A Sparse Differential Algebraic Equation (DAE) and Stiff Ordinary Differential Equation (ODE) Solver in Maple

TAEJIN JANG, The University of Texas at Austin, Austin, Texas
MAITRI UPPALURI, The University of Texas at Austin, Austin, Texas
KYEONGJUN SEO, The University of Texas at Austin, Austin, Texas
VENKATASAILANATHAN RAMADESIGAN, Indian Institute of Technology, Bombay, India
VENKAT R. SUBRAMANIAN, The University of Texas at Austin, Austin, Texas

This paper implements efficient numerical methods in Maple to solve index-1 nonlinear Differential Algebraic Equations (DAEs) and stiff Ordinary Differential Equations (ODEs) systems. Single-step methods (like Trapezoid (TR), Implicit mid-point (IMP), Euler-Backward (EB), Radau IIA (Rad) methods, TRBDF2, TRX2) and Backward-Difference Formula (BDF) of order 2 are implemented with adaptive time-stepping methods in Maple to solve index-1 nonlinear DAEs. Maple's robust and efficient ability to search within a list/set is exploited to identify the sparsity pattern and automatically calculate the analytic Jacobian. The algorithm and implementation are robust and efficient for index-1 DAE problems and scale well for finite difference/finite element discretization of two-dimensional models with system size up to 10,000 nonlinear DAEs and solve the same in a few seconds. The computational efficiency of the proposed algorithm (provided as an open-access code) compares favorably with the commercial solver gPROMs, one of the most commonly used sparse DAE solvers in the industry.

Key Words and Phrases: Sparse Solver, Analytical Jacobian, index-1 DAEs.

Recommended Reference Format:
Taejin Jang, Maitri Uppaluri, Kyeonjun Seo, Venkatasailanathan Ramadesigan, and Venkat R. Subramanian. 2023. A Sparse Differential Algebraic Equation (DAE) and Stiff Ordinary Differential Equation (ODE) Solver in Maple. *Maple Trans*. 3, 4, Article 16701 (December 2023), 22 pages.
https://doi.org/10.5206/mtv3i4.16701

1 Introduction

Several physical phenomena are represented by partial differential equations (PDEs) equations that describe the changes in space and time. Today, solvers such as IDA (Hindmarsh et al., 2005) in SUNDIALS's implementation of the BDF, DASKR (Brown, Hindmarsh, & Petzold, 1994), Hairer's FORTRAN implementation of RADAU (Hairer & Wanner, 1999), MATLAB's ode15s (Vie & Miller, 1986), Cash's MEBDFDAE (Cash & Considine, 1992), Julia's wide range of in-house solvers (Rackauckas & Nie, 2017), Maple's inbuilt solvers are all good solvers for small scale index 1- DAEs. ("Maplesoft Website," n.d.)
There are two popular approaches for solving index-1 DAEs – (1) Backward and modified-backward difference methods (IDA, DASKR, DASSL (Petzold, 1982), MEBDFDAE), and (2) Implicit Runge-Kutta methods (RADAU). Both approaches involve solving a system of nonlinear algebraic equations at every time step with a Newton-type approach which involves calling linear solvers until convergence within the Newton step.
Most DAE solvers provide options for different linear solvers (iterative, banded, and dense). Few solvers provide the option to call sparse linear solvers, particularly parallel sparse linear solvers. Solvers benefit from using analytical Jacobians and sparse solvers to improve computational speed and memory usage when solving a system of DAEs. Typically, the user provides the analytical Jacobian and the sparsity pattern.
gPROMs ("gPROMS ModelBuilder," n.d.) and DAEPACK ("DAEPACK," n.d.) are likely the only solvers that automatically calculate the Jacobian analytically in symbolic form and use the same with a sparse linear solver. This paper demonstrates that Maple is competitive in identifying the sparsity pattern and using the

analytic Jacobian. Including robust-time stepping methods and sparse linear solvers for Newton-Raphson iteration helps us arrive at a robust DAE solver that scales well for large systems.
The efficiency of a code strongly depends on multiple testing and tuning of adaptive-solver parameters, choice of Jacobian updates, and linear solvers, to name a few. We observe that the developed solver and its implementation are competitive compared to many existing solvers for the various cases we tested. The current implementation can be further optimized by compiling many steps to C language and tuning the solver parameters.
This paper and the code are built on the strengths of Maple and its ability to quickly search variables in a list/set, perform symbolic differentiation, and provide the analytic Jacobian. As the system size increases, it is necessary to use sparse linear algebra. The code developed can solve a wide range of index-1 DAEs with a minimal set of user inputs and is released under an MIT license without any restrictions. The following section describes various methods implemented using Maple for solving DAEs.

2 Algorithm Description

The objective of this work is to develop an algorithm to solve the Hessenberg index-1 DAEs of the form:

$$\begin{aligned} \frac{dy}{dt} &= f(y,z) \\ 0 &= g(y,z) \end{aligned} \tag{1}$$

where y is a vector of differential variables, and z is a vector of algebraic variables. f and g are assumed to be differentiable functions. Note that only autonomous systems are considered in this paper. If the time t explicitly occurs in a system, it can be converted to the form in equation (1) using a dummy variable y defined by $dy/dt = 1$.

Maple does not have a direct DAE solver. Its inbuilt DAE solver differentiates the algebraic constraints and converts the system to explicit ODEs. This restricts the usability to only a small number of DAEs. In addition, using a dense linear solver means that Maple's inbuilt DAE solver cannot handle a large number of DAEs (typically more than 200). The proposed solver enables the solution of greater than 100,000 DAEs in Maple. Seven different algorithms are considered in this paper.

2.1 Euler-Backward Method

The Euler-Backward (EB) method can be written for equation (1) as

$$\begin{aligned} y_1 &= y_0 + hf(y_1, z_1) \\ 0 &= g(y_1, z_1) \end{aligned}$$

(2)(Butcher, 2003)

where y_0 is the initial condition for y, h is the time step, y_1 and z_1 are differential and algebraic variables after the first step, $t = h$, Though the time step is a constant, later it will be considered variable using error control. Note that the initial condition for algebraic variables z_0 does not appear in (2), so any Backward-Difference method (BDF) method should ideally not need an exact initial condition for z, but code failures can happen if h is too large and initial guess is far off from the expected value when performing a Newton-Raphson iteration to solve equation (2). Typically, f and g are nonlinear functions of y and z. Equation (2) is a system of nonlinear equations of size $N_{ode} + N_{ae}$, where N_{ode} is the number of ODE variables, and N_{ae} is the number of algebraic variables.

2.2 Trapezoid Method

Trapezoid Method (TR), also called Crank Nicolson (CN) method, is used for solving semi-discretized systems of parabolic partial differential equations (PDEs) and will be referred to as CN in this paper and code. This method can be written as follows:

$$\begin{aligned} y_1 &= y_0 + \frac{h}{2} f(y_1, z_1) + \frac{h}{2} f(y_0, z_0) \\ 0 &= g(y_1, z_1) \end{aligned} \qquad \text{(3)(Iserles, 1996)}$$

For the CN method, there is a need to solve $g(y_0,z_0)$ to find a consistent initial condition at $t = 0$. The code is written such that z_0 is found for all the algorithms before marching in time, as small errors in z can affect the results in future times. This is particularly important for state estimation problems such as Battery-Management Systems (BMS).

