} \tag{7}$$

Additionally the boundary conditions must be provided for the initial value problem that is to be solved. In this case it is known that the horizontal velocity (u), and so the stream function (ψ) must both be zero at the wall (y=0). Additionally, it is known that at the boundary layer edge, the velocity is equivelant to the freestream ($u = u_e$). This translates to the boundary conditions in Eq 8

$$f(0) = 0 \quad f'(0) = 0 \quad f'(\infty) = 1 \tag{8}$$

It is noticed that there is no initial condition for the z_3 or f" term. In to determine this value, it is required to perform the shooting method, or in layman's terms guess. The criteria that will be met to determine if the initial guess is good is to find where $f'(\eta_n) - 1 = 0$. But to do a little better than just guessing, the secant method will be used to find the zero of this criteria.

Solving $\bar{Z}'$ in order to determine what the values of f, f', and f'' are will be done using the Runge-Kutta 4-step (RK4) method dicussed in class. A note here that any method for solving an ODE could be used here but the RK4 was choosen for nostalgic reasons.

In the attached code the method of how the RK4 method was nested in the secant method can be seen. The essence of the idea was to wrap the RK4 solving the given ODE in a function of a single variable, $f''(0)$ or $z_{2,0}$, and return the result of $f'(\eta_{max}) - 1 = 0$. Then this was nested inside of another loop that check for convergence of $f''(0)$ as η_{max} increased in the domain $\eta_{max} \in [1, \infty)$ with increment steps of 1. This was done to capture the effect of ∞ while performing the minimum number of iterations. Note, as η_{max} increased the step size if the ODE solver remained at 0.1 using a scaling factor of 10 in order to maintain uniformity in each iteration.

The last test performed for this problem was with numerical integration. In order to verify that the f" and f' values agree, as they should from how the problem was defined, f" was integrated over the interval using simpson's 1/3 rule. The reason this method was choosen over the trapezoidal method or a gauss quadrature method was due to the higly contoured shape of the f" solution and the lack of an actual function to evaluate at specific points respectively.

II.B. Poisson's Equation

The poisson eqaution is used in many diciplines as it is a common mathematical model found in physical systems. Because of the relative complexity of these physical systems, a simplified and solvable example was used simple to demonstrate the application of two different solution methods.

The first thing to consider is the general form of Poisson's equation as seen in Eq 9. It is clear that this is a second order, linear, partial differential equation in three dimensions, x, y. The forcing function, f(x,y), can be anything depending on the problem and is one of five drivers for the solution.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \tag{9}$$

Being a second order PDE, it is required to have two boundary conditions for each independent variable. In this case that would be x and y, resulting in a required four boundary conditions.

For the problem to be solved here, the forcing function can be seen in Eq 10 and the boundary conditions are writen in Eq 11 and q 12.

$$f(x,y) = -2(x^2+y^2) \quad \text{for} \quad (0 \le x \le 1, 0 \le y \le 1) \tag{10}$$

$$u(x,0) = 1-x^2 \quad , \quad u(x,1) = 2(1-x^2) \quad \text{for} \quad 0 \le x \le 1 \tag{11}$$

$$u(0,y) = 1+y^2 \quad , \quad u(1,y) = 0 \quad \text{for} \quad 0 \le y \le 1 \tag{12}$$

It can be shown that the exact solution for these constraints is as seen in Eq 13.

$$u(x,y) = (1-x^2)(1+y^2) \tag{13}$$

Reguardless of the forcing function and boundary conditions, the set up of the problem is done so generically, independant of any set of inputs.

This first step is to descretize the original PDE as seen in Eq 9 into the form seen in Eq 14. It can be seen that the second order, central difference scheme was used.

