AIAA AVIATION Forum
August 2-6, 2021, VIRTUAL EVENT
AIAA AVIATION 2021 FORUM

10.2514/6.2021-2458

VORLAX 2020: Making a Potential Flow Solver Great Again

Tyler J. Souders[1] and Timothy T. Takahashi[2]
Arizona State University, Tempe, AZ, 85287-6106

***VORLAX* is a vortex-lattice potential flow solver written by Luis R. Miranda for Lockheed California in the early 1970's. While the tool has remained a viable solution for resolving shock-free flow conditions on a wetted surface, many aspects of the code have become dated, lowering its practicality. This paper describes methods to improve this code centering on revising the solver and memory management techniques. Through simple code changes, the integrity of the original method has remained intact while yielding performance improvements in excess of 90% in some usage cases.**

Nomenclature

α	=	Angle of Attack (deg)
C_D	=	Drag Coefficient
C_L	=	Lift Coefficient
ITRMAX	=	Maximum Number of Iterations
LAX	=	X-Direction Spacing Method
LAY	=	Y-Direction Spacing Method
M	=	Mach Number
N	=	Number of Control Points
NVOR	=	Spanwise Control Points
R	=	Residual
RNCV	=	Chordwise Control Points
t/c	=	Thickness Normalized to Chord
x/c	=	X-Dimension Normalized to Chord
y/c	=	Camber Displacement Normalized to Chord

I. Introduction

MODERN computational fluid dynamics (CFD) are integral to the design of any aircraft. Companies such as Boeing rely heavy on the aerodynamic insights provided by CFD, though the insights come at great cost in terms of computing resources [1]. Years ago, vortex-lattice methods were commonly used as aerodynamic design tools, providing accurate flow resolution despite the technical limitations at the time. Historically, the cost of running the vortex-lattice simulations may have been comparably high as the computational cost of modern CFD suites, such as *ANSYS Fluent*. However, as technology has progressed, vortex-lattice methods have fallen out of use, often in favor of higher-precision solvers.

Despite the fact that the modern suites may provide better resolution on a microscopic scale, they are merely more accurate than a vortex-lattice method. Nothing has happened in the last 40 years to suddenly make the vortex-lattice methods obsolete, and in fact they remain very lightweight and powerful tools to analyze the flow over an aircraft. This paper will show the relative speed of a particular vortex-lattice method, *VORLAX*, when better optimized to take advantage of the power of modern PC's.

[1] M.S. Candidate Mechanical Engineering, School for Engineering of Matter, Transport & Energy, P.O. Box 876106, Tempe, AZ. Student Member AIAA
[2] Professor of Practice, School for Engineering of Matter, Transport & Energy, P.O. Box 876106, Tempe, AZ. Associate Fellow AIAA

II. What is *VORLAX* ?

A. General Usage and Capabilities

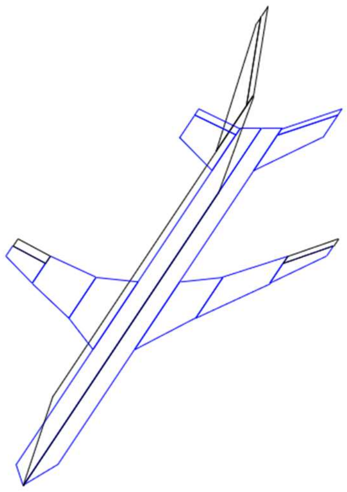
FIGURE 1 *VORLAX* Representation of Boeing 737-300

VORLAX is a potential flow solver utilizing a generalized vortex lattice method to resolve flow field behavior for shock-free, attached-flow conditions [2]. *VORLAX* is one specific example of an entire family of vortex lattice methods, with other examples existing such as *Tornado*, which has been developed to run in *MATLAB*. While other methods exist, they tend to be much more complex, either relying on proprietary software (such as *MATLAB*) or having rather interactive user interfaces that take away from the main benefits of using a vortex lattice software.

The reason that *VORLAX* remains such a powerful and useful software is because of its framework developed in FORTRAN IV. While the code has been updated during the years, having features most comparable to FORTRAN 77, modern compilers are capable of handling the old syntax via legacy flags. Thus, *VORLAX* remains in a form capable of running on any modern Windows PC. *VORLAX* reads input files which define flight configuration information and geometric properties of the body in a simple 10-column format, allowing rapid preprocessing of hundreds of test cases using a simple scripting language capable of writing text files, such as *MATLAB*, VBA, or Python. FIGURE 1 shows a drawing of an entire Boeing 737-300 using VORLAX, while FIGURE 2 shows the standard coordinate plane used to define the panel locations.

VORLAX offers multiple usage modes. This includes running at multiple Mach numbers, multiple angles of attack, and sideslip angles, with the additional ability to resolve pitching, rolling, and yawing moments as well as the dynamic derivatives for the configuration. For each aerodynamic configuration, one may represent the structure as a series of flat panels, upon which the vortex lattice structure is defined; see FIGURE 1. It is possible to include both thickness and camber effects on any of the panels by including chord-normalized coordinates along the panels. The user may also include a fusiform, cylindrical body in their analysis, useful for representing a fuselage or engine. Finally, *VORLAX* allows the user to work with either sub- or supersonic flow conditions, but inherently lacks the ability to resolve shocks or other "jump" conditions in the transonic regime.

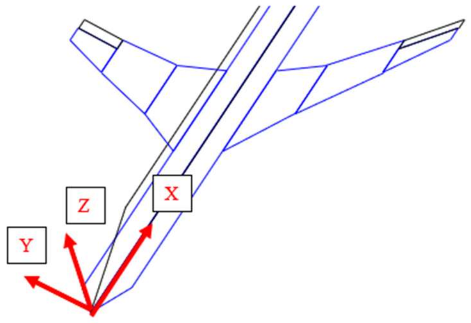

FIGURE 2 VORLAX Panel Method Representation of an Aircraft

B. Flat Panel Mode

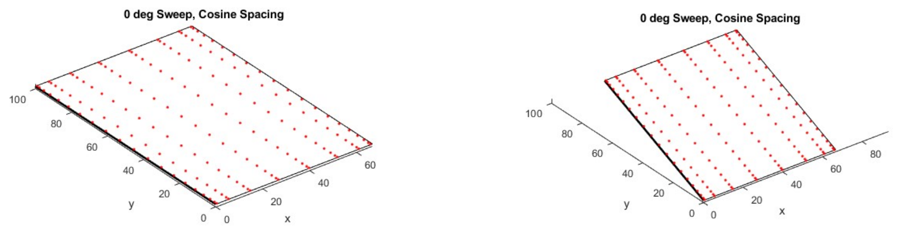

FIGURE 3 Flat Panel Visualization with Grid Points

The most basic of geometric representations in *VORLAX* is the "flat panel" mode. This works by representing the geometry by a series of two-dimensional, infinitesimally thin plates. FIGURE 3 shows a standard arrangement of these plates in both a swept and unswept configuration. The code enforces a zero-mass flux condition, definable by the user, to compute the circulation strength of the horseshoe vortices comprising the lattice.

Each panel is defined by two collinear leading-edge coordinates in conjunction with station chord lengths, which the code uses to draw the two-dimensional panel. Such a configuration allows the user to define sweep at both the leading and trailing edge using a simple set of coordinates and trigonometry. The code supports up to twenty individual panels, each of which can be mirrored across the central axis of the aircraft without counting as an additional panel. Thus, an entire airframe, including a main wing with multiple control stations, a vertical tail, a horizontal tail, and a fuselage can be drawn very easily.

