A STRESS TEST FOR THE MIDPOINT TIME-STEPPING METHOD

JOHN BURKARDT*, WENLONG PEI†, AND CATALIN TRENCHEA‡

Abstract. The midpoint method can be implemented as a sequence of Backward Euler and Forward Euler solves with half time steps, allowing for improved performance of existing solvers for PDEs. We highlight the advantages of this refactorization by considering some specifics of implementation, conservation, error estimation, adaptivity, stability, and performance on several test problems.

1. Introduction. In the paper [9], the midpoint method was reformulated in an unusual way, and numerous properties and advantages of the method and its implementation were claimed. Here, we propose to highlight these features by considering some specifics of implementation, conservation, error estimation, adaptivity, stability, and performance on several test problems. The refactorized method was successfully applied to partial differential equations and partitioning algorithms for fluid-structure interaction and magnetohydrodynamics [6–8, 52], and the idea was further developed for multistep methods in [35, 36].

2. The midpoint method and its “relatives”. We wish to estimate some quantity \( y(t) \), for which we have an initial value \( y_0 \) at time \( t_0 \), and an evolution equation:

\[
y'(t) = f(t, y(t)).
\]

We shall produce estimates \( y_n \) at a discrete sequence of times \( \{t_n\}_{n \geq 0} \), using stepizes \( \tau_n = t_{n+1} - t_n \). For convenience, we define \( t_{n+1/2} = t_n + \frac{1}{2} \tau_n \).

The (implicit) midpoint method for this problem can now be defined by:

\[
\frac{y_{n+1} - y_n}{\tau_n} = f(t_{n+1/2}, \frac{y_{n+1} + y_n}{2}) = f(t_{n+1/2}, y_n + \frac{1}{2} \tau_n \frac{y_{n+1} - y_n}{\tau_n}).
\]

Unless the right hand side function \( f(\cdot, \cdot) \) is linear in \( y \), each time we want to take a step by applying this formula, we must solve an implicit nonlinear equation for the unknown value \( y_{n+1} \). This implicit equation solution cost is part of the overhead of the midpoint method. This cost varies depending on the nonlinearity of \( f(\cdot, \cdot) \) (associated with the problem), on the stepsize being used, (associated with the midpoint method), and on the robustness of the implicit equation solver (depending on the underlying nonlinear solver employed).

It should be noted that a number of ODE solution methods are loosely termed “midpoint methods”. To avoid confusion, it may be helpful to first name several of these related methods and indicate their distinguishing features.

<table>
<thead>
<tr>
<th>Title</th>
<th>Formula for ( \frac{y_{n+1} - y_n}{\tau_n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit midpoint:</td>
<td>( f(t_{n+1/2}, \frac{1}{2} (y_n + y_{n+1})) )</td>
</tr>
<tr>
<td>Explicit midpoint:</td>
<td>( f(t_{n+1/2}, \frac{1}{2} (y_n + y_{n+1} + \tau_n f(t_n, y_n))) )</td>
</tr>
<tr>
<td>Implicit trapezoidal:</td>
<td>( 1/2 (f(t_n, y_n) + f(t_{n+1}, y_{n+1})) )</td>
</tr>
<tr>
<td>Explicit trapezoidal:</td>
<td>( 1/2 (f(t_n, y_n) + f(t_{n+1}, y_n + \tau_n f(t_n, y_n))) )</td>
</tr>
</tbody>
</table>

*Department of Mathematics, University of Pittsburgh, Pittsburgh, PA 15260, USA. Email: jvb25@pitt.edu.
†Department of Mathematics, University of Pittsburgh, Pittsburgh, PA 15260, USA. Email: wep17@pitt.edu. The research herein was partially supported by NSF grant 2110379.
‡Department of Mathematics, University of Pittsburgh, Pittsburgh, PA 15260, USA. Email: trenchea@pitt.edu.
If we move to partial differential equations, a very popular procedure is known as the Crank-Nicolson method, which involves both space and time discretization. For time stepping, it is possible to implement the Crank-Nicolson method using any one of the four above methods. Interestingly, in the original paper [12], it seems that the implicit midpoint method is described.