Let $u(x_i, y_j) = u_{i,j}$, where $i \in [0, N]$, $j \in [0, M]$, and $h = \frac{x_N - x_0}{N}$ and $k = \frac{y_M - y_0}{M}$

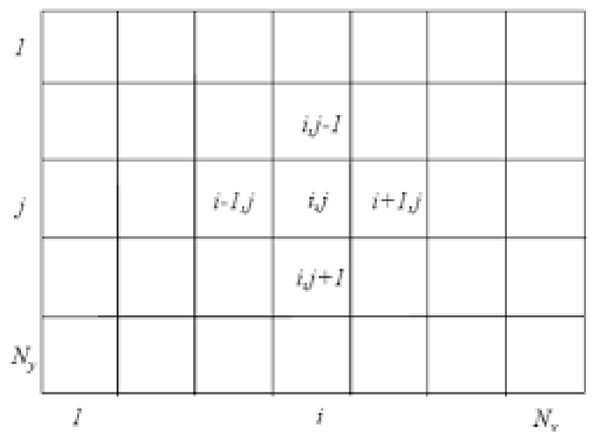
$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{k^2} = f(x_i, y_j) \tag{14}$$

Further simplifying this equation and letting $\frac{h^2}{k^2} = \lambda$ gives a general simplified form for the numerical scheme (Eq 15).

$$-2(\lambda+1)u_{i,j} + \lambda u_{i-1,j} + \lambda u_{i+1,j} + u_{i,j-1} + u_{i,j+1} = f(x_i, y_j) \tag{15}$$

After some carefull consideration it can be shown that the above eqation can be applied to the grid structure in figure II.B. It can also be seen that at the boundaries, where the values are known from boundary conditions, can be looked at as constants.

Figure 1. Grid with points used in central difference discretized Poisson



In order to put the now discretized equation into a solvable form it is translated to a system of linear equations. After some work a matrix-vector pair can be found and the system and be written in the form of $A\bar{x} = \bar{b}$

where A is an MxM block diagonal matrix

$$\begin{bmatrix} T & I & 0 & \dots & 0 \\ I & T & I & & \vdots \\ 0 & & \ddots & & 0 \\ \vdots & & I & T & I \\ 0 & \dots & 0 & I & T \end{bmatrix}$$

and T is a NxN tridiagonal matrix

$$\begin{bmatrix} -2(\lambda+1) & \lambda & 0 & \dots & 0 \\ \lambda & -2(\lambda+1) & \lambda & & \vdots \\ 0 & & \ddots & & 0 \\ \vdots & & \lambda & -2(\lambda+1) & \lambda \\ 0 & \dots & 0 & \lambda & -2(\lambda+1) \end{bmatrix}$$

and I is an NxN identity matrix.

In total, A is (NxM)x(NxM) which is quite large. Luckily, there is sparse matrix support in most linear algebra/numerical libraries used in computing to handle this.

The $\bar{b}$ vector is composed of two parts. This first is made up of the boundary conditions ($\bar{c}$), and the second is the forcing vector ($\bar{d}$). The resulting $\bar{b}$ is the denoted as $\bar{b} = \bar{c} + \bar{d}$.

$\bar{c}$ can be defined as

$$\begin{bmatrix} -(u_{1,0} + \lambda u_{0,1}) \\ -u_{2,0} \\ \vdots \\ -(u_{N-1,0} + \lambda u_{N,1}) \\ \lambda u_{0,2} \\ 0 \\ \vdots \\ 0 \\ \lambda u_{N,2} \\ \vdots \\ -(u_{1,M} + \lambda u_{0,M-1}) \\ -u_{2,M-1} \\ \vdots \\ -(u_{N-1,M} + \lambda u_{N,M-1}) \end{bmatrix}$$

$\bar{d}$ can be defined as

$$\begin{bmatrix} f(x_1, y_1) \\ f(x_1, y_2) \\ \vdots \\ f(x_{N-1}, y_{M-1}) \end{bmatrix}$$

And finally $\bar{x}$, the solution vector is the set of unknown nodes on the grid. The (i,j) coordinate of the grid is translated using a linear relation to give a vectorized form of the matrix such that

$$\begin{bmatrix} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ \vdots \\ u_{N-1,M-1} \end{bmatrix}$$

In the final form of $A\bar{x} = (\bar{c} + \bar{d})$ the Poisson equation is in a form that is solvable. The two methods that will be shown here are the Liebmann Method (also known as the Gauss-Seidel Method) and with a Steepest Decent optimization scheme.