The ease of pre- and postprocessing is one of the strongest usage cases of the flat panel mode. The input, output, and log files produced by *VORLAX* are formatted very systematically, lending themselves to very user-friendly automation. The flat panel mode can deliver accurate results about the pressure distribution, aerodynamic performance data, and stability and control derivatives in very little time. Generally speaking, the error of the flat panel configuration is within a few percent of real-world testing, thereby offering useful insights in a fraction of the time required with complex CFD suites, such as *ANSYS Fluent*.

C. Cambered Panel Mode

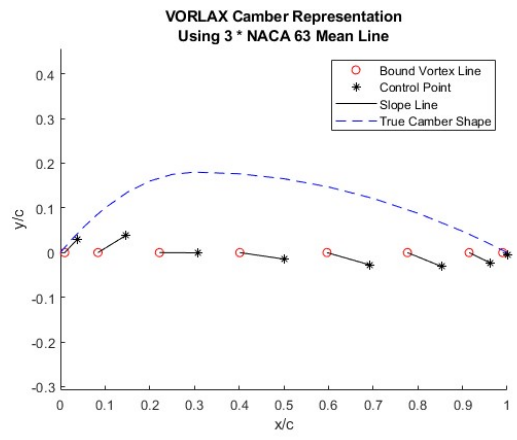

FIGURE 4 VORLAX Camber Representation

The cambered panel mode builds seamlessly from the flat panel mode. *VORLAX* can represent a cambered plate by adjusting the direction of the local normal vector used in the zero-flux boundary condition. Because of the theory governing the vortex lattice method, there is no significant change necessary, other than altering a coefficient attached to the dot product producing the normalwash component. FIGURE 4 shows the *VORLAX* equivalent drawing of a NACA 63 mean line to visualize how the code sees the shape.

Camber is important in wing design because camber effects the lift generated by a wing section and contributes to the three-dimensionality of the airflow over the wing [3]. Thus, adding camber is a relatively simple way of obtaining more accurate aerodynamic performance data for the aircraft, such as better resolution of zero-pitch lifting coefficient and the drag associated with it. Because *VORLAX* will calculate the pressure distribution over each vortex point, it also becomes the first step in determining the critical Mach number for the flow configuration and determining the efficiency of the wings, per the methods described by Oswald [4][5].

D. Thick "Sandwich" Panels

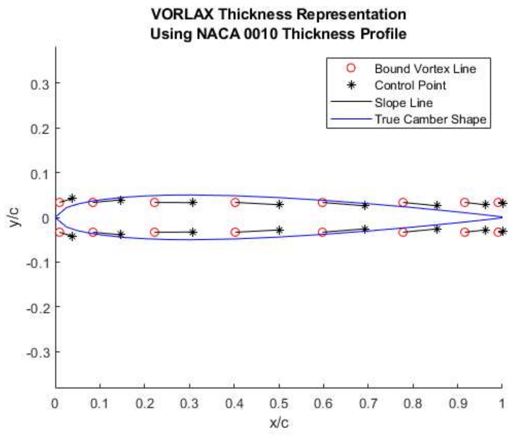

FIGURE 5 VORLAX Thickness Representation

Continuing from the discussion regarding cambered panels, it is possible to simulate wing thickness effects, including thickness tapering, by arranging two flat or two cambered panels with a small amount of separation between one another. FIGURE 5 shows that the manner in which the normal coefficient is resolved is nearly identical in nature to that of the standard cambered panel, with the minor difference that the zero-mass flux condition is applied only to the outer "wetted" surface. Miranda described the optimal distance of panel spacing to be about 2/3 that of the local maximum thickness, given via

$$z = \pm\frac{1}{3}\left(\frac{t}{c}\right)_{\text{max}}(y) \tag{1}$$

where y is the station location along the span [2]. Thus, by perturbing the z-coordinate of the vortex point in the body frame, it becomes possible to include the effects induced by wing thickness. To ensure correct flow relations over these spaced panels, the code only enforces the zero-mass flux condition on the wetted surfaces, as defined by the user. By leaving the boundary condition flag as user-defined, it allows the program to remain robust. Our companion paper, AIAA 2021-xxxx [6], discusses the applications and paneling strategies for *VORLAX.*

Furthermore, *VORLAX* allows the inclusion of thickness effects in conjunction with cambered effects. This is done by superposing the changes to the local normal vector in the boundary condition application. This is also simplified by allowing the user to include the effects from both thickness and camber in one single input. Thus, the true station location is given via

$$\left(\frac{y}{c}\right)_{\text{Top}} = \frac{1}{2}\left(\frac{y}{c}\right)_{\text{Thickness}} + \left(\frac{y}{c}\right)_{\text{Camber}} \tag{2a}$$

$$\left(\frac{y}{c}\right)_{\text{Bottom}} = -\frac{1}{2}\left(\frac{y}{c}\right)_{\text{Thickness}} + \left(\frac{y}{c}\right)_{\text{Camber}} \tag{2b}$$

in the chord reference frame. Similar to the case with camber, this allows for better critical Mach number prediction and wing efficiency determination. While these modes are very useful for real, nuanced wing design, the relative disturbances are generally unnecessary for accurate stability and control data, due to their relatively small effect when compared to the massive overall forces experienced on the tail surfaces.

E. Fusiform Body

In addition to flat panels, *VORLAX* has the capability to represent cylindrical "fusiform" shapes. Most applicable to the fuselage and engine nacelles, the fusiform body constructs the cylinder from discrete, two-dimensional plates. The user can describe station/radius pairs at which the control points are defined. Thus, for these radii of the body, a "ring" of 2D panels forms the set of horseshoe vortices necessary for computing the pressure distribution about the body; see FIGURE 6.

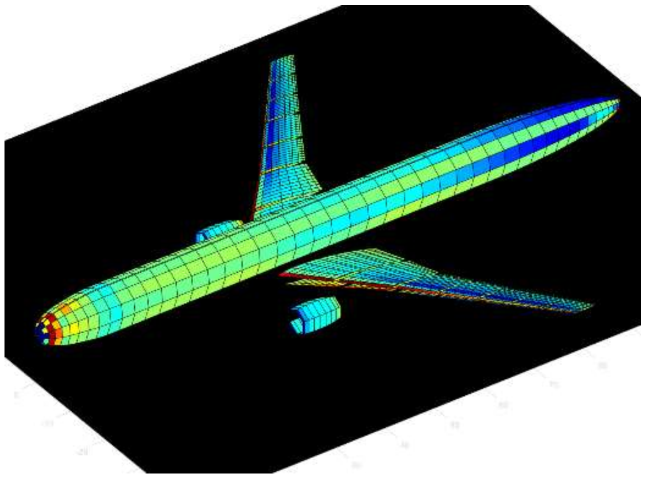

FIGURE 6 Sandwich Panel with Fusiform Body (for Fuselage and for Nacelles)

F. Supersonic Capabilities

In the theory governing *VORLAX*, there is nothing prohibiting supersonic flow calculation. To include supersonic effects, the code simply treats the subsonic vortex filament quantity as a numerical representation to the surface vorticity density, rather than as a true physical quantity [2]. Because *VORLAX* cannot, by limitation of the governing theory, handle transonic flow shock-jump relations, this change to supersonic consideration happens when the user inputs a Mach number exceeding 1 in the input file, i.e. $M_\infty \geq 1$. By altering the way the discrete panel locations are handled with supersonic flow, *VORLAX* computes a contribution to the velocity field, thereby tying in with the subsonic flow theory and contributing to the small perturbation method utilized to calculate the flow.

