Development of a Rapid First-Order Differential Equation Solver for Stiff Systems

Lisa M. Kuhn\textsuperscript{1}, Steele Russell\textsuperscript{2}, Matthew Boudreaux\textsuperscript{1,2}, Jessica Gaboury\textsuperscript{1,2}, and Brandon Maulding\textsuperscript{2}

\textsuperscript{1}Department of Mathematics, Southeastern Louisiana University, Hammond, LA
\textsuperscript{2}Department of Computer Science, Southeastern Louisiana University, Hammond, LA
{lisa.kuhn, steele.russell, matthew.boudreaux-4, jessica.gaboury, brandon.maulding}@southeastern.edu

Abstract—Large stiff ordinary differential equation systems often require a significant amount of computation time due to numerical instabilities. It is the goal of this work to develop a fast solver which will enable the user to achieve quick results without having to leave the MATLAB environment. Two first order differential equation solvers are developed and compared with MATLAB’s \texttt{ode15s} solver. The CODES solver is an abbreviated version of \texttt{ode15s} modified to address the constant coefficient problems of interest. The RODES solver implements the CODES algorithm in C interfaced with MATLAB. The solvers are employed on four test problems: three ordinary differential equation models and one partial differential equation model. Numerical simulations are provided and the results are analyzed with respect to error, time, and space. All three solvers provide accurate approximations to the test problems. CODES and RODES outperform \texttt{ode15s} in speed. RODES significantly outperforms CODES.

Index Terms—stiff systems, ordinary differential equations, partial differential equations, MEx, MATLAB

I. INTRODUCTION

The authors seek to develop a rapid stiff system solver for ordinary differential equations (ODEs) that is interfaced with MATLAB. MATLAB was developed in the early 1970’s as a means to make high performance computer libraries written in FORTRAN more accessible. It is now widely used for scientific computing. Stiff equations often contain numerical instabilities, requiring extremely small step sizes, which results in a high computational cost. Solvers for these equations are one of the most common computational bottlenecks in mathematical modeling. Furthermore, the extremely small steps sizes required to solve large stiff systems often result in memory issues for the user. In [1] it was shown that C++ is significantly faster than MATLAB in processing speed. In [2] numerical integration functions implemented in C and executed in the MATLAB environment showed a reduction in computation time.

Fast first-order ODE solvers would be advantageous for many applications since higher order ODEs can be rewritten as a system of first order equations [3]. Furthermore, many partial differential equations (PDEs) can be rewritten as a system of ODEs using the Finite Element Method (FEM) [4]. Therefore, the development of an expedited solver for this class of systems applies to systems of ODEs, PDEs and PDE systems. Some applications include the simulation of mass-spring systems, beams, plates, and heat transfer. A few of these models can be found in [5], [6], [7], [8].

The fundamental goal of this project is to develop an initial first-order, linear differential equation solver for stiff systems which reduces computation time. Such a solver will enable users to achieve a solution quickly without having to leave the MATLAB environment, providing seamless prototyping, visualization, analysis, and optimization. The authors seek to develop a solver and compare the solver’s performance with MATLAB with respect to time, space, and error.

The outline of the paper is as follows. The numerical algorithms are described in Section II. The test models applied to the solvers are detailed in Section III. Section IV provides the numerical results of the models and performance comparisons with respect to time, space, and accuracy. Conclusions and directions for future work are given in Section V.

II. METHODOLOGY

Although MATLAB has a suite of ODE solvers, the \texttt{ode15s} function, which exploits numerical differentiation formulas, is considered the reference standard due to its reputation for being a very efficient stiff system solver. Furthermore, Runge-Kutta algorithms, which are available in other MATLAB solvers, often do not perform well on stiff systems [9]. Computational solutions to selected problems using three solvers are compared: 1) solutions obtained via \texttt{ode15s}; 2) solutions obtained using the Concise Ordinary Differential Equation Solver (CODES); 3) solutions obtained using the Rapid Ordinary Differential Equation Solver (RODES).

A solution is sought for initial value problems of the form

\[ y'(t) = Ay(t), \quad y(t_0) = y_0, \]

where \( y'(t) = \frac{d}{dt} y(t) \), \( A \) is a constant coefficient \( n \times n \) matrix, and \( y(t_0) \) is the initial condition vector.