2.3 Implicit Mid-Point-Trapezoid Method

The implicit mid-point-trapezoid method (IMPTRAP) is given by

$$\begin{aligned} y_1 &= y_0 + hf(y_{mid}, z_{mid}) \\ 0 &= g(y_{mid}, z_{mid}) \end{aligned} \qquad \text{(4) (Hairer \& Wanner, 1999)}$$

In the implicit mid-point method, residues are evaluated at the mid-point of the time interval $h/2$. Since the method is not stiffly accurate, there is a need to find z_1 by solving $g(y_1,z_1)$ after finding y_{mid} and z_{mid}. y_{mid} can be approximated using linear interpolation as $y_{mid} = (y_1+y_0)/2$. This does not introduce additional errors.

$$\begin{aligned} y_1 &= y_0 + hf\left(\frac{(y_0+y_1)}{2}, z_{mid}\right) \\ 0 &= g\left(\frac{(y_0+y_1)}{2}, z_{mid}\right) \end{aligned} \qquad (5)$$

Equation (5) still requires a nonlinear solution for z_1 at the end of each step. In this paper, z_{mid} is approximated to be $z_{mid} = (z_0+z_1)/2$ for the efficiency of implementation and $g(y_1,z_1)$ is forced to be zero (as a projected step). This method can be viewed as a hybridization of the implicit mid-point method for ODE variables and the trapezoid method for algebraic variables. This hybridization approach seems to work well for smooth DAEs, and it might fail for problems in which z is discontinuous with time or for very stiff DAEs that require L-stable schemes. (Ascher, 1989)

$$\begin{aligned} y_1 &= y_0 + hf\left(\frac{(y_0+y_1)}{2}, \frac{(z_0+z_1)}{2}\right) \\ 0 &= g(y_1, z_1) \end{aligned} \qquad (6)$$

2.4 Radau IIA Method

The third-order Radau IIA (Rad) method for equation (1) can be written as follows:

$$\begin{aligned} y_1 &= y_0 + \frac{h}{4} f\left(y_1, z_1\right) + \frac{3h}{4} f\left(y_{int}, z_{int}\right) \\ 0 &= g(y_1, z_1) \\ y_{int} &= y_0 - \frac{h}{12} f\left(y_1, z_1\right) + \frac{5h}{12} f\left(y_{int}, z_{int}\right) \\ 0 &= g\left(y_{int}, z_{int}\right) \end{aligned} \tag{7}$$

where y_{int} and z_{int} are ODE and algebraic variables at $t = h/3$. The system size is doubled for the Newton-Raphson solver for this method.
Equation (7) can be rewritten (by using the inverse of Runge-Kutta coefficient matrix A) as

$$\begin{aligned} \frac{5}{2}\left(y_1 - y_0\right) - \frac{9}{2}\left(y_{int} - y_0\right) &= hf\left(y_1, z_1\right) \\ 0 &= g(y_1, z_1) \\ \frac{1}{2}\left(y_1 - y_0\right) + \frac{3}{2}\left(y_{int} - y_0\right) &= hf\left(y_{int}, z_{int}\right) \\ 0 &= g\left(y_{int}, z_{int}\right) \end{aligned} \tag{8}$$

This change makes a significant difference in computational efficiency. For finite difference discretization of a single PDE in 2D with a five-point stencil, equation (7) will have 10 non-zero entries in the Jacobian for a particular row instead of only 6 non-zero entries in equation (8).

2.5 Trapezoid-Backward Difference

Trapezoid-Backward Difference (TRBDF2) is a single-step method that uses a Trapezoid step followed by second-order backward difference formula, providing an L-stable method (Hosea & Shampine, 1996). This can be written for index-1 DAEs as:

$$\begin{aligned} y_{int} &= y_0 + \frac{h\gamma}{2} f\left(y_0, z_0\right) + \frac{h\gamma}{2} f\left(y_{int}, z_{int}\right) \\ 0 &= g\left(y_{int}, z_{int}\right) \end{aligned} \tag{9}$$

where y_{int} and z_{int} are the ODE and algebraic variables at $g = h\,\gamma$, and $\gamma=2-\sqrt{2}$. The trapezoid step is followed by the backward difference formula.

$$\begin{aligned} y_1 &= -\frac{(1-\gamma)}{2} y_0 + \frac{1}{2(1-\gamma)} y_{int} + h\frac{\gamma}{2} f\left(y_1, z_1\right) \\ 0 &= g(y_1, z_1) \end{aligned} \tag{10}$$

The Trapezoid and BDF steps have the same coefficients for implicitly evaluating the slopes. This belongs to singly Diagonally implicit Runge-Kutta methods. (Kennedy & Carpenter, 2016) The same Jacobian and a single LU decomposition can be used for the Trapezoid and BDF steps.

2.6 TRX2

Two half-steps of the Trapezoid method can be used to arrive at this algorithm. The main advantage is the higher-order embedded error estimate and lower error constant compared to TRBDF2 for non-stiff and mildly-stiff systems that do not require L-stability. (Hosea & Shampine, 1996)

The method can be written for index-1 DAEs as

$$\begin{aligned} y_{int} &= y_0 + \frac{h}{4} f\left(y_0, z_0\right) + \frac{h}{4} f\left(y_{int}, z_{int}\right) \\ 0 &= g\left(y_{int}, z_{int}\right) \end{aligned} \tag{11}$$

where y_{int} and z_{int} are the ODE and algebraic variables at $t = h/2$. The trapezoid step is followed by a second trapezoid step.

$$\begin{aligned} y_1 &= y_0 + \frac{h}{4} f\left(y_0, z_0\right) + \frac{h}{2} f\left(y_{\text{int}}, z_{\text{int}}\right) + \frac{h}{4} f\left(y_1, z_1\right) \\ 0 &= g(y_1, z_1) \end{aligned} \tag{12}$$

The Trapezoid steps have the same coefficient for the implicit evaluation of the slopes. This method also belongs to singly Diagonally implicit Runge-Kutta methods.

2.7 Adaptive Backward Difference Formula

Variable time step adaptive backward difference formula (BDF2) for index 1 DAE is given by:

$$\begin{aligned} y_n &= \frac{(\alpha+1)^2}{2\alpha+1} y_{n-1} - \frac{\alpha^2}{2\alpha+1} y_{n-2} + \frac{h(\alpha+1)}{2\alpha+1} f_n\left(y_n, z_n\right) \\ 0 &= g\left(y_n, z_n\right) \end{aligned} \tag{13}$$

(Hairer & Nørsett, Syvert P. Wanner, 1993)

The first step is taken to be a Euler-backward step.

All the methods discussed in this work require a nonlinear solver to update and find y_1 and z_1. The difference between y and z is defined as uu to facilitate this.

$$y_1 = uu_i + y_0,\ i = 1..N_{ode},\ z_1 = uu_i + z_0,\ i = N_{ode} + 1..\ N_{ode} + N_{ae}. \tag{14}$$

y_1 and z_1 are stacked together in vector form as $\boldsymbol{Y_1}$

$$\mathbf{Y_1} = \mathbf{Y_0} + \mathbf{uu} \tag{15}$$

Table 1 summarizes the residual form for all the different methods.

All the methods involve finding the ***uu*** vector for a given $\mathbf{Y_0}$ vector as input and can be implemented with h = 0 to find the consistent initial condition for z at t = 0 without creating a new set of equations just for initialization.

Table 1. Residual form of the different numerical methods.

Method	**Residual**
EB	$uu_i - h\cdot f_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = 1..N_{ode}$ $g_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = N_{ode}+1..N_{ode}+N_{ae}$
TR/CN	$uu_i - \frac{h}{2} f_i(\mathbf{uu}+\mathbf{Y_0}) - \frac{h}{2} f_i(\mathbf{Y_0}) = 0, i = 1..N_{ode}$ $g_i(\mathbf{uu}+\mathbf{Y_0}) = 0,\ i = N_{ode}+1..N_{ode}+N_{ae}$
IMPTRAP	$uu_i - h\cdot f_i\left(\frac{\mathbf{uu}}{2}+\mathbf{Y_0}\right) = 0, i = 1..N_{ode}$ $g_i(\mathbf{uu}+\mathbf{Y_0}) = 0,\ i = N_{ode}+1..N_{ode}+N_{ae}$
Rad2	$\frac{5}{2}uu_i - \frac{9}{2}uu_{i+N_{ode}+N_{ae}} - h\cdot f_i(\mathbf{uu}+\mathbf{Y_0}) = 0,\ i = 1..N_{ode}$ $g_i(\mathbf{uu}+\mathbf{Y_0}) = 0,\ i = N_{ode}+1..\ (N_{ode}+N_{ae})$ $\frac{1}{2}uu_{i-(N_{ode}+N_{ae})} + \frac{3}{2}uu_i - h\cdot f_i(\mathbf{uu}+\mathbf{Y_0}) = 0,\ i = (N_{ode}+N_{ae})+1..(N_{ode}+N_{ae})+N_{ode}$ $g_i(\mathbf{uu}+\mathbf{Y_0}) = 0,\ i = (N_{ode}+N_{ae})+N_{ode}+1\ ..\ 2(N_{ode}+N_{ae})$
TRBDF2	$uu_i - \frac{h\gamma}{2} f(\mathbf{Y_0}) - \frac{h\gamma}{2}\cdot f_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = 1..N_{ode}$ $g_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = N_{ode}+1..N_{ode}+N_{ae}$ $\mathbf{Y}_{\text{int}} = \mathbf{Y_0} + \mathbf{uu}$ $uu_i + \frac{(3-\gamma)}{2}\mathbf{Y_0} - \frac{1}{2(1-\gamma)}\mathbf{Y}_{\text{int}} - \frac{h\gamma}{2}\cdot f_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = 1..N_{ode}$ $g_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = N_{ode}+1..N_{ode}+N_{ae}$ $\mathbf{Y}_1 = \mathbf{Y_0} + \mathbf{uu}$
TRX2	$uu_i - \frac{h}{4} f(\mathbf{Y_0}) - \frac{h}{4}\cdot f_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = 1..N_{ode}$ $g_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = N_{ode}+1..N_{ode}+N_{ae}$ $\mathbf{Y}_{\text{int}} = \mathbf{Y_0} + \mathbf{uu}$ $uu_i + 2(\mathbf{Y_0} - \mathbf{Y}_{\text{int}}) + \frac{h}{4} f(\mathbf{Y_0}) - \frac{h}{4}\cdot f_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = 1..N_{ode}$ $g_i(\mathbf{uu}+\mathbf{Y_0}) = 0, i = N_{ode}+1..N_{ode}+N_{ae}$