III. Classic *VORLAX* Solvers

A. Background

VORLAX was originally designed with two built-in numerical solving routines, one applying Purcell's Vector Method and the other applying a controlled successive over-relaxation method, CSOR. [7][8] The solvers are necessary to solve a dense system of linear equations that compute the vortex strengths based on the applied boundary condition. The solvers are called in the code sequence after reading the geometric input file and constructing the matrices to be solved, and before determining the flow conditions.

The reason for the implementation of iterative methods lies in the nature of the system of equations to be solved. The system is linear, however it is filled with nonzero values, has no diagonal dominance, nor is it positive-definite. While the solution may be obtained using a direct-solve procedure, such procedures are inefficient, making them unfavorable in the modern iteration of *VORLAX*, and making them impossible with the memory constraints present in the era that *VORLAX* was written. Furthermore, one inherent challenge with the implementation of the vortex-lattice method is that the influence coefficient system of equations is not sparce. Thus, for any N grid points, the computer is required to compute and store an $N \times N$ matrix, which proves troublesome. Unlike finite difference methods, which store their discretization coefficients in vectors of length N, the arrays of $N \times N$ take up an exponentially larger amount of RAM. *VORLAX* remains an ia-32 compile, due to the fact that the single precision compile typically runs faster and is better for redistribution to students on low-power machines.

VORLAX traditionally utilized scratch files in a machine-readable form during its execution. This is reminiscent of the tapes used in legacy IBM systems consistent with the era in which *VORLAX* was written. While this was a good option with the technical limitations of the time, the practice is outdated and slow by modern standards. However, in order to better characterize the performance improvements present in the new compile, the baseline benchmark cases utilized this antiquated method.

B. Gauss-Seidel Controlled Over-Relaxation

The Gauss-Seidel Controlled Over-Relaxation method is a slightly modified Block Gauss-Seidel method. The difference lies in the way the over-relaxation factor is handled in the solution. Rather than prescribing a single, fixed over-relaxation factor, α, the method defines two factors, $\alpha_1 = 1.1$ and $\alpha_2 = 0.9$. By tracking the relative changes in the solution vector, the convergence is characterized as either monotonic or oscillatory. Thus, when the convergence is monotonic (i.e. converging to a set of values), the *over*-relaxed parameter, α_1, is applied. Conversely, when the solution behavior is of oscillatory fashion, the *under*-relaxed parameter, α_2, is applied. By implementing two different factors, the iterative approach may accelerate convergence, while dampening the "overshoot" instabilities present with typical SOR methods.

C. Purcell's Vector Method

Purcell's Vector Method was originally published in 1952. The methods works by putting the system of linear equations into the standard $\boldsymbol{A}\vec{x} = \vec{b}$ format, though instead of the typical $N \times N$ spacing as one would expect, the matrix A takes the dimensions of $N \times (N + 1)$. The reason for the introduction of the additional column is because it provides an additional term in the system which exists to enforce homogeneity in the right-hand side of the equation. Thus, because of this extra column, the vectors used in the computation are all of length $N + 1$ rather than of length N. This is relevant because it destroys the consistency of the source code indexing used for all of the other solvers. Instead of working in the range of (1, ITOTAL), where ITOTAL is the total number of grid points, this subroutine has constant references in the range of (1, ITOTAL+1).

The method works by multiple different vectors that are solved in order to obtain a single solution vector that is orthogonal to each of the other rows in Matrix $\boldsymbol{A}$. Thus, there existed a "direct solution" solver in the original version of *VORLAX*. This left the previous engineers with two solving methods: CSOR and Purcell, one of which was an iterative solver, and the other direct. Given the computational limitations of the time, both methods were expected to run incredibly slow, but they were dependable.

IV. Known Bugs

At the time of writing, *VORLAX* has a few bugs that have remained in the code since its original development in the 1970's. The first of which is that the method for integrating the induced drag C_{D_i} falls apart when using the "linear" spacing option. While this is easily remedied by utilizing the alternative "cosine" grid spacing, it makes verification and validation of the results more difficult due to the inability to maintain equidistant grid spacing during grid refinement studies. Unfortunately, this bug is a direct consequence of the method utilized for the computation of the leading-edge thrust coefficient. VORLAX utilizes Lan's Method for leading-edge thrust computation, which inherently requires that the spacing along the chordwise component of the panel is of the cosine type [9]. To avoid this conflict, the code was modified to disallow the specification of linear chordwise control point spacing when running VORLAX with a subsonic freestream Mach number, thereby preventing ambiguity between results. There are also bugs when computing the induced drag with a sandwich panel configuration, discussed in greater detail in the companion paper [6]. Generally, these errors are resolved by using a flat panel equivalent, which calculates drag with much better accuracy.

Previously, there were bugs with the methodology of generating the fusiform bodies. Previously, panels were drawn under the misconception that the defined points were the locations of the control points, while that is not the case. Using advanced visualization tools (particularly those which can handle 3-D fusiform body presentation), this misconception was cleared. While it was previously believed that the user was defining the location of the control points in radial coordinates, it is now known that the user is defining the location of the vertices between which the control points are drawn. Thus, the flag "NVOR" relates to the number of vertices to be drawn, minus one, while the "RNCV" flag dictates the number of "chordwise" locations along the fusiform body at which the radial coordinates are drawn according to the thickness profile provided by the user.

Because of the panel generation method in fusiform construction, there were further bugs involved in the development of the visualization tool. This led to gaps between the panels that were not included with the contours. To better generate pressure visuals, some tricks were employed in the code to interpolate between these gaps in order to complete the image. This does not introduce any inaccuracies because it uses the same type of interpolation as between the other vertices, however it had to be deliberately extended to include the edges of the mirrored domain.

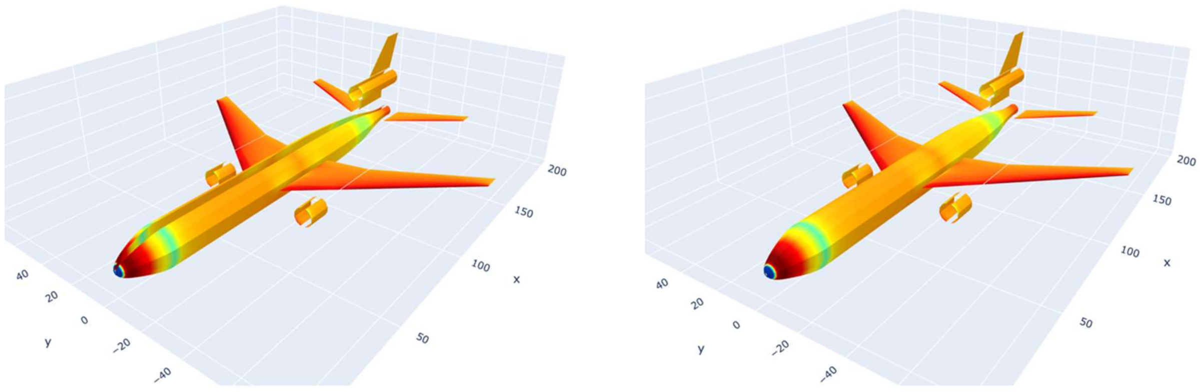

FIGURE 7 Visual Improvements to Fusiform Visualizer

VORLAX originally had a "synthesis mode", in which the user could define a basic panel structure that the code would use to optimize the camber of a wing for a user-defined loading configuration. This mode is only compatible with the CSOR solver, as written by Miranda in the original compile. While the design tool seems useful, it has fallen out of use dating back 30+ years. The current equivalent method of attaining a desired loading is a type of "guess and check" approach using a script to write and check dozens of input files, however this method is unable to "draw up" a wing using the inverse approach and solving backwards. Thus, an operational design mode would be a large leap in terms of efficient design via *VORLAX*.