II.B.1. Liebmann Method

As an iterative solving technique, Liebmann Method, also known as the Gauss-Seidel method, is best applied to sparse structured matrices rather than dense matrices. The reason for this is due to the need to perform O(N^2) operations, or one operation for each non zero element of the matrix, to compute the values of $\bar{x}$ in each iteration. If there are many non-zero values, there are more operations required for each iteration. Since the matrix proposed above is indeed sparse, this method is more desirable than a traditional Gauss-Eliminiation technique. Selecting an initial guess, say the $\bar{b}$ vector from above, it is possible to now begin iterations.

For iteration k+1, the general form for the x_i element is Eq 16 for $i \in 0...N$ and N is the size of the solution vector.

$$x_i^{k+1} = \frac{b_i - \sum_{j=0}^{i-1} x_j^{k+1} a_{i,j} - \sum_{j=i+1}^{N} x_j^k a_{i,j}}{a_{i,i}} \tag{16}$$

In implementation the solution vector would be a single variable, so as x_i was updated, the update would be seen when solving for x_{i+1}. With this in mind, the above Eq 16 can be changed into a simpler form seen in Eq 17, recalling that the projection of two vectors (or the dot product) is $\bar{a} \cdot \bar{b} = \sum_{i=0}^{N} a_i b_i$. Also let $\bar{a}_i$ be the vector created by the ith row of the matrix A.

$$x_i = \frac{b_i - \bar{a}_i \cdot \bar{x} + a_{i,i} x_i}{a_{i,i}} \tag{17}$$

By iterating of Eq 17 until convergence of the residual vector $||r_k||_2 \leq tolerance$. The reason to rewrite the solver in this way is to give way to allow the underlying operation of the dot product be optimized for sparse vector types. In this sense, for each product of the A matrix row and the current solution vector, a minimum number of operations will be performed. This also allows for the solver to work for multiple types of matrix problems and still obtain optimal performance without having to write a special solver for every sparse or dense matirx used. The same idea will be used again in the the next solver.

II.B.2. Steepest Decent

Another iterative solver can be used, however the Steepest Decent method looks at this problem as an optimization problem rather than a system of linear equations solving problem.

In general an optimization problem for a function F($\bar{x}$) is as follows:

Determine a search direction $\bar{s}_k$

Solve for the optimal α_k^* to find to minimum in the search direction use the knowns $\bar{x}_k$ and the search direction $\bar{s}_k$ and then use 1D minimization on F($x_{k+!}(\alpha_k^*)$). The below is an example of using the derivative of F to find the minimum.

$$F'(\bar{x}_{k+1}) = 0 \quad \text{where} \quad \bar{x}_{k+1} = \bar{x}_k + \alpha_k^* \bar{s}_k$$

Finally update $\bar{x}$

$$\bar{x}_{k+1} = \bar{x}_k + \alpha_k^* \bar{s}_k$$

and check if the solution meets convergence criteria using the norm of the residual vector ($\bar{r}_k = A\bar{x}_k - \bar{b}$).

$$||r_k||_2 \leq tolerance$$

The current problem to be solved is $Ax - b = \bar{0}$; however, a linear system is not a form that is particularly well suited for optimization since the zero vector is not the minimum. By integrating the system the resulting equation is $\frac{1}{2}x^T Ax - x^T b = 0$. This form of the equation does have a minimum as it is a quadratic system of eqations.