	$\mathbf{Y}_1 = \mathbf{Y}_0 + \mathbf{uu}$
BDF2	$\mathbf{uu} = \frac{(\alpha+1)^2}{2\alpha+1}\mathbf{Y}_{n\text{-}1} - \frac{\alpha^2}{2\alpha+1}\mathbf{Y}_{n\text{-}2} + \frac{h(\alpha+1)}{2\alpha+1} f_n$ $g_n(\mathbf{uu}+\mathbf{Y}_0) = 0$

3 Error Control and Time-Stepping

Error control in the code is achieved based on the absolute tolerance requirement. In this paper, relative tolerance is taken to be a scalar quantity and set to be 10 times the absolute tolerance. This is achieved for the first four algorithms by running a single step of the algorithm with time step = *h* and two-half steps (*t*=*h/2* twice to complete the same step) and finding the error using two different estimates for the first four algorithms.

$$
\begin{aligned}
y_{err} &= \frac{\left(y_{h_2} - y_h\right)}{\left(2^p - 1\right)} \\
err &= \left\| \frac{y_{err}[i]}{\left(a_{tol} + y_{err}[i].r_{tol}\right)} \right\| \\
h_{new} &= min\left(h_{max}, h_{old}\, min\left(3.0, 0.9\left(\frac{1}{err}\right)^{\left(\frac{1}{(p+1)}\right)} \right)\right)
\end{aligned}
\tag{16}
$$

In equation (16), y_{h2} and y_h are solutions after two half steps *h/2*, and one single time step *h*, respectively and a_{tol} and r_{tol} represent the absolute and relative tolerance, while *p* is the order of accuracy and is 1, 2, 2, 3 for EB, CN, IMPTRAP and Rad methods, respectively, considered in this paper. Both the ODE and algebraic variables are included in the error estimates. After the error calculation, the new step size h_{new} is chosen between the maximum step size h_{max} and previous step size h_{old} with correction. This approach also provides a higher-order accuracy at the end of each time step with Richardson extrapolation as:

$$
y_R = \frac{\left(2^p y_{h_2} - y_h\right)}{2^p - 1} \tag{17}
$$

For certain stiff problems, this might cause stability issues, and the code can be modified to use y_{h2} at the end of each step. Also, ODE and algebraic variables are extrapolated using this formula at the end of each step.

While the number of Newton-Raphson iterations required for convergence can be used to find a criterion for updating the Jacobian, in this code, if *err* is greater than 0.1, the Jacobian is updated. If *err* is greater than 1, the step is rejected, and the time step is reduced by 4. Also, the Jacobian obtained at a particular time *t* is used for all the linear solves for the Newton-Raphson method for every iteration for both the calculation with *t* = *h* and two repeated steps with *t* = *h/2*. This means the code will not perform LU Decomposition more than once for a given time step. For both TRBDF and TRX2, an embedded error estimate is used to arrive at the error and identify the next step. For the adaptive BDF2, a predictor based on past values were used to estimate the errors. The backward difference formula of order greater than 2 is not A-stable and is not considered in this paper.

4 Symbolic Math and Use of Search Tools

One of the unique aspects of the developed code is the analytic Jacobian found using search tools in Maple. When a wide range of spatial discretization methods is used to convert PDEs to DAEs (finite difference, collocation, spectral, and finite volume methods), the sparsity pattern is hard to identify and code for general PDEs and boundary conditions for different spatial discretization approaches. In this code, for a given system of DAEs, Maple's search (ListTools:-Search) and symbolic capabilities are exploited to arrive at a robust and efficient way to (1) search, sort and label variables and indices, (2) differentiate expressions for analytic Jacobian (3) create sparse Jacobian and procedures for the same.

The code implemented scales well for two-dimensional PDEs and takes very little time (seconds) to find the analytic sparse Jacobian even for >100,000 DAEs resulting from the semi-discretization of two-dimensional PDEs.

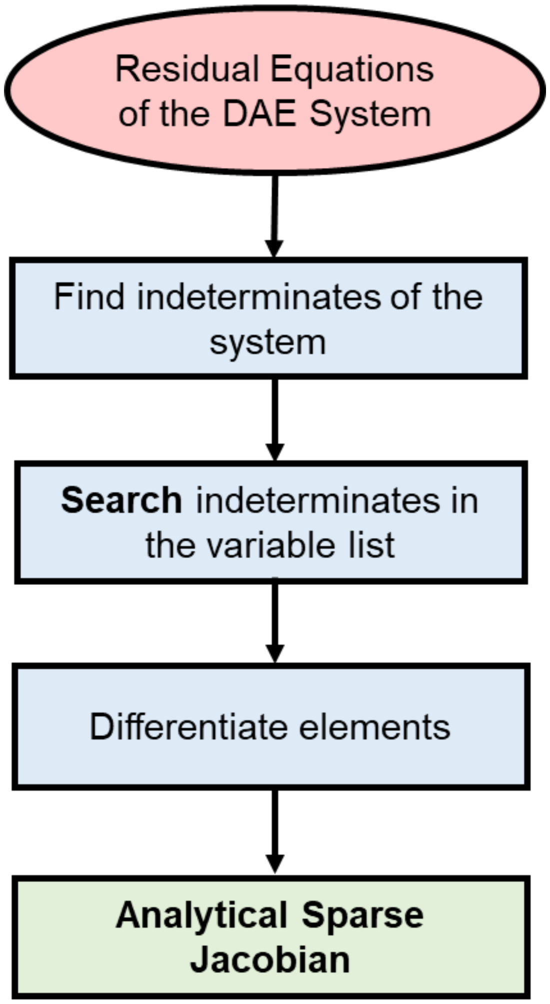
Residual Equations of the DAE System
Find indeterminates of the system
Search indeterminates in the variable list
Differentiate elements
Analytical Sparse Jacobian

Figure 1. Schematic of the search algorithm to evaluate the sparse analytical Jacobian.

5 Code Implementation with Examples

The solver can be easily called from Maple to solve index-1 DAEs. This section goes over implementing the proposed solver to solve some examples of problems of index-1 DAEs. Examples 1-3 show the capabilities of the DAE solver to find initial conditions and solve stiff problems. Examples 4-6 demonstrate the solver's ability to solve complex problems resulting in a large system of DAEs.

5.1 Example 1:

Consider the DAE system:

$$\begin{aligned} \frac{dy}{dt} &= z \\ y^2 + z^2 &= 1 \\ y(0) = 0&; z(0) = 0.95 \end{aligned} \tag{18}$$

The exact initial condition for *z* is 1.0, but 0.95 is provided as an approximate value for *z* to test the code's ability to initialize algebraic variables. First, the solver is called from Maple, and then the equations are called and solved as below.