Finally, the wake survey tool had fallen into disuse due to a runtime error generated when using this feature. Work done by Professor Takahashi made the tool viable for use once again. Much like real-world testing, the wake survey feature operates by inserting planes perpendicular to the freestream flow and calculates the flow velocity components normalized to the freestream (as defined by the user). By extension, with trivial calculations with the dataset the user may compute the local vorticity by employing finite-difference schema. Users may use this feature both to determine the flow components relative to the aircraft (useful for determining optimal propellor angle, for instance) and may also use it to visualize the vorticity concentrations, useful for intuitively analyzing phenomena such as wingtip vortices, an integral part of induced drag.

V.Solver Improvements

A. Testing Setup:

Table 1 Grid Densities

	NVOR	RNCV
Config. 1	25	10
Config. 2	50	20
Config. 3	100	40

VORLAX 2020 was developed and benchmarked on a consumer-grade Windows PC running Windows 10 with an Intel Core i9-9900K and 32GB of DDR4 RAM running at 2666MHz. The test input was a single AR = 20 wing in three grid configurations. Each grid density is defined by the number NVOR of spanwise stations and RNCV chordwise stations (Table 1). The configurations were run at a three freestream Mach numbers and three angles of attack. The grid spacing on the panel was varied in accordance with parameters described in the subsequent tables detailing performance results. When inspecting the runtime of the program, the most reliable metric is the "wall time" computation requirement, which was most accurately reported via CPU time tracking within the *VORLAX* program and printing to the log file. This was preferable compared to timing the run in VBA, as it takes time for the script to spawn the command line shell and execute the batch file, thereby creating some variance external to the *VORLAX* program itself.

B. Legacy Summary and Goals

While the old solving methods were not bad, they left much to be desired when considering modern computational resources. *VORLAX* has its main strength as a robust, *fast*, method, and thus making the code as fast as possible is desirable to outweigh the fact that it is a “90% synthesis tool”. Thus, it was imperative to explore the source code of the classic *VORLAX* compile found in the NASA CR [2] and update it to modern standards. Upon inspection, there were two main areas of exploration for improving *VORLAX* as a rapid aerodynamic analysis tool. The first area involved moving *VORLAX* into an “in-memory” solver. This change removed the need for read/write cycles with a scratch file saved on the hard drive, and instead utilized computer RAM in order to store arrays.

The second, more strenuous, area of exploration was that involving the solving methods themselves. It was critical that this step came after moving the code to a RAM based configuration, as scratch files were an outdated technical practice that broke compatibility with modern libraries. After moving to RAM-based execution, the original Purcell and Gauss-Seidel CSOR methods were tested again. In addition to these new methods, three new methods were tested in hopes of achieving improved solution time: a conjugate gradient method, a stabilized biconjugate gradient method, and an Intel MKL-based direct LU factorization solve. Each method comes with its pros and cons regarding solution accuracy and runtime.

To minimize the runtime, it was imperative to test many different test cases, hypothesizing that the increased computational overhead of allocating memory for a method such Bi-CGSTAB may become relatively worth it as the grids become dense. Conversely, the reduced overhead of Miranda’s original implementation of CSOR may prove worthwhile for small problems.

C. CSOR Method

Miranda’s original CSOR method was reasonably quick, even on the “slow” compile of *VORLAX*. However, without altering the integrity or functionality of the original subroutine, the in-memory compile lead to the performance improvements, seen in Table 2. It shows that the 2020 version of *VORLAX* runs significantly faster in all configurations than the classic version. As the complexity of the linear system decreases (i.e. as N becomes smaller), the computational overhead of the preprocessing, despite its incredibly fast nature, becomes the dominant factor in the compute “wall time”. Furthermore, it is clear that as the complexity increases, the relative gains in the in-memory solver become even better. The Gauss CSOR method, as implemented by Miranda, relies heavily on read/write cycles to the tape deck – whether that be the scratch files on the hard drive in the case of classic *VORLAX* or the arrays in *VORLAX* 2020. However, because the RAM-storage method is so much faster than

Table 1 Gauss CSOR Speed Comparison (s)

	NVOR = 25, RNCV = 10	NVOR = 50, RNCV = 20	NVOR = 100, RNCV = 40
VORLAX 2014	0.078130	0.984380	20.296880
VORLAX 2020	0.046880	0.328130	5.718750
% Improvement:	**40.0%**	**66.7%**	**71.8%**

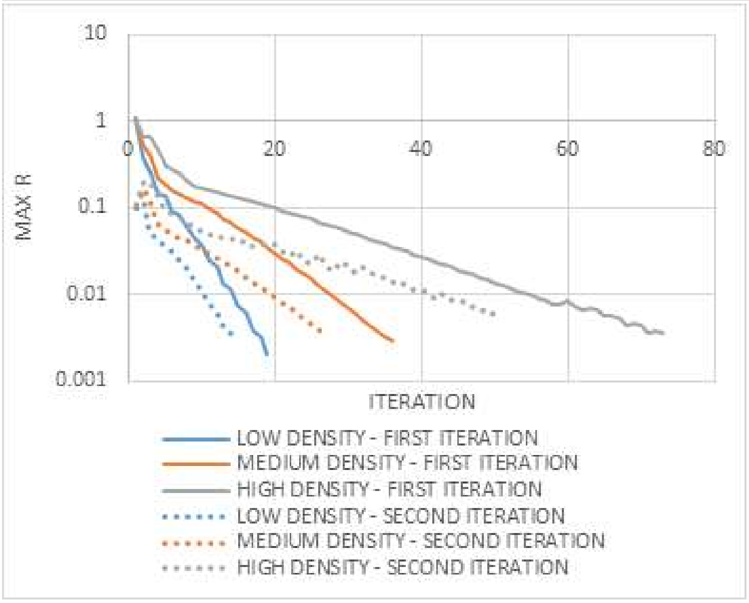

FIGURE 8 CSOR Residual Behavior

the hard drive method, the time savings are more pronounced as the size of the arrays, and thus the number of read/write cycles, increases.