From now on in this discussion, the expression “midpoint method” will be used exclusively to refer to the implicit midpoint method.

3. Implementation. The midpoint method is a single step method; that is, the approximation of \( y_{n+1} \) at time \( t_{n+1} \) depends on the current values \( t_n \) and \( y_n \), but not on any previous data. For the moment, we will assume that an appropriate stepsize \( \tau_n = t_{n+1} - t_n \) has been specified, so that we only need to address the implicit equation that defines \( y_{n+1} \).

While adaptivity and variable step sizes are a vital feature of modern ODE solvers, we will defer discussion of those matters (see e.g. [7, 9] and the references therein). We begin with the assumption that the desired task is to produce a sequence \( y_i \) of solution estimates at \( n+1 \) time values in the interval \([t_0, t_n]\), with equal spacing \( dt = \frac{t_n - t_0}{n} \). The solution process starts from \((t_0, y_0)\) and repeatedly calls a solver to advance the solution from time \( t_i \) to time \( t_{i+1} \).

A simple version of the midpoint method would use a fixed number \( n \) of steps of size \( dt \) starting at \( \bar{0} \). To handle the implicit nonlinear equation, we invoke a “magical” routine we might call \( \text{SOLUTION}(\text{variable}, \text{equation}) \), which determines a value for \( \text{variable} \) which satisfies the system symbolized by \( \text{equation} \). A sketch of the procedure might be:

```
1 t_1 = t0
2 y_1 = y0
3 for i from 1 to n
4    t_h = t_i + dt / 2
5    y_h = SOLUTION ( y_h, ( y_h − yo ) / ( t_h − t_o ) − f ( th, yh ) == 0 )
6    t_i+1 = t_i + dt
7    y_i+1 = 2 * yh − yo
8 for i = 1 : n
9    to = t(i,1);
10   yo = y(i,:);
11   th = to + 0.5 * dt;
12   yh = yo + 0.5 * dt * f ( to, yo );
13   yh = fsolve ( @(yh) backward_euler_residual(f, to, yo, th, yh), yh );
14   tp = to + dt;
15   yp = 2.0 * yh − yo;
16   t(i+1,1) = tp;
17   y(i+1,:) = yp;
```

Since MATLAB [51] provides a procedure \( \text{fsolve} \) which does exactly the nonlinear solving that we need, we can demonstrate a working version of this calculation:

```
for i = 1 : n
    to = t(i,1);
    yo = y(i,:);
    th = to + 0.5 * dt;
    yh = yo + 0.5 * dt * f ( to, yo );
    yh = fsolve ( @(yh) backward_euler_residual(f, to, yo, th, yh), yh );
    tp = to + dt;
    yp = 2.0 * yh − yo;
    t(i+1,1) = tp;
    y(i+1,:) = yp;
```

There is some confusion on what the Crank-Nicolson method actually is. For example, the wikipedia page (November 14, 2021) claims that Crank-Nicolson is based on the trapezoidal rule, and also Hundsorfer and Verwer [28, page 125] state that “In the classic numerical PDE literature, backward Euler and the trapezoidal rule are also known under the names Laasonen scheme [31] and Crank-Nicolson [12] scheme, respectively”. The Crank-Nicolson appellation for the trapezoidal rule is used in Ascher [2, page 41], Canuto, Hussaini, Quarteroni and Zhang’s book [10, page 521], Hairer, Lubich and Wanner [23, page 28], Leveque [38, pages 121, 185], Quarteroni, Sacco and Saleri [44, page 483], Quarteroni, Valli [45, page 149], Volker [29, page 394], Heywood and Rannacher [25, page 355] and Gunzburger [22, page 131] call trapezoidal rule as Crank-Nicolson, and then use the midpoint rule. In [34, page 162], Layton also refers to the midpoint rule by the trapezoidal cognomen. On the other hand, Glowinski [21, page 267], Hoffman and Johnson [27, page 216], Kalnay [30, page 83], Layton and Rebholz [37, page 137], refer to the midpoint rule as the Crank-Nicolson method. Finally, it is worth mentioning that the method which John Crank and Phyllis Nicolson literally used in their 1947 paper [12] on numerical solutions of a nonlinear partial differential equation for heat flow is the midpoint rule.
Here, the first argument of fsolve() identifies the variable to be manipulated, and the user procedure that defines the system; the second argument is the current value of the variable. Note that a similar procedure, also called fsolve() is available in Octave [17], Python [43], and R [46], and so the implicit midpoint method can easily be implemented in those languages as well.