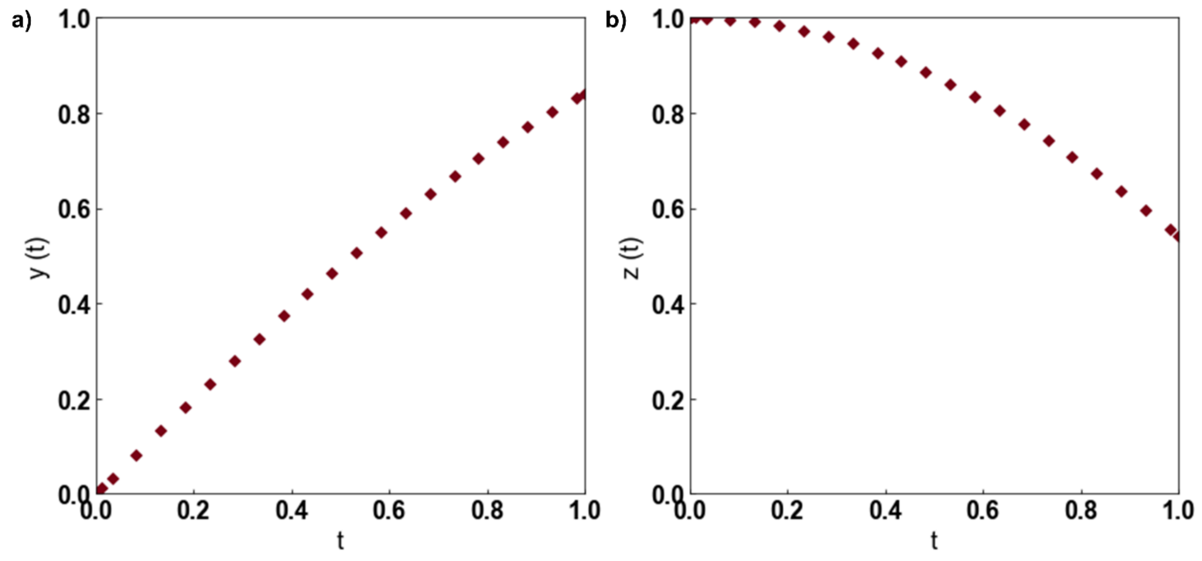

Figure 2. a) $y(t)$ vs t and b) $z(t)$ vs t plot using the data obtained with the implemented DAE solver with the IMP algorithm. The code can initialize z correctly and then solve the DAE model.

The solver took only 29 time steps to simulate this example. The user has to stack all the ODEs in "eqodes" and all the AEs in "eqaes". If there are no algebraic variables, the users should provide an empty list [] for eqaes.

The solver options include the following inputs:
t_f = total time of simulation
a_{tol} = absolute tolerance expected for the simulation
h_{init} = starting time step for simulation, expected to be less than t_f. Suggested value is $h_{init} = min(10^{-6}, t_f, a_{tol})$
h_{max} = maximum value of the time step. Having a large h_{max} value may result in not having enough points for plots and may also result in code failure for some problems. The suggested value in the code is $h_{max} = t_f/20$
N_{tot} = maximum number of time steps taken to be 1000.

The initial conditions for ODE and algebraic variables are listed in "ICs". This should follow the order specified in eqodes (list of ODEs) and then eqaes (list of AEs). The initial conditions are provided for algebraic variables as well. These can be approximate values for the algebraic variables as the solver does the initialization. While Maple can automatically find and separate ODEs and AEs from a given system, this is avoided in the solver for efficiency purposes. The solver can be called to provide the CPU time as well.

5.2 Example 2:

Van der Pol's system is considered (Van der Pol, 1926) as a test case for stiff solvers. The developed solver can simulate this model, as shown in Figure 2.

$$\begin{aligned} \frac{dx}{dt} &= \mu\left(1-y^2\right)x-y \\ \frac{dy}{dt} &= x \\ x(0) = 0;\ & y(0) = 2;\ \mu = 2 \end{aligned} \tag{19}$$

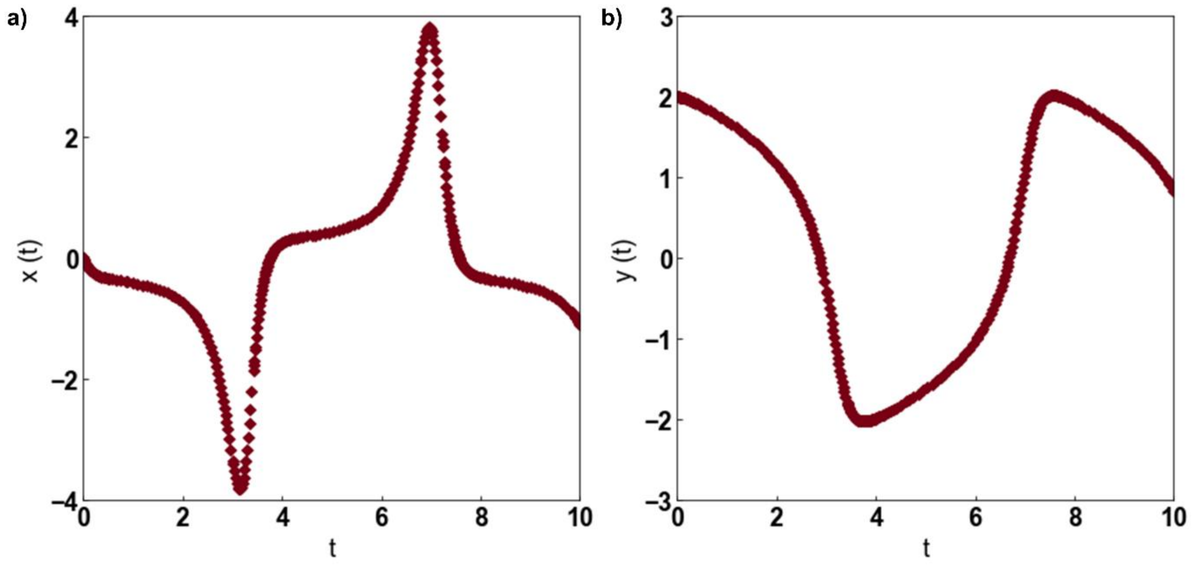

Figure 3: a) *x*(*t*) vs. *t* and b) *y*(*t*) vs. *t* plot using the data obtained with the implemented DAE solver with the IMP algorithm. The code can solve the stiff system of ODEs.

5.3 Example 3:

Next, a simple DAE system that fails with Maple's default DAE solver (because of an unknown initial condition for the algebraic variable) is solved. The solver can find the exact initial condition for the algebraic variable *z* and solve the system, as shown in Figure 4.

$$\begin{aligned} \frac{dy}{dt} &= -2y+z^2 \\ -100\ln(z) &+ 2y = 5 \\ y(0) = 2;\ & z(0) = 1 \end{aligned} \tag{20}$$

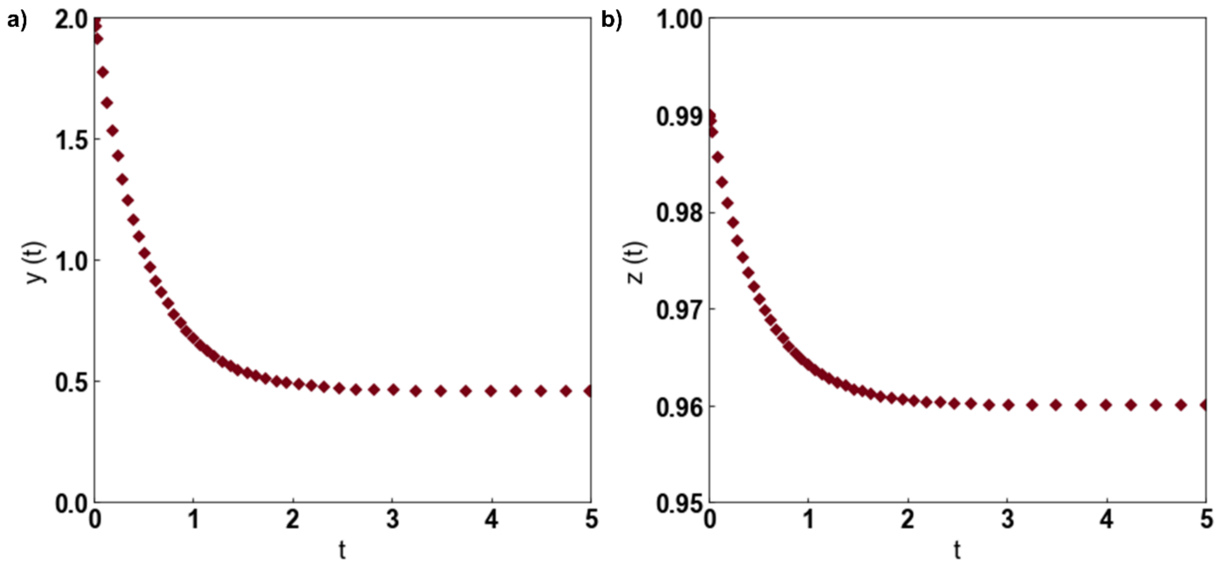

Figure 4. a) *y*(*t*) vs. *t* and b) *z*(*t*) vs. *t* plot using the data obtained with the implemented DAE solver with the IMP algorithm. The code can initialize *z* correctly and then solve the DAE model.