The version present in *VORLAX* 2020 has one minor change compared to the original. In the original version, the initial x_0 "guess" array was populated with only zeros. This has been updated to utilize the previous answer as the initial guess in each call of the solver within each distinct Mach number. In the case of the very first run, the array is initialized to have a single nonzero value in the first slot, which typically improves iterative performance. It should be noted that not all subsequent iterations benefit from this strategy, hence the initialization technique being only for each angle of attack. When iterating between Mach numbers, the system of equations changes to be sufficiently different as to cause the program to run slower when utilizing the previous Mach number's solution [10]. This can be attributed to the change in the Prandtl-Glauert coefficient applied to the perturbation velocities causing the overall system to vary significantly between Mach numbers. It becomes apparent when plotting the residual convergence (FIGURE 8) for the second call of the solver, in which the second initial residual is much lower than the first, but the two methods retain the same order of convergence. The effectiveness of this approach comes from the order in which *VORLAX* calculates the influence coefficients. The code includes a double nested FOR-loop, looping through angles of attack within the Mach number loop. Thus, at a constant Mach number, the influence coefficients are not drastically different between the angles of attack, especially in a typical application where the angle of attack is perturbed in increments of 1-2 degrees. Because of the effectiveness when applied to the CSOR method, the same method was applied within the other tested solvers.

D. Purcell's Vector Method

Table 2 Purcell's Vector Method Speed Comparison (s)

	NVOR = 25, RNCV = 10	NVOR = 50, RNCV = 20	NVOR = 100, RNCV = 40
VORLAX 2014	1.390630	40.218750	1370.531000
VORLAX 2020	0.156250	29.046880	N/A
% Improvement:	**88.8%**	**27.8%**	**N/A**

While it would first appear that Purcell's method would be favorable, being a direct-solve method, that is quite simply not the case. In practice, the Purcell method is impractically slow to run, and thus does not fit with the fast nature of *VORLAX*. We see in Table 3 that the run took 22.8 minutes to compute in the dense grid case, relative to the 20.297 second runtime of the CSOR method also running on the classic *VORLAX*. Furthermore, memory limitations in the *VORLAX* 2020 32-bit compile do not allow for a proper implementation of Purcell's method for the largest of linear systems. However, it is apparent that even by the NVOR = 50, RNCV = 20 case, the Purcell solver is so much slower than the CSOR method that is not worth using. The speed comparisons for this method were included because they provide insight into the history of *VORLAX* and help characterize upper and lower "goal" speeds for the new proposed solvers.

Furthermore, while execution time is not the only defining factor of the quality of the method, the direct solve results did not provide answers that were more accurate to any meaningful figure – the differences were typically constrained to the 4th-5th decimal place for figures such as the lift and drag coefficients. In the grand scheme of things, this is largely insignificant. It is understood that *VORLAX* operates under a set group of assumptions as it is, and there is a point where these small "improvements" are overshadowed by these assumptions. This is not to say that *VORLAX* is inaccurate, as it does a remarkable job giving big picture information about a configuration and its aerodynamic efficiency, however it is also understood that the program will not provide information that is accurate to five significant digits.

E. Conjugate Gradient Method

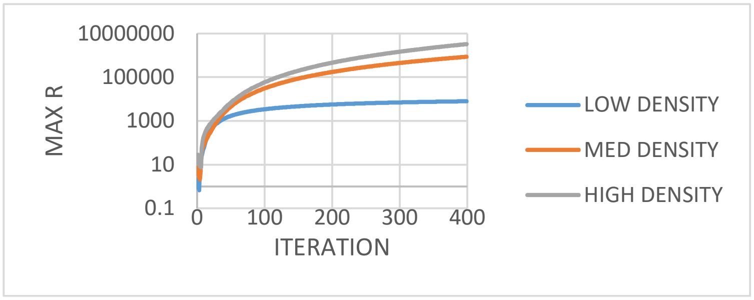

FIGURE 9 Conjugate Gradients Residual Behavior

The Conjugate Gradients (CG) Method is an iterative method in the family of Krylov Subspace Methods. The version experimentally implemented into *VORLAX* comes from Henk van der Vorst's book, Iterative Krylov Methods for Large Linear Systems. The implementation of the CG method was mostly experimental in nature, as it utilizes the same algorithmic structure necessary to construct the more useful and reliable variants of the method for later testing. In part, the reason that little was expected of this method was due to the fact that the CG method is slow, computationally expensive, and susceptible to convergence errors when applied to an unsymmetric system [11]. Thus, given that *VORLAX* works by solving a very unsymmetric and dense system, the outlook was meek.

Convergence testing confirmed that the CG method on its own was unsuitable as a solver within *VORLAX*. During trials, this method was shown to be incredibly unreliable and divergent in each of the test cases, as seen in FIGURE 9. This behavior was expected, though not necessarily guaranteed. The conjugate gradients method assumes that the matrix is sparse, positive-definite, and symmetric; and the system constructed by *VORLAX* does not meet this criteria. Thus, the method was assumed to be unreliable at best. Additionally, because the behavior was almost exclusively divergent, *VORLAX* would iterate until its iteration limit (ITRMAX = 399). This could be remedied by implementing a method of advancing divergent cases without progressing through the full method, however this was not the focus of the study.

F. Stabilized Bi-Conjugate Gradients Method

The Stabilized Bi-Conjugate Gradient Method (Bi-CGSTAB) was another method presented by van der Vorst [12]. This method was appealing due to its reliable nature for solving a system of unsymmetrical equations. While more complex than the CG method, it was promising due to its reliability. However, when testing the convergence behavior, the results were less than impressive. The solution converged in our trial runs, however it was very slow, seen in Table 4.

Table 3 Bi-CGSTAB Runtime Comparison

Grid Density	Runtime w/o Preconditioning (s)	Runtime w/ Jacobi Preconditioning (s)
25x10	0.32813	0.625
50x20	4.03125	5.64063
100x40	66.1875	49.53125

In an attempt to accelerate the convergence of the method, an alternate subroutine was developed utilizing the application of a simple Jacobi preconditioner. While there are plenty of more effective preconditioners, the ease of inverting the diagonal matrix lead to it being a good method for implementation within *VORLAX* [13]. The results of this implementation were very interesting, looking at the wall time for each case, we saw that the two lower grid densities saw an *increase* in runtime, while the largest density saw a drastic *decrease* in the runtime. For better insight, we turn to the convergence behavior of each method.

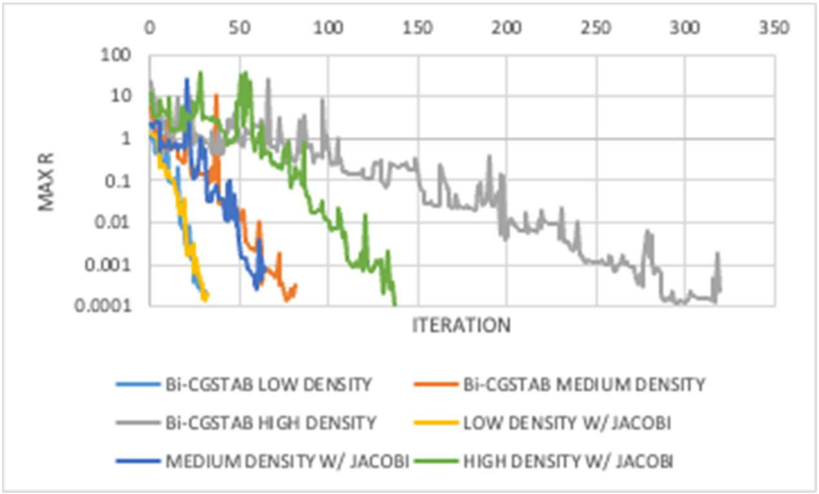

FIGURE 10 - Bi-Conjugate Gradient Method Residual Behavior

We see in FIGURE 10 that the difference between the non-preconditioned and preconditioned trials is highly dependent on the size of the problem at hand. For the two lowest densities, we see that the convergence is largely the same, with the medium density showing slightly improved convergence, but only by 16 iterations. Conversely, the high-density case shows a reduction of 183 iterations with the application of the preconditioner. When we tie these observations into the runtime of the program, it all begins to make more sense. Essentially, due to the simplistic nature of *VORLAX* and its inherently small grid sizes (remnants of the fact that it was developed in a manner bound by the technological constraints of the IBM 360), there exists a point where the computational resources necessary to compute the preconditioner outweighs the cost to "brute force" the solution without preconditioning. Thus, when there is only a small improvement to be had, it is not worthwhile to compute and apply this preconditioner, because it is mostly "getting in the way" of the method while calculating the solution.