A. \texttt{ode15s} Solver

MATLAB’s \texttt{ode15s} function is the motivating algorithm. It uses numerical backward differentiation formulas with a maximum order of \( k = 5 \) using a quasi-constant step method. In addition to solving the constant coefficient problems of interest, \texttt{ode15s} can solve differential algebraic equations. Further details of the algorithm are available in MATLAB’s ODE suite [10].
B. CODES Solver

Some notable changes are made to ode15s. First, in order to accommodate the specific problems of interest (1) an initial working algorithm is obtained which removes any excess overhead from ode15s related to differential algebraic equations and variable step methods. Second, the max order of the difference is considered to be \( k = 1 \) in this work. Though this will obviously affect the maximum time step required to achieve a prescribed accuracy, this is done as a preliminary step due to the complexity of the algorithm. Higher order differences will be explored in future work. The inputs to the program are the initial condition vector, \( y_0 \), the starting and stopping time values, \([\text{tstart}, \text{tend}]\), and the constant \( n \times n \) matrix \( A \). The outputs are the solution vector, \( y_{soln} \), and the time values \( t_{soln} \). The algorithm is implemented in MATLAB code. Since this is considered a concise form of ode15s it is hereinafter referred to as CODES.

First a parameter \( \psi \) is computed where

\[
\psi = \text{dif}(:, 1) \times \frac{1}{1 - \kappa}, \quad (2)
\]

where \( h \) is the current step size, \( y \) is the current solution, \( \kappa = -0.185 \) is the numerical differentiation coefficient for first order differences from [10]. Here \( \text{dif} \) is the difference matrix of size \( n \times (k + 1) \) whose first column is initialized to \( h \times A \times y \). The value of \( \psi \) is fixed throughout the computation of the solution at the next time step. The predicted solution at the next time step is

\[
\begin{align*}
t_{new} &= t + h, \\
y_{new} &= y + \text{sum}(\text{dif}(:, 1), 2).
\end{align*}
\]

Consider

\[
(I - \frac{h}{(1 - \kappa)}) A \Delta^{(i)} = \frac{h}{(1 - \kappa)} A y_{new} - (\psi + \text{difkp1}),
\]

where \( I \) is the \( n \times n \) identity matrix, \( \text{difkp1} \) is the difference between the predicted and final value of \( y_{new} \) (initially zero), and \( J \) is the Jacobian of the right hand side of Equation (1). Note the Jacobian is computed using MATLAB’s \text{numjac} \ function and remains constant throughout the algorithm. Equation (4) is solved for \( \Delta \). A maximum of four iterations of Newton’s Method are applied on (4) until \( \Delta \approx 0 \). If convergence does not occur within four iterations, the algorithm reduces the step size by \( \frac{1}{2} \). The solution to (4) is obtained using MATLAB’s matrix \text{divide} \ command.

Finally, the difference vectors and solution \( y \) are updated and the algorithm is iterated until the final time step \( \text{tend} \).

\[
\begin{align*}
\text{difkp1} &= \text{difkp1} + \text{del}; \\
y_{new} &= y_{new} + \text{del};
\end{align*}
\]

C. RODES Solver

The CODES algorithm is implemented in C and interfaced with MATLAB using the MATLAB Executables (MEx) framework. This is referred to as RODES. The basic structure of a MEx function involves two components: 1) the gateway function providing an interface between MATLAB and C/C++; and 2) the computational function which actually implements the desired algorithm. The gateway function translates the input parameters from MATLAB datatypes to C/C++ datatypes and output parameters from C/C++ to MATLAB. One issue that complicates this process is the use of single pointers in MATLAB for matrices while matrices are more naturally handled with double pointers in C/C++. Another complication is MATLAB’s use of column major arrays (as does FORTRAN) while C/C++ uses row major.

The Jacobian is constant. It is computed using MATLAB’s \text{numjac} \ function and then passed to RODES where the remainder of the algorithm is implemented. One modification from the CODES algorithm is that RODES solves (4) in C using the Jacobi Iterative Method for matrices [11]. It solves the linear system (4) which is of the form

\[
M \Delta = y,
\]

where

\[
M = \left( I - \frac{h}{(1 - \kappa)} \right) J,
\]

\[
y = \frac{h}{(1 - \kappa)} A y_{new} - (\psi + \text{difkp1}).
\]