5.4 Example 4:

Next, a system of partial differential equations in one dimension is considered:

$$\begin{aligned}
&\frac{\partial c}{\partial t} = \frac{\partial^2 c}{\partial x^2} - c(1+z)\\
&0 = \frac{\partial^2 z}{\partial x^2} - (1-c^2)\exp(-z)\\
&c(x,0) = 1; z(x,0) = 0\\
&\frac{\partial c}{\partial x}(0,t) = 0; \frac{\partial z}{\partial x}(0,t) = 0\\
&c(1,t) = 1; z(1,t) = 0
\end{aligned} \tag{21}$$

Discretized form of equation (21) with the cell-centered finite difference approach in *x* can be written as

$$\begin{aligned}
&\frac{dc_i}{dt} = \frac{c_{i+1} - 2c_i + c_{i-1}}{(\Delta x)^2} - c_i(1+z_i), i = 1..N\\
&0 = \frac{z_{i+1} - 2z_i + z_{i-1}}{(\Delta x)^2} - (1-c_i^2)\exp(-z_i), i = 1..N\\
&c_i(0) = 1; z_i(0) = 0, i = 1..N\\
&\frac{c_1 - c_0}{\Delta x} = 0; \frac{z_1 - z_0}{\Delta x} = 0\\
&\frac{c_N + c_{N+1}}{2} = 1; \frac{z_N + z_{N+1}}{2} = 0
\end{aligned} \tag{22}$$

where N is the number of elements in x.

The code is written so N can be changed from 2, 4, 8, or 16 until satisfactory results are obtained. It is noted that the code works even for N = 10,000 node points in x in very few seconds. For this problem, c and z at x = 0 and t = 1 (given by $(c_0+c_1)/2$ and $(z_0+z_1)/2$) converge with increasing values of N = 4, 8, 16, 32, 64 for IMP as given in Table 2. For N = 128 and N = 256, the h_{max} value was reduced to $t_f/45$.

Table 2. Convergence analysis of DAE variables in Example 4.

Number of node points	$c(0,1)$	$z(0,1)$
2	0.721262011548187	-0.277562804790677
4	0.714227915206798	-0.270426518683985
8	0.712469481991088	-0.268555018896386
16	0.712030211180292	-0.268083125242616
32	0.711920430403511	-0.267964915226396
64	0.711917661490991	-0.267911284467792
128	0.711892311529726	-0.267921896322330
256	0.711886155827820	-0.267927929302667

The CPU time for all the examples is tabulated in Table 2 for all the methods using Intel® Core™ i9–12900K CPU and 32 GB RAM.

5.5 Example 5:

A 2-Dimensional problem is next considered:

$$\begin{aligned} \frac{\partial c}{\partial t} &= \frac{\partial^2 c}{\partial x^2}+\frac{\partial^2 c}{\partial y^2}-\phi^2 c^2 \\ & c(x,y,0)=1\ \forall\ x,y\ except\ y=1\ and\ x\neq 0,1 \\ & c(0,1,0)=0 \\ & \frac{\partial c}{\partial x}(0,y,t)=0; \frac{\partial c}{\partial y}(x,0,t)=0 \\ & c(x,1,t)=1; c(1,y,t)=1 \end{aligned} \tag{23}$$

f can be varied as 0.1, 1, and 10 to see the need for a different number of node points in x. Typically, method of lines is useful for parabolic PDEs and elliptic PDEs. For hyperbolic PDEs, spatial discretization should be done carefully. Upwind (Osher & Sethian, 1988) and WENO (Jiang & Peng, 2000) methods are recommended for the same. A cell-centered finite difference (FD) was used to simulate this model for f = 0.5. Since the Maple code is provided and the FD scheme for a more detailed model is provided for example 6, the FD scheme is not provided for this model.

5.5 Example 6:

Next, the concentration and potential distributions (tertiary current distribution) in an electrolyte (electrochemical system) are considered. With the dilute solution theory and the electroneutrality assumption, the model considered is:

$$\begin{aligned}
&\frac{\partial c}{\partial t} = D_x \frac{\partial^2 c}{\partial x^2} + D_y \frac{\partial^2 c}{\partial y^2} \\
&\frac{\partial}{\partial x}\left(D_x c \frac{\partial \phi}{\partial x} \right) + \frac{\partial}{\partial y}\left(D_y c \frac{\partial \phi}{\partial y} \right) = 0
\end{aligned} \tag{24}$$

with the boundary conditions

$$\begin{aligned}
&\left.\frac{\partial c}{\partial y}\right|_{y=0} = 0, \ \left.\frac{\partial \phi}{\partial y}\right|_{y=0} = 0 \\
&\left.\frac{\partial c}{\partial y}\right|_{y=H} = 0, \ \left.\frac{\partial \phi}{\partial y}\right|_{y=H} = 0 \\
&D_x \left.\frac{\partial c}{\partial x}\right|_{x=L} = \delta, \ D_x c \left.\frac{\partial \phi}{\partial x}\right|_{x=L} = \delta \\
&\begin{cases} D_x \left.\dfrac{\partial c}{\partial x}\right|_{x=0} = Da \cdot c \cdot \phi, \quad D_x \left.\dfrac{\partial \phi}{\partial x}\right|_{x=0} = Da \cdot \phi, \qquad 0 < y <= \dfrac{H}{2} \\ \left.\dfrac{\partial c}{\partial x}\right|_{x=0} = 0, \qquad\qquad \left.\dfrac{\partial \phi}{\partial x}\right|_{x=0} = 0 \ \dfrac{H}{2}, \quad \dfrac{H}{2} < y <= H \end{cases}
\end{aligned} \tag{25}$$

The initial condition for *c* was taken to be 1 everywhere. Equation (23) is the final form of the model for transporting a binary electrolyte based on the Nernst-Planck equation for diffusive and migrative flux coupled with electroneutrality. Equal diffusivities were assumed for both the cation and anion, and the model assumes different constant diffusivities in the *x* and *y* direction (Newman & Thomas-Alyea, 2004).

Discretized form of equation (23) with cell-centered finite difference method in *x* and *y* can be written as

$$\frac{dc_{i,j}}{dt} = D_x \frac{c_{i+1,j} - 2c_{i,j} + c_{i-1,j}}{(\Delta x)^2} + D_y \frac{c_{i,j+1} - 2c_{i,j} + c_{i,j-1}}{(\Delta y)^2}, i = 1..N, \ j = 1..M$$

$$\begin{aligned}
&0 = \\
&\frac{\left(D_x \dfrac{c_{i+1,j} + c_{i,j}}{2} \left(\dfrac{\phi_{i+1,j} - \phi_{i,j}}{\Delta x} \right) \right) - \left(D_x \dfrac{c_{i,j} + c_{i-1,j}}{2} \left(\dfrac{\phi_{i,j} - \phi_{i-1,j}}{\Delta x} \right) \right)}{\Delta x} \\
&+ \frac{\left(D_y \dfrac{c_{i,j} + c_{i,j+1}}{2} \left(\dfrac{\phi_{i,j+1} - \phi_{i,j}}{\Delta y} \right) \right) - \left(D_y \dfrac{c_{i,j} + c_{i,j-1}}{2} \left(\dfrac{\phi_{i,j} - \phi_{i,j-1}}{\Delta y} \right) \right)}{\Delta y}, i = 1..N, \ j = 1..M
\end{aligned}$$

$$\frac{c_{i,1}-c_{i,0}}{\Delta y}=0;\frac{\phi_{i,1}-\phi_{i,0}}{\Delta y}=0, i=1..N$$
$$\frac{c_{i,M+1}-c_{i,M}}{\Delta y}=0;\frac{\phi_{i,M+1}-\phi_{i,M}}{\Delta y}=0, i=1..N$$
$$D_x\frac{c_{N+1,j}-c_{N+1,j}}{\Delta x}=\delta;D_x\frac{c_{N+1,j}+c_{N,j}}{2}\left(\frac{\phi_{N+1,j}-\phi_{N,j}}{\Delta x}\right)=\delta,\ j=1..M \tag{26}$$
$$\begin{cases} D_x\dfrac{c_{1,j}-c_{0,j}}{\Delta x}=Da\left(\dfrac{c_{0,j}+c_{1,j}}{2}\right)\left(\dfrac{\phi_{0,j}+\phi_{1,j}}{2}\right), D_x\dfrac{\phi_{1,j}-\phi_{0,j}}{\Delta x}=Da\left(\dfrac{c_{0,j}+c_{1,j}}{2}\right)=0,\ j=1..M/2 \\ \dfrac{c_{1,j}-c_{0,j}}{\Delta x}=0, \dfrac{\phi_{1,j}-\phi_{0,j}}{\Delta x}=0,\ j=M/2+1..M \end{cases}$$

In equation (26), *Da* is The Damkohler number and*d* is the applied current density in dimensionless form. Typically, D_x or D_y can be eliminated (with scaling), but we leave it as is in equation (26) and the code.