G. Intel MKL Direct Solve via LU Factorization

Finally, in an attempt to search for a viable "pre-packed" library solution, an implementation of the generalized solver available through Intel's Math Kernel Library was included. Using the Intel Visual Fortran Compiler, it was as simple as checking a box to utilize the features, and the results were promising. Of all of the attempted solvers, this was the only one that was even close to the CSOR method in terms of runtime commitment, seen in Table 5. In part, this is due to the parallelization methods done in the "black box" Intel library. Because the test machine was running with an Intel Core i9-9900k, it is not unlikely that the library was particularly optimized for Intel-based architectures.

Table 4 Intel MKL Runtime

Grid Density	Intel MKL Wall Time (s)
25x10	0.04688
50x20	0.39063
100x40	8.95313

Unlike the iterative methods, the MKL implementation completed the computation via a traditional *LU*-factorization technique. Thus, there is no "converge criteria" that is possible to compare with the earlier presented methods. While the method proved efficient, adding the library requires the distribution of several library files that greatly increase the size of the *VORLAX* package, which is undesirable given that *VORLAX* is often given to students as a learning tool. Furthermore, the proprietary nature of the library is undesirable as it comes with no guarantee that the performance will remain the same across all CPU architectures, both older Intel processors and modern AMD processors. Thus, this solver was deemed unfit for a mass-distributable version of *VORLAX*. However, it does serve as a good tool for a curated "Research Build" of the software, where it becomes possible to keep the code within a controlled environment.

VI. Other Expensive Subroutines

By inspecting the *VORLAX* source and timing specific calls to subroutines, it became apparent that the solving routine was not necessarily the only subroutine taking up significant runtime. There exists a routine called "MATRX", whose purpose is to generate the grid of control points, complete with all of the normal and axial wash coefficients, as well as the influence coefficients for each vortex at each control point. Thus, for sufficiently large problem sizes, this routine will need to calculate 5000 influence coefficients for each of 5000 control points, a task which takes up a considerable amount of time. By nature, the vortex-lattice method relies on a dense system of equations, and as a result there is little that can be done to lessen the impact of the MATRX subroutine.

To better understand the tradeoff of MATRX versus the solving routine, *VORLAX* was altered slightly to produce timings of each call to the two routines in a scratch file, and the results made it clear that the MATRX routine was more expensive per call than the solving subroutine. While at first glance this places the blame on the MATRX subroutine for being expensive, it is not the entire story. The solving routine was called for each Mach number and angle of attack combination, while MATRX was only called for each Mach number. Thus, sometimes the solver took up the majority of the time, while other times the MATRX routine took the majority of the time – dependent on the usage scenario.

There are three common usage cases for *VORLAX.* The first is using the program to model a configuration to run at a single angle of attack and a single Mach number – commonly seen in advanced wing design applications where the flight condition of interest is specifically known. The second is to model a body at a single Mach number and multiple angles of attack in order to generate lift and drag polar diagrams, which is useful for specific flight phases, such as takeoff. The third common case is to run a small number of Mach numbers, usually about three, and multiple angles of attack, which is used to generate data regarding the stability and control of an airframe. Each of these configurations was tested with the modified "stopwatch" version of *VORLAX* in order to quantify the amount of time – both absolute and broken down by subroutine, that was spent running both the MATRX and solve routines.

These tests were completed using a Visual Basic script to automate the input file generation and run timings. Each Mach number and angle of attack configuration was tested for grid densities ranging from 250 total control points to

5000 total control points – the program maximum. Because this was primarily a test of the numerical methods – the time which was necessary to construct the linear system and solve it, the exact spacing of the grid points was not of much concern. During the testing it was noticed that as the time to run became incredibly small, the accuracy of the timing function became inaccurate due to the sheer speed of program call – in some cases reporting 0.0000s for the calls. This is obviously incorrect; however, it serves as a testament to just how quickly the basic cases run.

The first tests were run for the single Mach and single angle of attack configuration. Without much surprise, the runtime grew exponentially overall as the number of grid points increased (recall that the system is [NxN]), seen in Figure 11. This was expected due to the nature of the vortex-lattice method, wherein each point is fundamentally dependent on each other point in the model.

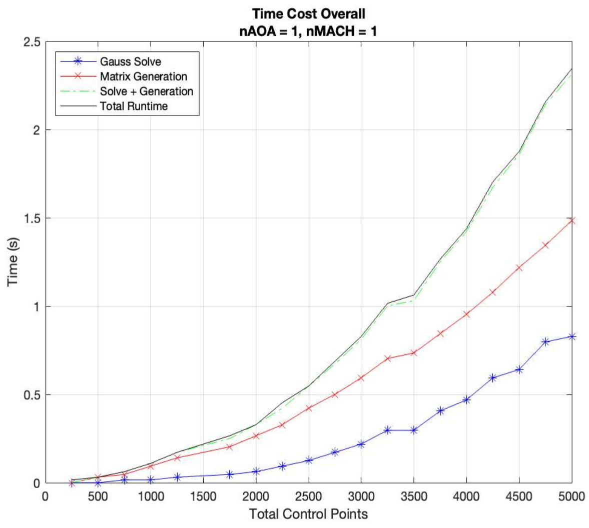

FIGURE 11 – Overall Time Cost for One Mach Number and One Angle of Attack

Figure 11 is telling, as it becomes clear that the MATRX subroutine is not a negligible contributor to the overall runtime! Understanding that there are some minor errors in the timing because of the intrinsic timing function, the overall trends are still very obvious. For each of the grid densities, both the MATRX and GAUSS subroutines are overwhelmingly the driving force of the total runtime – accounting for over 90% of the time allocated to each program call. Thus, it is clear that the two most expensive calls account for almost the entirety of the runtime. While this run shows that the MATRX call is more expensive, the story changes when running more angles of attack relative to Mach numbers.

While the proportions may seem largely different, the timing seen in Figure 12 shows that the two most expensive subroutines still account for most of the runtime. However, it is now the linear system solver driving the total runtime instead of the system generation. The MATRX subroutine, while considerably smaller in overhead than the GAUSS subroutine, is not negligible, and can still benefit from improvements.