For the Steepest Decent method, the search direction is determined by the local direction of steepest decent, or the negative gradient. The formula for this should look familiar as Eq 18, which is the residual vector.

$$\bar{s}_k = \bar{r}_k = \bar{b} - A\bar{x}_k \tag{18}$$

The α_k can be found from the above method finding the root of the derivative as

$$\alpha_k = \frac{s_k \cdot s_k}{s_k \cdot \bar{q}_k}$$

where $\bar{q}_k = As_k$.

With the search direction (s_k) and optimal α_k^* it is a simple matter to update the solution vector. Again, this will iterator until convergence is found.

There are two additional methods that are linked to Steepest decent. The first is Conjugant Gradient (CGM) and the second is Preconditioned Conjugant Gradient (PCGM). These methods both provide much faster convergence on average, but require more computations per step. More informatoin about these, and a brief study on how they improve the gradient based methods as applied to the solution of 2D Poisson's equation can be found in Holmes et. al.[1]

III. Results

The results presented here are simply to show the results and relevent error or convergence of methods used.

III.A. Blasius Boundary Layer

In figure 2 the solution to the $\bar{Z}$ parameters are shown. The values plotted are labeled as stream function, u/u$_e$ and shear stress but are actually f, f', and f" respectively. The purpose of the naming was simply to demonstrate the general profile of the flow for the relevent parameters calculated using f, f', and f".

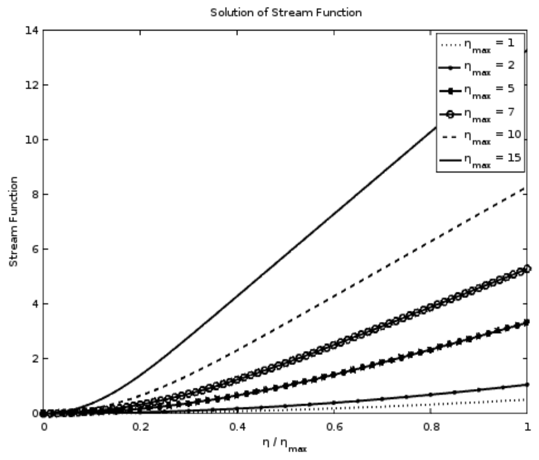
A second check to verify the solution obtained, the f" result was integrated over the range in the converged iteration, $\eta_{max} = 15$.

$$\int_0 15 f''(\eta) d\eta = f'(15) - f'(0) = 1$$

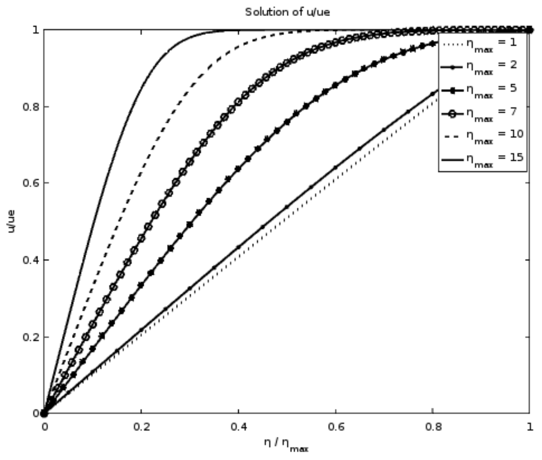
The result of this integration can be seen in the table table III.A. It can be seen that the numeric integral solution and the expected soltion are very close, especially for the integration results coming as the result of two seperate numerical methods.

Solution	Error
1.000023279653586	2.327965358617234e-05

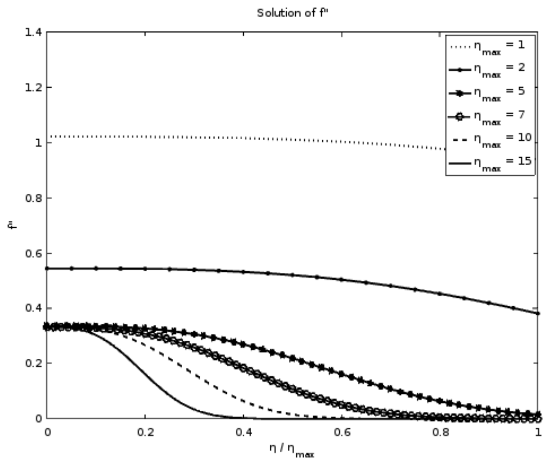
Figure 2. Results from solving the Blasius Boundary Layer equation