The initial guess is assumed to be \( \Delta^{(0)} = 0 \). Matrix \( M \) is split into its diagonal and off-diagonal parts,

\[
M = \begin{bmatrix}
m_{11} & m_{12} & \cdots & m_{1n} \\
m_{21} & m_{22} & \cdots & m_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
m_{n1} & m_{n2} & \cdots & m_{nn}
\end{bmatrix}
\]

\[
= \begin{bmatrix}
m_{11} & 0 & \cdots & 0 \\
0 & m_{22} & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & m_{nn}
\end{bmatrix}
\]

\[
- \begin{bmatrix}
0 & 0 & \cdots & 0 \\
-m_{21} & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
-m_{n1} & -m_{n2} & \cdots & 0
\end{bmatrix}
\]

\[
- \begin{bmatrix}
0 & -m_{12} & \cdots & -m_{1n} \\
0 & 0 & \cdots & -m_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0
\end{bmatrix} = D - L - U.
\]
Then (6) becomes

\[ (D - L - U)\Delta = y, \]  

which can be written in the form

\[ \Delta^{(k)} = D^{-1}(L + U)\Delta^{(k-1)} + D^{-1}y. \]  

Since D’s only nonzero elements appear on the diagonal, its inverse is

\[ D^{-1} = \begin{bmatrix} 1/m_{11} & 0 & \cdots & 0 \\ 0 & 1/m_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/m_{nn} \end{bmatrix}, \]

and no matrix inversion algorithm is needed. This algorithm is ideal for the solver since the left hand side of (4) does not have zeros on the diagonal, and the Jacobi Iterative Method requires that zeros not appear on the diagonal. The method is iterated the absolute error with respect to the infinity norm meets the tolerance, to1 = 10^{-10},

\[ \left\| \Delta^{(k)} - \Delta^{(k-1)} \right\|_\infty < to1, \]

where

\[ \left\| \Delta^{(k)} - \Delta^{(k-1)} \right\|_\infty = \max \left\{ \left\| \Delta_1^{(k)} - \Delta_1^{(k-1)} \right\|, \ldots, \left\| \Delta_n^{(k)} - \Delta_n^{(k-1)} \right\| \right\}. \]

III. TEST PROBLEMS

Four test problems are applied to the ode15s, CODES, and RODES solvers. The first three problems are first, second, and fourth order ODEs. It is important to note that though these problems are not stiff and can be solved quickly without requiring much computation time, they are considered here as initial test problems because an exact solution exists for them. Comparison of solver outputs to an exact solution will build confidence in the output from the developed solvers, enabling testing of accuracy, latency, and space consumption. The fourth test problem is a partial differential equation (PDE) which has been reduced to a stiff system of ordinary differential equations using the Finite Element Method. Since it does not have an exact solution, results are compared with a solution obtained via Separation of Variables [12].

1) Test Problem 1: The first order problem is

\[ y'(t) = Ay(t), \quad y(0) = 1, \quad t \in [0, 5], \]

where A = 1. Note this requires a scalar input into the algorithm. Equation (13) has the exact solution \( y(t) = e^t \).

2) Test Problem 2: The second order test problem is

\[ y''(t) = y(t), \quad y(0) = 1, \quad y'(0) = 0, \quad t \in [0, 5], \]

which has the exact solution \( y(t) = \frac{1}{2}e^{-t} (1 + e^{2t}) \). Equation (14) must be rewritten as a first order system prior to being passed to the solver. The first order system is of the form

\[ x'(t) = Ax(t), \quad x(0) = [1 \ 0]^T. \]

Note (16) requires the input to be a multidimensional array of size \( n \times n \) where \( n = 2 \).

3) Test Problem 3: The fourth order test problem is

\[ y''''(t) = \frac{1}{16} y(t), \]

\[ y(0) = y''(0) = y''''(0) = 0, \quad y'(0) = -1, \quad t \in [0, 5], \]

which has the exact solution \( y(t) = -\sin \left( \frac{t}{2} \right) - \sinh \left( \frac{t}{2} \right) \). Equation (17) is written as a first order system of the form

\[ x'(t) = Ax(t), \quad x(0) = [0 -1 0]^T, \]

where

\[ x(t) = \begin{bmatrix} y(t) \\ y'(t) \\ y''(t) \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1/16 & 0 & 0 \end{bmatrix}. \]

Note (19) requires the input to be a multidimensional array of size \( n \times n \) where \( n = 4 \).