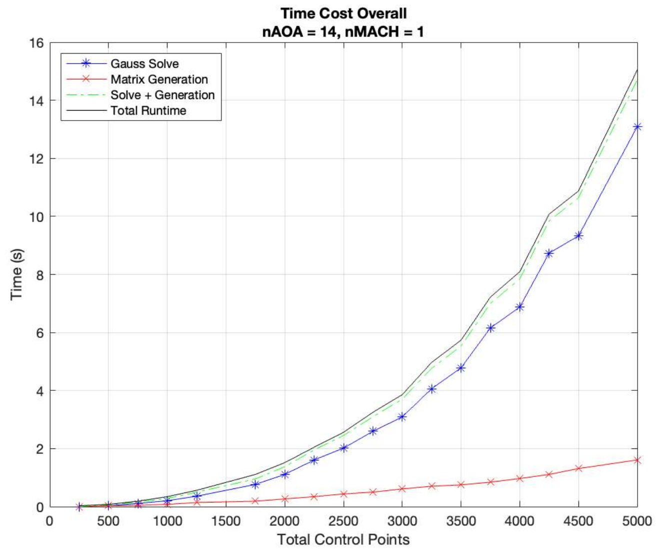

FIGURE 12 – Overall Time Cost for One Mach Number and Fourteen Angles of Attack

The final configuration was that commonly seen in stability and control applications, where the user will define a system with 3 freestream Mach numbers and 14 angles of attack. The array of pitching angles allows the user to compute the stability derivatives and having multiple Mach numbers will allow the user to interpolate between them to approximate performance derivatives in multiple flight conditions, such as takeoff, climb, cruise, descent, and landing. While this may seem like a lot, it is worth remembering that initializing the command window to run the *VORLAX* program on Windows can often amount to fairly considerable time contributions, thus it is advantageous to configure all of the desired combinations in a single file.

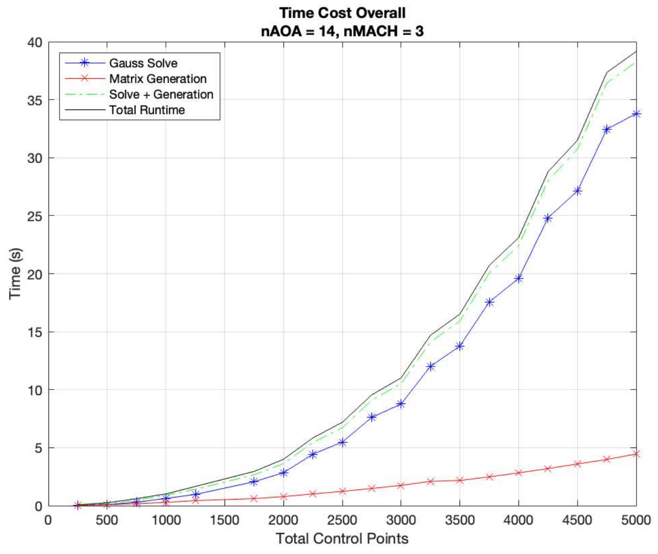

FIGURE 13 – Overall Time Cost for Three Mach Numbers and Fourteen Angles of Attack

Figure 13 shows the overall runtime of this test, once again demonstrating a parabolic growth in runtime as the number of control points increases linearly. This run took nearly 40 seconds to compute in its most expensive configurations, and while this is remarkably fast by modern CFD standards, it begins to add up depending on the number of designs necessary to test. Commonly, the stability and control applications of *VORLAX* will involve the program running five different files, one of which is in a neutral configuration, one at a sideslip angle, and three with control surface deflections. Thus, for a "common" application, this may amount to a considerable amount of time. Previously, each of the stability and control files with ~2,000 control points would take ~*O*(120) seconds to run (depending on the case), meaning that the whole stability and control case could take up to 10 minutes to run! Thus, every time a tail surface changed even slightly, or perhaps the aileron surface area increased incrementally, there would be nearly 10 minutes wasted to downtime waiting for the program to run. This was largely the inspiration of this work, and when considering this portion on timing, it is clear how helpful these improvements are in real-world applications.

Figure 13 shows again that while the overall runtime of the program in this configuration has increased, the proportional contributions from each subroutine remain more or less the same. Thus, for two of the common configurations, the amount of time spent solving the system is large relative to the amount of time constructing each system, except in the case of the single panel where the matrix construction was the more expensive operation. However, it is necessary to consider the overall runtime of the configurations tested. While the single panel shows a larger dependence on the MATRX subroutine, the total runtime is often significantly shorter. For instance, a very detailed wing design will often have ~100 spanwise control points and ~20 chordwise points, giving a total runtime on the order of 0.5 seconds, which leaves little to gain in terms of optimizations. Conversely, a complex stability and control model of a full aircraft may have nearly 4,000-5,000 total control points (depending on the configuration and number of panels), which will take ~30 seconds to run, but also needs to be run five times, giving a total "wall time" of each case of ~150 seconds. Thus, the potential to save time falls firmly in the solver subroutine, as even a 0.03% reduction in runtime would amount to more time than the single configuration takes altogether.

There are some nuances to the program that currently limit the program. Being limited to a 32-bit compile for compatibility reasons provides a maximum to the amount of RAM usable by the program. Furthermore, because the program ran using tape drives, everything is vector-based, when matrices could potentially allow for more powerful solving libraries. However, there is a cost to converting the vector-space terms to matrices, and then an additional cost to converting back to vectors. There are some changes that may be experimented with to decrease the runtime, however each change risks altering the dependability of the code or changing certain notational norms carried from the FORTRAN 66 framework, and as such any modifications should be made with caution.

VII.Benchmark Comparisons

Table 6 Runtime of Each Solver

Runtime (s)	**LOW DENSITY**		**MEDIUM DENSITY**		**HIGH DENSITY**	
	V 2014	*V 2020*	*V 2014*	*V 2020*	*V 2014*	*V 2020*
Gauss	0.07813	0.04688	0.98438	0.32813	20.29688	5.71875
BiCGSTAB	--	0.32813	--	4.03125	--	66.18750
BiCGSTAB + Jacobi	--	0.62500	--	5.64063	--	49.53125
INTEL	--	0.04688	--	0.39063	--	8.95313
PURCELL	1.39063	0.15625	40.21875	29.04688	1370.53100	--
CG	--	2.23438	--	10.48438	--	44.68750

Table 6 shows a summary view of all of the different runtime requirements of the different solvers in these tests. When comparing the wall time of each of the solving techniques, it becomes immediately apparent that the CSOR method present in the original *VORLAX* program is incredibly efficient. The CSOR method is universally faster than any of the other proposed methods, with only the Intel MKL solver proving to be of comparable performance. Looking at the performance increases to the CSOR method, we see drastic performance increases over the 2014 version, particularly when the solution of interest becomes large. We see comparable increases with the Purcell Vector Method; however, the method remains very slow in comparison to the alternatives. Thus, the method is not viable for this application.

None of the Krylov-subspace methods performed favorably. With the obvious convergence issues of the conjugate gradients method, the other methods were simply too slow to serve as a viable solver within *VORLAX*. The performance of the Krylov methods could be improved by more aggressive preconditioners, however that comes at the cost of implementing routines to calculate the preconditioners, which are often more expensive than the computation itself. Thus, none of the tested methods appear to be suitable for usage in *VORLAX* in their current state, nor do they appear to have 32-bit compatible alterations that would make them more attractive.