(a) Stream Function



(b) Scaled Velocity Profile



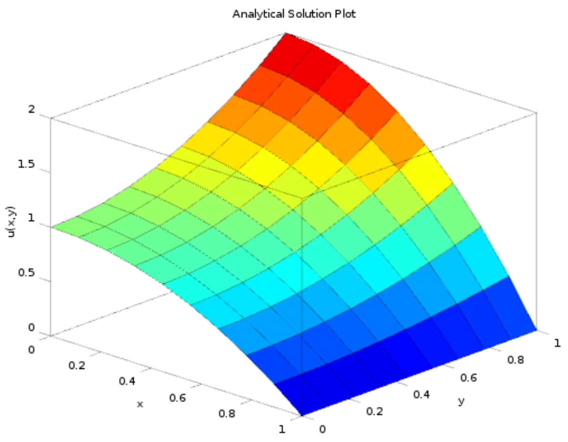
(c) Wall Shear Stress

III.B. Poisson's Equation

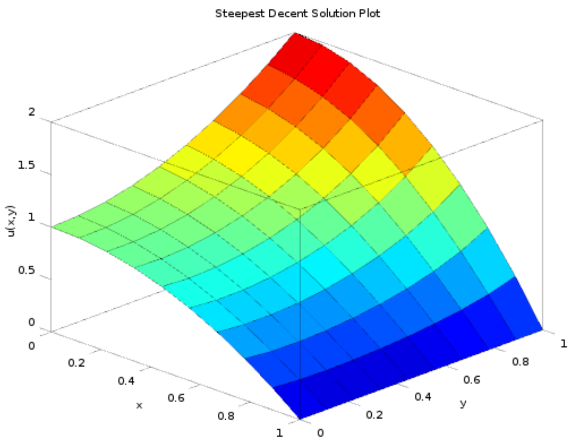
The figures 3 show the solution plots of each respective method. The plot of the norm of the residual as a function of iteration for each method is presented in figure 4. It can be seen that the steepest decent has a much slower convergence rate compared to the Liebmann method, taking roughly twice as many iterations to converge. The results from comparing the solution method for a 20x20 and 30x30 grid are not included here, but similar conclusions can be drawn.

Notice that for both methods the number of iterations required is greater than N. It may be tempting to say that this makes the cost of each of these methods roughly equal to that of a Gauss-Elimination as each step appears to be around O(N^2); however, that would be incorrect. Because of the sparse nature of the matrix a good numerical library will leverage the fact that most values are zero and only perform operations on the non-zero values in the matrix and vector. In Matlab, this may not always be the case. In the numerical library that was suppose to be used to solve this problem, and will be shortly after this paper is submitted, the described advantages are leveraged. So the actual complexity is only O(N) operations rather than O(N^2), which puts both of these methods ahead with a total complexity of roughly O(N^2).

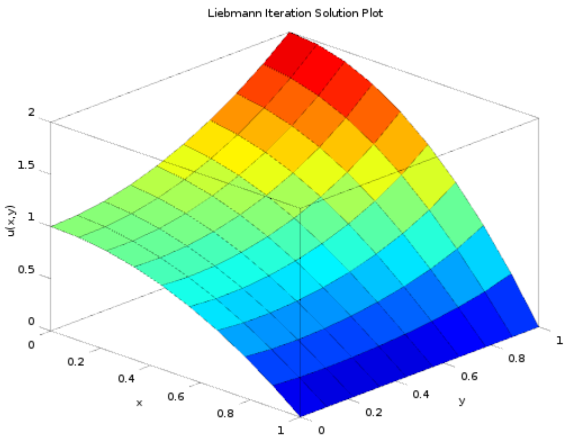
Figure 3. Results from solving the Blasius Boundary Layer equation