4) Test Problem 4: Here an Euler-Bernoulli beam partial differential equation (PDE) is considered with clamped-free ends. The equation is

\[ y_{tt}(t, x) + y_{xxxx}(t, x) = 0, \]

where \( t > 0, \ x \in (0, L) \). The essential boundary conditions, which represent zero displacement and slope at \( x = 0 \), are

\[ y(t, 0) = y_x(t, 0) = 0, \]

and the natural boundary conditions, which represent zero bending moment and shear force at \( x = L \), are

\[ y_{xx}(t, L) = y_{xxx}(t, L) = 0. \]

The initial conditions are

\[ y(x, 0) = -0.5x \quad \text{and} \quad y_t(x, 0) = 0. \]

The problem is rewritten as a system of stiff ODEs prior to being passed to the solver using a Galerkin finite element approximation. Additional details on the Finite Element Method can be found in [4], [13]. To obtain the weak formulation, multiply by a test function \( \phi(x) \in V \subset S = H^2[0, L] \), where \( H^2 \) denotes the Hilbert space with at most two derivatives, and integrate over the domain to obtain

\[ \int_0^L y_{tt}(t, x)\phi(x)dx + \int_0^L y_{xxxx}(t, x)\phi(x)dx = 0 \]

for all \( \phi(x) \in V = \{ \phi(\cdot) \in S : \phi(0) = \phi'(0) = 0 \} \). Next integration by parts is applied twice to the second integral in (24) and boundary conditions are applied,

\[ \int_0^L y_{tt}(t, x)\phi(x)dx + \int_0^L y_{xx}(t, x)\phi''(x)dx. \]
A basis vector $B^N_i(x)$, $i = 1, 2, \ldots, N$ is chosen for the approximating space $V^N \subset V$, where $N$ is the number of basis functions used in the approximation. Cubic $B$-splines are used to approximate the beam displacements. Since the $B$-splines do not naturally satisfy the essential boundary conditions (21), a modified cubic $B$-spline basis is employed as described in [14]. The state is

$$y(t, x) \approx \sum_{i=1}^{N} \alpha_i^N(t)B_i(x).$$

(26)

Substituting this state approximation into (25) and letting the test functions range over the appropriate basis vectors results in

$$\int_0^L \left[ \sum_{i=1,j=1}^{N} \alpha_i^N(t)B_i(x)B_j(x)dx + \sum_{i=1,j=1}^{N} \alpha_i^N(t)B''_i(x)B''_j(x) \right] dx = 0.$$ 

(27)

Note $\dot{\alpha}(t) = \frac{\partial}{\partial t}\alpha(t)$, $\ddot{\alpha}(t) = \frac{\partial^2}{\partial t^2}\alpha(t)$. Equation (27) can be written as

$$M\ddot{\alpha}(t) + K\alpha(t) = 0,$$

(28)

with

$$M = \int_0^L B_i(x)B_j(x)dx,$$

(29)

$$K = \int_0^L B''_i(x)B''_j(x)dx = 0.$$ 

Finally, (28) can be written as a first order system of the form

$$\dot{\hat{x}}(t) = Ax(t),$$

(30)

where

$$x(t) = \begin{bmatrix} \alpha(t) \\ \dot{\alpha}(t) \end{bmatrix}, \quad A = \begin{bmatrix} 0 & I \\ -M^{-1}K & 0 \end{bmatrix}.$$ 

(31)

An exact solution does not exist for this model, but other numerical solutions to beam equations are widely available. Here a solution via separation of variables derived by Asmar in [12] is used to validate the outputs of the solvers. The Asmar solution to (20) is

$$y(t, x) = \sum_{n=1}^{\infty} [X_n(x) \cdot A_n \cos \alpha_n t],$$

(32)

where

$$X_n(x) = \cos \alpha_n x - \cosh \alpha_n x$$

$$= \frac{\cosh \alpha_n L + \cos \alpha_n L}{\sinh \alpha_n L + \sin \alpha_n L} \cdot (\sin \alpha_n x - \sinh \alpha_n x)$$

(33)

$$A_n = \frac{1}{\kappa_n} \int_0^L -0.5xX_n(x)dx,$$

$$\kappa_n = \int_0^L X_n^2(x)dx,$$

and the $\alpha_n$'s are the positive roots of

$$\cosh \alpha L \cos \alpha L = -1.$$ 

(34)

Note (20) requires the input to the solver to be a multidimensional array of size $2N \times 2N$, where $N$ is the number of basis functions used in the FEM approximation.

IV. RESULTS

The numerical results obtained for (13), (14), and (17) with the three solvers are plotted against the exact solutions and can be seen in Figure 1. Since an exact solution does not exist for the PDE model (20), a solution obtained via separation of variables derived by Asmar in [12] is plotted for the first ten partial sums of the series solution. This solution is plotted against the solutions from the three solvers to validate the outputs. In the simulations provided here a beam length of $L = 2$ is used. For (13) convergence for the CODES and RODES solvers was obtained with a maximum step size of $h = 0.03$. For (14) and (17) convergence for the CODES and RODES solvers was obtained for a maximum step size of $h = 0.02$. Results indicate all three solvers obtain very good approximations of the exact solutions. For (20) a convergent finite element solution was obtained for $N = 20$ elements, which results in an input of size $40 \times 40$ to the ODE solvers for the $A$ matrix in (31). A maximum step size of $h = 0.0001$ in the ODE solver was employed. Since (20) is a stiff system, it is not uncommon for solvers to have to take unreasonably small step sizes in order achieve convergence.