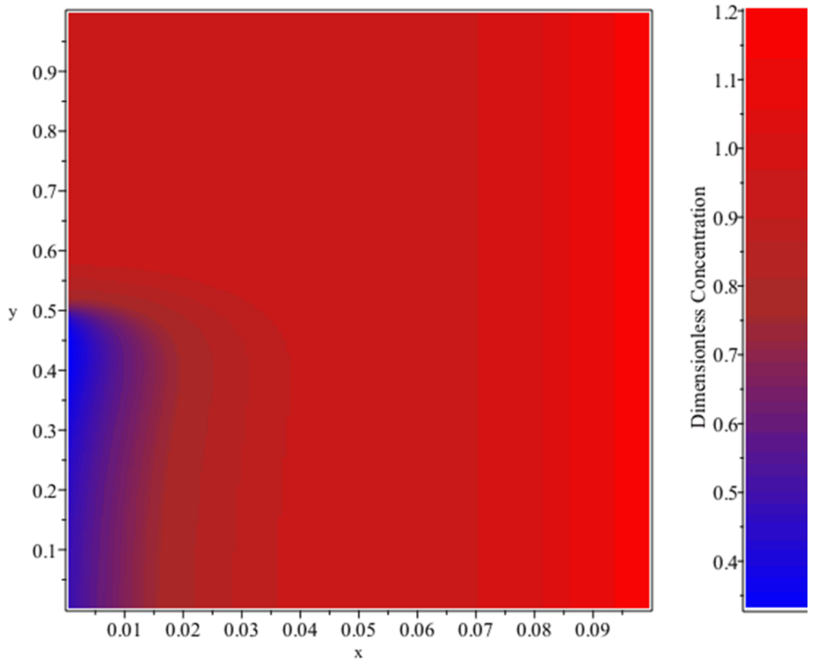

Figure 5: Surface plot of concentration at short times for the electrochemical model. A boundary layer is formed near the electrode at *x* = 0, which then diffuses with time.

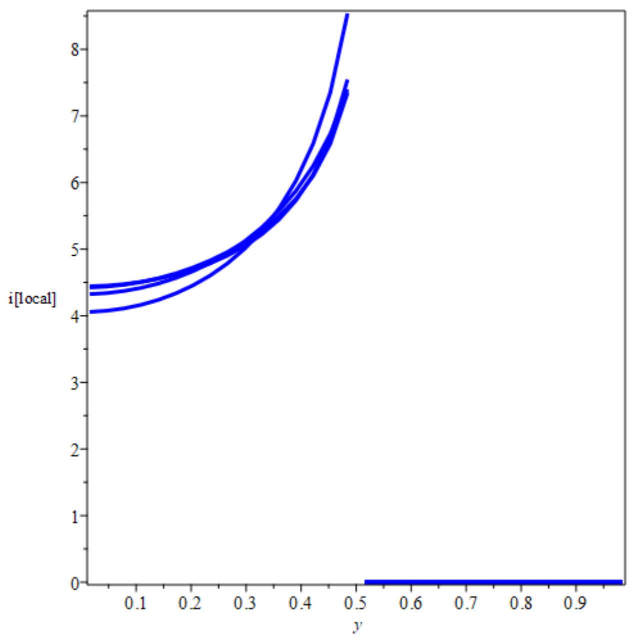

Figure 6: Current density (flux distribution) at x = 0 from the tertiary current distribution model at different times. Singularity is seen at y = 0.5, reducing the observed order spatial discretization accuracy.

Results at x = 0.5, y =0 and x = 0, y = 0.5 are used to study the convergence of results for N with M = 2N, Δx = 0.1/N, Δy = 1/M. Results obtained for different values of N are shown in Table 3.

Table 3. Convergence analysis for the DAE variables in Example 5 using δ=0.26 and Da = 1.3.

Number of node points	$c(0.5,0,1)$	$\phi(0.5,0,1)$	$c(0,0.5,1)$	$\phi(0,0.5,1)$
4	0.73141595297221	0.90153716519290	0.82780338761445	1.00313359989368
8	0.73856240938989	0.86892299701114	0.81787060876857	0.96255607933976
16	0.74101529446961	0.85857149097091	0.81305008905655	0.94834350783886
32	0.74181451287246	0.85533393255737	0.81101670965265	0.94350508469314
64	0.74206266027357	0.85435092997272	0.81024158000760	0.94192291344425
128	0.74213696509076	0.85406060115285	0.80996683223722	0.94142359307161
256	0.74215861876193	0.85397678526279	0.80987438323878	0.94127046045326

One can see that the code scales well for a large number of node points which are needed for this problem because of singularity at y = 0.5 and x = 0. Using variable grids in x and y and other numerical methods can help solve the problem more efficiently. Convergence analysis and finding the optimal method for spatial discretization are beyond the scope of this paper and the code developed. Surface plots at short times and current distribution at x = 0 are also plotted in Figures 5 and 6.

6 Analysis of Algorithms and Results

A comparison of properties of different algorithms considered in this paper is reported below in Table 4. Computational results are summarized in Table 5 and compared to the results obtained from gPROMs. For low tolerances (high accuracies), the Radau IIA method is recommended for a system with fewer DAEs.

For systems with a large number of DAEs, a linear solver dominates the total computational cost. For the PDE examples discussed in this paper, the IMPTRAP method was the most efficient for an absolute tolerance of 10^{-6}. Both Euler-backward and Radau IIA methods are L-stable and are recommended for very stiff DAEs. For Hamiltonian DAEs, the IMPTRAP method offers symplecticity for the ODE variables only, as the Trapezoid method is only weakly symplectic (symmetric). For problems requiring monotonicity, only Euler-backward method is offered.

Table 4. Properties of different time-stepping algorithms implemented in this paper.

Method	**Order**	**A Stability**	**L-Stability**	**Extrapolated Order**
Euler Backward	1	Yes	Yes	2
Crank-Nicolson	2	Yes	No	4
Implicit mid-point-Trapezoid	2	Yes	No	4
Radau IIA	3	Yes	Yes	4
Trapezoid-Backward Difference	2	Yes	Yes	NA
TRX2	2	Yes	No	NA
Adaptive BDF2	2	Yes	Yes	NA

Additional comments about different examples are given below.

Example 1: This is a simple index-1 DAE, but the profile is oscillatory. The code presented in this paper can meet the tolerances specified. For a set absolute tolerance of 10^{-6}, the proposed solver can simulate this model in 33 time steps and a CPU time of 58 ms with the Radau method with 2 failed time steps. The maximum time step used was 0.05.

Example 2: Van der Pol's model is a stiff system of ODEs. So, an empty list is passed to the solver for the algebraic equations. For a set absolute tolerance of 10^{-6}, the solver can simulate this model in 114 time steps and CPU time of 293 ms with the Radau method with 17 failed time steps. The maximum time step was chosen to be 0.1. One can see that the solver can adapt and use smaller time steps as needed.

Example 3: This simple DAE system was chosen to show the importance of consistent initial conditions for the algebraic equations and variables. Maple's inbuilt DAE solver fails to solve this DAE for the initial condition of $z=1$ for $y=2$. The implemented solver is able to simulate this model in 34 time steps and CPU time of 55 ms with the Radau method with 0 failed time steps. The maximum time step was chosen to be 0.5.