(a) Exact Solution

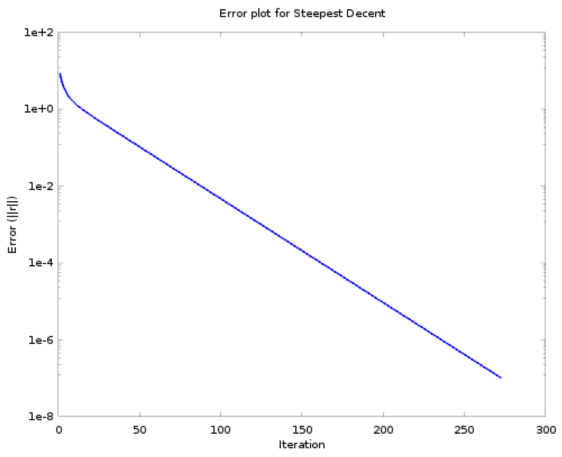


(b) Steepest Decent Solution

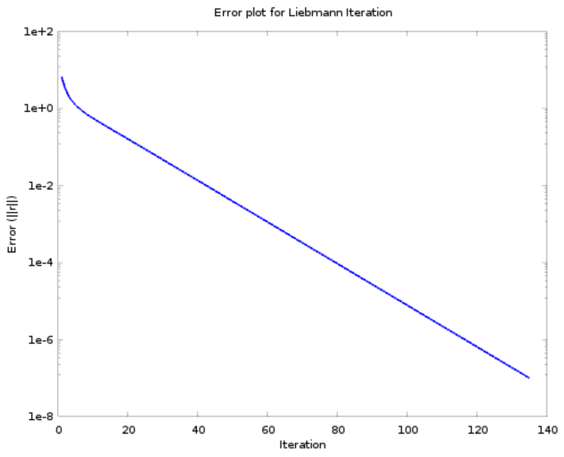


(c) Liebmann Solution

Figure 4. Results from solving the Blasius Boundary Layer equation



(a) Steepest Decent error vs iteration



(b) Liebmann error vs iteration

IV. Conclusion

It is clear that the numerical methods presented, when applied correctly as they have been here, are capable of providing extraordinarly powerful tools that allow humans to use computers to solve ever more complicated problems that occur in science and engineering. The study and advancement of numerical methods to solve these problems and make more robust and smarted (adaptive) solvers has been the key to increasing design and analysis speed in the past and will continue to be so in the future. Not to be excited about computing in this time is lame, which is why this project has been a pleassure to complete.

References

[1]Holmes, M. H., *Introduction to numerical mehtods in differential equations*, New York, London: Springer, 2007.

[2]Anderson, J. D., *Fundamentals of aerodynamics*, Boston: McGraw-Hill, 2001.

[3]Chapra, S. C., and Canale, R. P., *Numerical Methods for Engineers*, McGraw-Hill, 2015.

[4]Numerical methods, – *CFD-Wiki, the free CFD reference* Available: `http://www.cfd-online.com/wiki/numerical_methods`.

[5]Hosder, S., Applied Computaitonal Methods.