Finally, the Intel-based solver performed only slightly slower than the CSOR method, despite using the more inefficient method. This is largely due to the multithreaded capabilities of the Intel code, something which the original *VORLAX* code lacks. However, despite the Intel solver being close in speed to the CSOR solver, the results returned by the program are the same to four decimal place precision, and thus the Intel solver does not provide appreciable improvements to the accuracy of the code. Coupled with the increased complexity of distribution, the Intel solver is not viable to include within the standard *VORLAX* package given to students for educational purposes.

VIII. Future Areas for Experimentation

At the time of writing, *VORLAX* remains a strictly serial program, utilizing only a single core to perform calculations. There may be merit in modifying *VORLAX* in order for it to take advantage of multicore systems. Given that the vast majority of modern computers have multiple cores, it is a rather obvious idea to modify the code to take advantage of these cores! *VORLAX* already runs very quickly, but if the workload could be better distributed among a computer's resources, it would be helpful for the complex cases. This paper has shown deeply the single-core performance of the Gaussian CSOR method, however one drawback of the program is that the calls to the solver are within a double-nested FOR-loop. This causes the program to have to "wait" for the solver to finish its run before continuing to the next solve. If *VORLAX* was able to effectively make use of one of the numerous distributed computing libraries (i.e. Open MPI [14]) then its performance could see massive improvements, particularly within workflows that integrate thousands of runs of *VORLAX*.

Another possibility is to consider a version of *VORLAX* compiled for ARM-based system architectures. Currently, the ARM-based M1 chip designed by Apple has boasted fantastic performance figures while being incredibly power efficient [15]. With these performance figures, there is a chance that other manufacturers may follow suit, in which case it is important that *VORLAX* be cross-platform. Because the M1 chip often outperforms typical x64 chips for CPU-intensive tasks such as compiling programs, there may be further performance improvements for *VORLAX*. At the time of writing, *VORLAX* works best with Intel compilers. Attempts have been made to compile the program with other compilers, however the efforts are generally futile due to various errors that arise because of the legacy FORTRAN programming.

Some testing was completed within the timings which demonstrated a fair sensitivity to the initial guess of the CSOR solver. This indicates that there may be further improvements to be made by introducing a form of a multigrid solver, using coarse grid circulation solutions to accelerate dense grid solutions. This would take advantage of the exponential nature of the solver cost, potentially saving time overall. It may also be worth working with the array structures of the program to see if modern FORTRAN compilers can run more efficient on an array-based system. One hurdle with this is that *VORLAX* operates in a very vector-focused manner, and all of the arrays and loops are currently geared for single dimension iterators, so this would not be a trivial change.

Finally, there may be merit in porting *VORLAX* to a different programming language altogether, from the ground up. In its current state, the program is *remarkably* efficient. However, its source code is largely written in FORTRAN IV with occasional use of FORTRAN 77 standards. The source code compiles easily under Intel Visual FORTRAN with certain compatibility flags set, but throws many errors under gfortran and other modern “FORTRAN-TO-C” “pre-processor” type compilers. [16] This presents a looming long-term maintenance problem when old IBM and VAX backward-compatible FORTRAN compilers disappear from the marketplace. In the interest of futureproofing the program, it may be wise to rewrite it using either updated FORTRAN standards, thereby widening the pool of individuals who can meaningfully contribute to and maintain the source code.

IX.Conclusions

Luis R. Miranda’s talents as a programmer and aerospace engineer were no secret, and the methods he originally developed have withstood the test of time, working with incredible precision and speed, even in the face of more “modern” techniques. The CSOR method implemented early-on into *VORLAX* remains one of the best options for solving the small, dense systems of equations incredibly quickly. The memory commitment is minimal, lending itself to a small, fast 32-bit compile, and the results are just as accurate as more advanced solving methods. Thus, it appears that the CSOR method remains the best option to include in the “standard” version of *VORLAX*, turning to more complicated methods only in the niche cases where they are required.

Future plans for *VORLAX* often depend on the work at hand. Because the program returns accurate results, even with small grid sizes, there currently is no reason to further complicate the code, increasing its size and runtime, by switching to a 64-bit compile. However, if noticeable gains can be obtained using 64-bit libraries, it may become worthwhile for research purposes. Currently, the prospect of using GPU-accelerated libraries, such as those offered by NVIDIA, are being considered, along with libraries such as HYPRE. Overall, the main consideration at hand is the balancing act between the increased runtime of a more complex code relative to the improvements to accuracy, problem complexity, and/or more involved simulation models.

Acknowledgements

This revised version of *VORLAX* is currently in use in Professor Takahashi’s senior- and graduate-level courses beginning in the Fall 2020 semester. The final version released at AIAA AVIATION 2021.

This manuscript derives from work Mr. Souders performed in partial fulfillment of the degree requirements for obtaining his M.S. in Mechanical Engineering from Arizona State University. All design analysis on this unfunded project was completed at Arizona State University.

References

[1] Johnson, F. T., Tinoco, E. N., and Yu, N. J. "THIRTY YEARS OF DEVELOPMENT AND APPLICATION OF CFD AT BOEING COMMERCIAL AIRPLANES, SEATTLE." Computers & Fluids, Vol. 34, No. 10, Dec, 2005, pp. 1115–1151.

[2] Miranda, L. R., Elliot, R. D., and Baker, W. M. "A Generalized Vortex Lattice Method for Subsonic and Supersonic Flow Applications." NASA CR 2865, 1977.

[3] Melin, T. A. "Vortex Lattice MATLAB Implementation for Linear Aerodynamic Wing Applications". Master's Thesis, Royal Institute of Technology, Sweden, 2000.

[4] Küchemann, D. *The Aerodynamic Design of Aircraft.* AIAA 2012.

[5] Jensen, J., and Takahashi, T. "Wing Design Challenges Explained: A Study of the Finite Wing Effects of Camber, Thickness, and Twist." AIAA 2016-0781, 2016.

[6] Souders, T. J., and Takahashi, T. T. "VORLAX 2020: Benchmarking Examples of a Modernized Potential Flow Solver". AIAA 2021-2459, 2021.

[7] Oswald, W. B. "General Formulas and Charts for the Calculation of Airplane Performance." NACA TR-408, 1933.

[8] Purcell, E. W. "The Vector Method of Solving Simultaneous Linear Equations." Journal of Mathematics and Physics, Vol. 32, Nos. 1–4, 1953, pp. 180–183.

[9] Lan, C. E. "A Quasi-Vortex-Lattice Method in Thin Wing Theory." Journal of Aircraft, Vol. 11, No. 9, 1974, pp. 518--527.

[10] Souders, T. J., "Modernization of a Vortex-Lattice Method with Aircraft Design Applications", M.S. Thesis, Department of Mechanical Engineering, Arizona State University, Tempe, AZ, 2021.

[11] Bratkovich, A., and Marshall, F. J. "Iterative Techniques for the Solution of Large Linear Systems in Computational Aerodynamics." Journal of Aircraft, No. 12.2, 1975, pp. 116–118.

[12] van der Vorst, H. A. *Iterative Krylov Methods for Large Linear Systems*. Cambridge University Press, 2003.

[13] Shewchuk, J. R. *An Introduction to the Conjugate Gradient Method without the Agonizing Pain.* Carnegie-Mellon University. Department of Computer Science, 1994.

[14] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S., "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation." 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004 (pp. 97–104).

[15] "Apple unleashes M1," Apple Newsroom, November 2020. [https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/. Accessed 6/4/21.]

[16] "The Fortran compiler gfortran will not compile files," see https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96033 [accessed 6/24/21]