Example 4: This example was chosen to show the scalability of the solver for a large-scale system and highly nonlinear algebraic equations. The PDE system is simulated with a cell-centered finite difference method in x. Maple's inbuilt DAE solver fails to solve the PDE system for $N>5$ elements in x. The implemented solver is able to simulate this model in 45 time steps and CPU time of 311 ms with the IMPTRAP method with 0 failed time steps with $N = 128$ elements (total of 260 DAEs). The maximum time step was chosen to be 0.05.

Example 5: This example was chosen to show the scalability of the solver for two-dimensional PDEs. The PDE system is simulated with a cell-centered finite difference method in x. The implemented solver is able to simulate this model in 37 time steps and CPU time of 3.0 s with the IMPTRAP method with 0 failed time steps with $N\times M = 64\times64$ elements (total of 4352 DAEs). The maximum time step was chosen to be 0.25.

Example 6: This example was chosen to show the scalability of the solver for two-dimensional coupled PDEs resulting in large-scale DAEs and similarity of CPU time requirements with ODE models (example 5). The model chosen has a singularity at $x = 0$ and $y = 0.5$, and the model needs a large number of grids for uniform node spacing. The model solves both concentration and potential in an electrolyte. The PDE system is simulated with a cell-centered finite difference method in x. The implemented solver is able to simulate this model in 38 time steps and CPU time of 16.65 s with the IMPTRAP method with 0 failed time steps with $N\times M = 64\times128$ grids (total of 17152 DAEs). Note that the maximum time step was chosen as 0.05, and the maximum growth rate was restricted to 3 (in the solver).

It is to be noted that restricting the value of the maximum time step will significantly slow down the solver.

Table 5. Computational (CPU) time for different algorithms for different examples considered. For all the models chosen, the following solver parameters* were fixed h_{init} = 1e-6, a_{tol} = 1e-6, and the setup time did not include in CPU time. Three different CPU times from separate runs are reported to record the CPU time at three instances. The unit of CPU Time is in milliseconds (green), seconds (blue), or minutes (red).

Example	**Methods**	**CPU Time [ms, s, m]**	**Number of DAEs**	**Number of Time Steps**	**Number of Rejected Steps**
1	CN	33,31,26	2	31	0
	IMPTRAP	33,32,32		31	0
	EB	81,84,79		129	1
	Rad	31,30,31		31	0
	TRBDF2	24,23,25		34	0
	TRX2	16,16,15		32	0
	BDF2pred	27,27,26		74	1
	gPROMs	31,47,16		31	-
	IDA Dense	74,72,73		-	
2	CN	169,165,206	2	299	9
	IMPTRAP	153,158,160		278	17
	EB	1.26,1.32,1.27		2687	3
	Rad	97,131,94		114	17
	TRBDF2	285,243,289		369	6
	TRX2	165,141,146		301	8
	BDF2pred	370,393,374		1044	7
	gPROMs	31,31,47		231	-
	IDA Dense	69,72,71		-	
3	CN	49,47,49	2	55	0
	IMPTRAP	39,47,56		55	0
	EB	158,157,166		321	0
	Rad	41,32,34		34	0
	TRBDF2	51,46,46		76	0
	TRX2	31,30,28		65	0
	BDF2pred	69,55,67		196	1
	gPROMs	16,31,31		52	-
	IDA Dense	72,75,71		-	
4	CN	167,161,170	260	36	0

(N=128)	IMPTRAP	112,161,165		36	0
	EB	266,276,314		74	4
	Rad	273,312,264		40	4
	TRBDF2	188,181,185		33	0
	TRX2	178,142,177		32	0
	BDF2pred	215,216,214		62	2
	gPROMs	62,47,31		54	-
	IDA Dense	143,144,165		-	
4 (N=256)	CN	259,253,289	516	35	0
	IMPTRAP	244,283,292		35	0
	EB	480,488,484		64	3
	Rad	626,642,645		44	6
	TRBDF2	319,308,309		32	0
	TRX2	302,303,333		32	0
	BDF2pred	556,538,559		88	10
	gPROMs	62,78,62		54	-
	IDA Dense	298,295,312		-	
5 (N=64, M=64) Phi = 0.5 UMFPACK PARDISO	CN	2.35,2.33,2.48 3.52,3.66,3.62	4352	37	0
	IMPTRAP	2.12,2.09,2.10 2.96,2.48,3.00		37	0
	EB	2.44,2.43,2.47 3.61,3.72,3.55		40	3
	Rad	5.07,5.10,5.11 7.03,6.91,7.08		41	4
	TRBDF2	2.88,2.86,2.87 3.97,3.95,3.90		33	0
	TRX2	2.88,2.88,2.90 3.94,3.91,3.81		33	0
	BDF2pred	3.25,3.25,3.24 4.47,4.51,4.47		53	2
	gPROMs	1.297,1.312,1.297		126	-
	IDA Dense	14.533,14.552,14.490		-	
5 (N=128, M=128) Phi = 0.5 UMFPACK PARDISO	CN	12.39,12.11,12.23 15.98,16.26,15.90	16896	40	2
	IMPTRAP	11.60,11.58,11.60 14.12,14.59,14.48		40	2
	EB	12.73,12.68,12.48 16.82,16.61,16.57		43	6
	Rad	25.78,25.95,25.70 32.26,32.16,31.95		42	5
	TRBDF2	13.30,13.29,13.29 17.33,17.88,17.65		33	0
	TRX2	13.37,13.30,13.33 17.45,17.71,17.71		33	0
	BDF2pred	12.84,12.86,12.86 17.55,17.70,17.47		46	1
	gPROMs	6.328,6.328,6.359		126	-

6 (N=128, M=256) UMFPACK PARDISO	CN	83.21,84.50,84.98 79.99,85.22,90.16	67072	38	1
	IMPTRAP	83.59,84.11,83.02 88.93,82.38,83.37		38	1
	EB	102.27,100.84,108.84 105.47,105.06,102.44		43	6
	Rad	3.98,4.01,4.19 3.40,3.38,3.35		43	4
	TRBDF2	105.48,105.01,105.38 92.22,91.76,92.00		34	0
	TRX2	101.40,102.26,102.04 88.59,87.45,87.99		33	0
	BDF2pred	123.02,113.69,112.92 88.29,87.88,87.64		54	0
	gPROMs	198.922,191.172,202.391		107	-
6 (N=256, M=512) UMFPACK PARDISO	CN	8.68,9.14,8.65 7.79,7.84,7.80	265216	36	0
	IMPTRAP	8.13,9.15,8.17 7.46,7.65,7.31		36	0
	EB	10.35,10.75,10.29 9.21,8.95,8.97		42	6
	Rad	41.33,43.56,41.37 23.48,22.73,22.50		42	4
	TRBDF2	11.77,11.78,11.80 8.67,8.74,8.68		33	0
	TRX2	11.95,11.64,11.89 8.49,8.49,8.49		32	0
	BDF2pred	11.76,11.54,11.51 7.97,7.92,7.96		48	0
	gPROMs	42.64,50.26,54.89		109	-

*The solver parameters for gPROMs *atol* = 1e-6, *rtol* = 1e-5, hmax = 0.05 for all examples except example 5, where hmax = 0.25.

While it is not a fair comparison, MATLAB's ode15i was also used to benchmark the results obtained and the CPU time for different models. MATLAB's ode15i was run with a maximum order of 2 to ensure A-stability and with the same solver parameters for absolute tolerance, relative tolerance, and maximum time step. This was done without providing the sparsity pattern or analytic Jacobian, forcing MATLAB to find the numerical Jacobian. MATLAB's ode15i had similar efficiency (often more efficient) compared to the developed solvers for the same solver parameters, even though it calculates the numerical Jacobian for examples 1-3. The developed solvers are more efficient when the problem size increases to more than 1000 DAEs (examples 4-6). In particular, for example 6, MATLAB's ode15i cannot handle more than 40 × 80 grid points (requiring more than one hour of simulation time). Of course, optimizing the MATLAB code and providing the sparse analytic Jacobian can alleviate this. However, to use the developed DAE solver in Maple, the user has to provide only the DAEs, neither the sparsity pattern nor the analytic Jacobian. MATLAB codes that solve examples 1-6 using ode15i can be obtained upon request from the corresponding author. The table also lists the calculation time with IDA, a BDF solver implemented in C from SUNDIALS. (Hindmarsh et al., 2005) One can see that the default dense matrix linear solver in IDA enables very fast computation for smaller problems. But beyond example 5, the dense matrix algebra failed for this problem.