Appendix

Listing 1. Main for Blasius Solver

```
#include "blasius.h"
#include "../../include/integration/integrators.h"
#include <fstream>


int main()
{
  numlib::Vector<numlib::Vector<double>> solution;
  numlib::Vector<double> fpp_toint;
  std::ofstream fout;
  numlib::converge::tolerance<double> tol;
  double upper = 1, s_old=0, diff = 1;;

  fout.open("results.m");

  while(diff > tol.HYPER_FINE)
  {
    solution = blasius(upper, upper*10);
    diff = fabs(solution[0][2] - s_old);
    s_old = solution[0][2];

    if(upper == 1 || upper == 2 || upper == 5 || upper == 7 || upper == 10)
      fout << "blas_" << static_cast<int>(upper) << "␣=␣[" << solution << "];\n\n";

    upper++;
  }

  fpp_toint.set_size(solution.get_size());
  int it = 0;
  for(auto& v: solution)
    fpp_toint[it++] = v[2];

  std::cout << fpp_toint << '\n';

  double integral_s = simpson3(fpp_toint, {0,upper-1}), integral_ode = solution[(upper-1)*

  std::cout << std::setprecision(15) << integral_s << '\n';
  std::cout << fabs(integral_s - integral_ode) << '\n';
  fout << "converge_at␣=␣" << upper - 1 << ";\n";
  fout << "blas_converged␣=␣[" << solution << "];\n\n";



  fout.close();

  return 0;
}
```

Listing 2. Implementation of Blasius Routine

```
#include "blasius.h"
#include <iostream>
```

```
numlib::Vector<numlib::Vector<double>>
blasius(
  const int ubound,
  const int n)
{
  numlib::Vector<numlib::Vector<double>> solution;

  auto fode = [](const double dummy, const numlib::Vector<double> z)
  {
    return numlib::Vector<double>({z[1],z[2],-0.5*z[0]*z[2]});
  };

  auto f = [&](const double x) -> double
  {
    solution = rk2_heun(fode,
                        numlib::Vector<double>({0,0,x}),
                        numlib::Vector<int>({0,ubound}),
                        n);
    return solution[n][1]-1;
  };

  secant(f,0.3,0.4);

  return solution;
}
```

Listing 3. Octave to Plot Results for Blasius

```
results

vname = cell(3);
vname{1} = 'Stream␣Function';
vname{2} = 'u/ue';
vname{3} = 'f"';

for item=1:3
  figure(item)
  hold on
  plot(linspace(0,1,max(size(blas_1))/3), blas_1(item:3:end), 'LineWidth', 2, ':k')
  plot(linspace(0,1,max(size(blas_2))/3), blas_2(item:3:end), 'LineWidth', 2, '.-k')
  plot(linspace(0,1,max(size(blas_5))/3), blas_5(item:3:end), 'LineWidth', 2, '*-k')
  plot(linspace(0,1,max(size(blas_7))/3), blas_7(item:3:end), 'LineWidth', 2, 'o-k')
  plot(linspace(0,1,max(size(blas_10))/3), blas_10(item:3:end), 'LineWidth', 2, '--k')
  plot(linspace(0,1,max(size(blas_converged))/3), blas_converged(item:3:end), 'LineWidth',
  title(['Solution␣of␣',vname{item}])
  legend('\eta_{max}␣=␣1','\eta_{max}␣=␣2','\eta_{max}␣=␣5','\eta_{max}␣=␣7','\eta_{max}␣=
  xlabel('\eta␣/␣\eta_{max}')
  ylabel(vname{item})
end

input('Pause...');
```