In order to obtain a more precise measurement of the accuracy of the approximations, a relative error is computed with respect to the infinity norm. If the exact solution is $\hat{x}$ and the approximate solution is $\hat{x}$, the error is computed as

$$\frac{||\hat{x} - x||_{\infty}}{||x||_{\infty}},$$

(35)

where $||x||_{\infty} = \max \{|x_1|, |x_2|, \ldots, |x_n|\}$. The errors are given in Table I. As expected, the ode15s solver achieves better accuracy since it exploits higher order differences.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Test Problem 1</th>
<th>Test Problem 2</th>
<th>Test Problem 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ode15s</td>
<td>$9.6 \times 10^{-4}$</td>
<td>$6.0 \times 10^{-5}$</td>
<td>$3.4 \times 10^{-4}$</td>
</tr>
<tr>
<td>CODES</td>
<td>$4.9 \times 10^{-4}$</td>
<td>$3.3 \times 10^{-4}$</td>
<td>$6.6 \times 10^{-4}$</td>
</tr>
<tr>
<td>RODES</td>
<td>$6.0 \times 10^{-4}$</td>
<td>$3.1 \times 10^{-4}$</td>
<td>$1.5 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

TABLE I

RELATIVE ERRORS

In Figure 2 the computation time for 50 runs of each solver is plotted. The CODES and RODES consistently perform faster than ode15s, with the RODES solver being the fastest. While the relative performance of the solvers remains consistent the difference is most pronounced when Problem 4 (20) is examined. This is particularly encouraging because Problem 4 represents the type of model for which the solvers are intended.
Since memory issues are not uncommon in mathematical software, it is important to ensure the CODES and RODES solvers are not requiring unrealistic amounts of memory. Memory consumption is shown for 10 runs of each solver in Figure 3. Initially, memory (in megabytes) was computed for 10 single runs of each solver. However, since the ODE models require very small amounts of memory, these results were very small in magnitude. In fact, it takes approximately 1100 calls to the RODES function for the scalar problem (13) to show any significant memory usage. Therefore, the memory usage for the ODE models (13), (14), (17) are given for a total of 100 calls to the solver for each run. The memory usage for RODES is still essentially zero for (13). For the PDE model (20) each run number represents one call to the solver. For all models, the memory usage for CODES and RODES is either less or comparable to ode15s. At times, RODES is significantly less.

V. CONCLUSIONS AND FUTURE WORK

A. Conclusions

A first order differential equation solver was developed for linear systems using MATLAB’s ode15s function as the motivating algorithm. First a version of ode15s modified to only consider first order, constant coefficient ODE systems is implemented in native MATLAB code. This is known as the CODES solver. The RODES solver implements the CODES algorithm in C using a Jacobi Iterative Method in lieu of a matrix divide within the algorithm. The RODES algorithm is then ported to MATLAB where it can then be called as a native function. Four sample problems were passed to each of the three solvers. Three of the test problems were ODEs ((13), (14), (17)) for which the solver solutions are plotted against the exact solutions in Figure 1. Relative errors with respect to the infinity norms were computed and are available in Table I. The fourth test problem was a PDE (20) which was plotted against a solution obtained via Separation of Variables [12].

All three solvers achieve very good approximations to the problems. The solvers were analyzed with respect to time and space. Both the CODES and RODES solvers were observed to outperform ode15s with respect to latency. Memory overhead was less or comparable to that needed to execute ode15s.

B. Future Work

Future work includes an extension of the current algorithm to a more sophisticated variable step method and the inclusion of a numerical method for computation of the Jacobian matrix in C for the RODES solver. Further work involves implementation of higher order differences, stability analysis, and an extension to nonlinear systems.
Fig. 2. Computation Time for 50 Runs of Each Solver: Equation (13) (top left), Equation (14) (top right), Equation (17) (bottom left), Equation (20) (bottom right).

ACKNOWLEDGMENT

The authors gratefully acknowledge reviewers’ comments.

REFERENCES

Fig. 3. Space Required for Each Solver: Equation (13) (top left), Equation (14) (bottom), Equation (17) (bottom left), Equation (20) (bottom right).