IDA coupled with sparse linear solver will enable faster computation for examples 6 and 7, but that requires the user to provide the analytic Jacobian and sparse matrix entries. That step is not needed in the newly developed DAE solver in this paper. IDA based on Krylov linear solvers (matrix-free methods) scale well for example 5. It should be noted that example 4 will fail with diagonal preconditioner-based Krylov methods in IDA. So, we report the CPU times only based on IDA's default dense matrix solvers in Table 5.

In addition, all the examples were simulated with gPROMs with their default DASOLV solver settings (except for the solver parameters specified above). The solver uses the backward difference formula with varying time steps and varying order (1 to 5). For large problems (Examples 5 and 6), the developed solver can run the models faster than gPROMs, a commercial solver. Note that though both IDA and gPROMS used the BDF formula, the results from IDA were obtained with a maximum order of 2 (to guarantee A stability). gPROMS was run without adding this restriction.

7 Future work, Perspectives, Code-dissemination, and Summary

In this paper, a sparse DAE solver was developed and implemented in Maple. Some of the aspects of the solver and future work are summarized below:

1. Seven different algorithms were considered and implemented in this paper. For large-scale problems from the discretization of PDEs in 2D, the IMPTRAP method was the most efficient for the tolerance specified. For extremely stiff problems, both Euler-backward and Radau methods might be more efficient. In particular, the Euler-backward method is recommended if unrealistic oscillations are observed. The code finds the analytic Jacobian by differentiating the functions in the model equations. If the function is not differentiable, then the code may not work. The solver has been found to work for some cases involving discontinuities. For example 1, replacing the right-hand side in equation 1 with a piecewise function in *z* as given below works;

$$\frac{d}{dt}y(t) = z(t)\left(\begin{cases} 1 & 0.7 \le z(t) \\ \frac{1}{2} & \textit{otherwise} \end{cases}\right) \tag{27}$$

2. Continuous extension in time was not included in the current version of the solver. Maple's spline functions can be used for this. Our experience suggests that interpolations and continuous extensions are not as accurate and stable for the algebraic variables as the predicted values at the terminals (end-of-time steps) in the algorithms. Also, when a particular spatial discretization is used in a model with singularities (as in example 6), not all the variables will converge at the same order of accuracy, requiring different orders for interpolation and continuous extension for different variables.
3. Implicit DAEs and ODEs (problems with non-constant mass-matrix form) can be addressed by making small algorithm changes, particularly for the Radau method. Adaptive BDF2 was faster than other solvers reported in this paper for many of the examples, but and is available upon request from the authors. It was not included in this paper as the method is not A-stable.
4. Future versions of the solver could include integration with Krylov-type linear solvers and parallel sparse direct solvers (Saad, 2003), event detection, *etc*. Reordering the equations and variables can reduce the bandwidth of the linear solver and reduce the computational time further.
5. Hardware floats and compiled codes will make the codes run faster with minimal memory requirements. This is easily doable based on the Gauss elimination method already implemented in Maple for 100-200 DAEs. However, this will not scale for a large number of DAEs. Providing the

residues and the Jacobian in an efficient format might further speed up the code. This requires minor changes to the code to take the function *F* and Jacobian *Jac* as inputs to the solver.

6. The code is developed and distributed under MIT license without any restrictions. Two versions of the developed DAE solver are provided. The UMFPACK version (the default solver DAESolver.txt) works in any version of Maple installation (after 2020) by directly calling from Maple. The user should have a valid Intel MKL license and the required binaries to use the PARDISO-based DAE solver (Alappat et al., 2020)-(Bollhöfer, Eftekhari, Scheidegger, & Schenk, 2019), DAESolverP.txt which is also directly called from Maple.

The results obtained from various algorithms were compared to those obtained from gPROMs. While gPROMs is faster than the developed DAE solver in terms of CPU time for smaller problems, the developed method performs better for large-scale DAE problems.

8 Code

The Maple code, developed using Maple (2022), for the solvers and the example problems have been posted online and may be accessed through this link: https://sites.utexas.edu/maple/sparsedaesolver/

9 Acknowledgments

Funds from Texas Materials Institute and the University of Texas at Austin's start-up package and past funds from the Department of Energy and multiple industries are gratefully acknowledged. Dr. Michael Baldea's contributions to the discussions that shaped this work are greatly appreciated. Venkat R. Subramanian teaches the adaptive time-stepping procedure implemented in this code as a part of his graduate-level numerical methods course. Inputs from various students at the University of Texas at Austin and the University of Washington who took Venkat R. Subramanian's graduate course on numerical methods are gratefully acknowledged.

Bibliography

Alappat, C., Basermann, A., Bishop, A. R., Fehske, H., Hager, G., Schenk, O., … Wellein, G. (2020). A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric Sparse Matrix-Vector Multiplication. *ACM Trans. Parallel Comput.*, *7*(3).
Ascher, U. (1989). On Symmetric Schemes and Differential-Algebraic Equations. *SIAM Journal on Scientific and Statistical Computing*, *10*(5), 937–949.
Bollhöfer, M., Eftekhari, A., Scheidegger, S., & Schenk, O. (2019). Large-scale Sparse Inverse Covariance Matrix Estimation. *SIAM Journal on Scientific Computing*, *41*(1), A380–A401.
Brown, P. N., Hindmarsh, A. C., & Petzold, L. R. (1994). Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems. *SIAM Journal on Scientific Computing*, *15*(6), 1467–1488.
Butcher, J. C. (2003). *Numerical Methods for Ordinary Differential Equations*.
Cash, J. R., & Considine, S. (1992). An MEBDF Code for Stiff Initial Value Problems. *ACM Transactions on Mathematical Software (TOMS)*, *18*(2), 142–155.
DAEPACK. (n.d.). Retrieved July 25, 2023, from https://yoric.mit.edu/daepack
gPROMS ModelBuilder. (n.d.). Retrieved July 25, 2023, from https://www.psenterprise.com/products/gproms/modelbuilder
Hairer, E., & Nørsett, Syvert P. Wanner, G. (1993). *Solving Ordinary Differential Equations I: Nonstiff Problem*.
Hairer, E., & Wanner, G. (1999). Stiff differential equations solved by Radau methods. *Journal of Computational and Applied Mathematics*, *111*(1–2), 93–111.
Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., & Woodward, C. S. (2005). SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions*

on Mathematical Software, *31*(3), 363–396.
Hosea, M. E., & Shampine, L. L. (1996). Analysis and Implementation of TR-BDF2. *Applied Numerical Mathematics*, *20*, 21–37.
Iserles, A. (1996). *A First Course in the Numerical Analysis of Differential Equations*.
Kennedy, C. A., & Carpenter, M. H. (2016). *Diagonally Implicit Runge-Kutta Methods for Ordinary Differential Equations. A Review*.
Maplesoft Website. (n.d.). Retrieved July 25, 2023, from www.maplesoft.com
Newman, J., & Thomas-Alyea, K. E. (2004). *Electrochemical Systems*.
Petzold, L. R. (1982). A Description of DASSL: A Differential/Algebraic System Solver. *IMACS World Congress, Montreal*, *94550*.
Rackauckas, C., & Nie, Q. (2017). DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software*, *5*(1), 15.
Saad, Y. (2003). Iterative Methods for Sparse Linear Systems. *Iterative Methods for Sparse Linear Systems*.
Van der Pol, B. (1926). LXXXVIII. On “relaxation-oscillations.” *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *2*(11), 978–992.
Vie, H., & Miller, R. A. (1986). Estimation by limiting dilution analysis of human IL 2-secreting T cells: detection of IL 2 produced by single lymphokine-secreting T cells. *Journal of Immunology (Baltimore, Md. : 1950)*, *136*(9), 3292–3297.
Osher, S., & Sethian, J.A. (1988). Fronts Propagating with Curvature Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *Journal of Computational Physics*, 79, 12-49.
Jiang G.-S.& Peng D. (2000). Weighted ENO Schemes for Hamilton--Jacobi Equations. *SIAM Journal on Scientific Computing*, 21(6), 2126.