Listing 4. Main for Poisson's Equation

```
clear all
close all
clc
```

```
% Input constraints
x_min = 0;
x_max = 1;
y_min = 0;
y_max = 1;

lx0_bound = @(yy) 1 + yy.^2;
ux1_bound = @(yy) 0;

ly0_bound = @(xx) 1 - xx.^2;
uy1_bound = @(xx) 2.*ly0_bound(xx);

f = @(xx,yy) -2*(xx^2 + yy^2);

% Solution grid sizing
N = 10;
M = 10;
n = N-2;
m = M-2;

x = linspace(x_min,x_max,N);
y = linspace(y_min,y_max,M);

lam = (m/n)^2;

% Initialize A and b building blocks
I = spdiags([1].*ones(n,1), [0],n,n);
B = spdiags([1 -2*(lam + 1) 1].*ones(n,1), [-1 0 1],n,n);
b = zeros(n*m,1);
U = zeros(N,M);

% Assign boundary conditions to solution matrix
U(1,:) = lx0_bound(y);
U(end,:) = ux1_bound(y);
U(:,1) = ly0_bound(x);
U(:,end) = uy1_bound(x);

% Initialize A as a sparse matrix
A = sparse(m*(n+1)*n/2 + m*n);

% Assign values to A
A(1:n,1:n) = B;
A(1:n,n+1:2*n) = I;

for j=n+1:n:(m-1)*n
  A(j:j+n-1,j:j+n-1) = B;
  A(j:j+n-1,j+n:j+2*n-1) = I;
  A(j:j+n-1,j-n:j-1) = I;
end

A(n*(m-1)+1:n*m, n*(m-1)+1:n*m) = B;
A(n*(m-1)+1:n*m, n*(m-2)+1:n*(m-1)) = I;

% Assign values to b
index = 1;
```

```
for i = 2:1:n+1
  for j = 2:1:m+1
    % b(index) = f(x(i),y(j));

    if(i==2)
      b(index) = b(index) - lam*lx0_bound(y(j)); index;
    end

    if(i==n+1)
      b(index) = b(index) - lam*ux1_bound(y(j)); index;
    end

    if(j==2)
      b(index) = b(index) - ly0_bound(x(i)); index;
    end

    if(j==m+1)
      b(index) = b(index) - uy1_bound(x(i)); index;
    end
    index = index + 1;
  end
end
b;

% u_inner = A\b;
[u_inners, its, errvs] = sd_solver(A,b);
its
[u_innerl, itl, errvl] = liebmann(A,b);
itl

fign=1;

figure(fign)
semilogy(1:1:its-1,errvs, 'LineWidth', 2)
title('Error␣plot␣for␣Steepest␣Decent')
xlabel('Iteration')
ylabel('Error␣(||r||)')
fign = fign + 1;

figure(fign)
semilogy(1:1:itl-1,errvl, 'LineWidth', 2)
title('Error␣plot␣for␣Liebmann␣Iteration')
xlabel('Iteration')
ylabel('Error␣(||r||)')
fign = fign + 1;

Us = U;
Ul = U;

index = 1;
for i = 2:1:n+1
  for j = 2:1:m+1
    Us(i,j) = u_inners(index);
    Ul(i,j) = u_innerl(index);
    index = index + 1;
    end
end

[X,Y] = meshgrid(x,y);

u_af = @(xx,yy) (1 - xx.^2).*(1 + yy.^2);

u_a = u_af(X,Y);

figure(fign)
hold on
surf(X,Y,Us')
title('Steepest Decent Solution Plot')
ylabel('y')
xlabel('x')
zlabel('u(x,y)')
fign = fign + 1;

figure(fign)
hold on
surf(X,Y,Ul')
title('Liebmann Iteration Solution Plot')
ylabel('y')
xlabel('x')
zlabel('u(x,y)')
fign = fign + 1;

figure(fign)
hold on
surf(X,Y,u_a)
title('Analytical Solution Plot')
ylabel('y')
xlabel('x')
zlabel('u(x,y)')
fign = fign + 1;
```

Listing 5. Implementation of Steepest Decent

```
function [x, it, errv] = sd_solver(A,b)
    x = b;
    sk = b - A*x;
    it = 1;
    while(norm(sk) > 1e-7)
        alpha = dot(sk,sk)/dot(sk,A*sk);
        x = x + alpha*sk;
        sk = b - A*x;
        errv(it) = norm(sk);
        it = it + 1;
    end
end
```

Listing 6. Implementation of Liebmann Iteration

```
function [x, it, errv] = liebmann(A,b)
    x = b;
    r = b - A*x;
    it = 1;
```

```
  while(norm(r) > 1e-7)
    for i = 1:1:max(size(b))
      x(i) = (b(i) - dot(A(i,:),x) + A(i,i)*x(i))/A(i,i);
    end
    r = b - A*x;
    errv(it) = norm(r);
    it = it + 1;
  end
end
```