The implicit midpoint method can also be implemented in the finite element FreeFem++ [24]. Suppose that we are solving the heat equation in a square, with a source function \( f(x, y, t) \). Our generic finite element test function will be designated \( \psi \). For time \( t_{old} \), we have computed an approximate solution \( u_{old} \), and we now want to advance to time \( t = t_{old} + dt \) and compute \( u \). Inside the time loop, we simply state:

\[
\text{tmid} = t_{old} + dt / 2.0; \\
// At time tmid, take a backward Euler step of size dt/2. \\
// halfbackwardeulerstep; \\
// Follow this with a forward Euler step from time tmid of size dt/2. \\
// u = 2.0 * umid - uold; \\
\]

The function halfbackwardeulerstep() is responsible for finding a solution \( umid \) of the implicit equation posed by the midpoint formulation:

\[
\text{problem halfbackwardeulerstep ( umid, vh ) =} \\
\text{int2d ( Th ) ( umid * vh ) - int2d ( Th ) ( uold * vh )} \\
\text{+ int2d ( Th ) ( 0.5 * dt * ( dx ( umid ) * dx ( vh ) + dy ( umid ) * dy ( vh ) ) )} \\
\text{- int2d ( Th ) ( 0.5 * dt * f(x,y,tmid) * vh) + on ( 1, 2, 3, 4, umid = 0.0 );} \\
\]

The details of the nonlinear solution process are handled by the FreeFem++ package.

4. Sample Problems. Subsequent investigations will focus on a number of test differential equations, written here as first order systems. The independent variable is written as \( t \), the unknowns are generally \( u, v, w \), and other symbols represent problem parameters.

<table>
<thead>
<tr>
<th>Title</th>
<th>ODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential:</td>
<td>( u'' = \lambda u )</td>
</tr>
<tr>
<td>Stiff:</td>
<td>( u' = \lambda (\cos(t) - u) )</td>
</tr>
<tr>
<td>Lotka-Volterra:</td>
<td>( u'' = 2u - 0.001luv )</td>
</tr>
<tr>
<td></td>
<td>( v' = -10uv + 0.002uv )</td>
</tr>
<tr>
<td>Rigid body:</td>
<td>( u' = (1/c - 1/b)v )</td>
</tr>
<tr>
<td></td>
<td>( v' = (1/a - 1/c)uw )</td>
</tr>
<tr>
<td></td>
<td>( w' = (1/b - 1/a)uv )</td>
</tr>
<tr>
<td>Vanderpol:</td>
<td>( u'' = v )</td>
</tr>
<tr>
<td></td>
<td>( v' = \mu(1 - u^2)v - u )</td>
</tr>
<tr>
<td>Nonlinear pendulum:</td>
<td>( u' = v )</td>
</tr>
<tr>
<td></td>
<td>( v' = -\frac{g}{l} \sin(u) )</td>
</tr>
<tr>
<td>Double pendulum:</td>
<td>( u''_1 = v_1 )</td>
</tr>
<tr>
<td></td>
<td>( v''_1 = \frac{g(2m_1+m_2)(\sin(u_1)+m_2(\sin(u_1-2\alpha_2)+2(2v_2+v_1^2+l_1^2\cos(u_1-u_2))\sin(u_1-u_2)) \sin(u_1-u_2))}{2l_1(m_1+m_2-m_2\cos(u_1-u_2)^2)} )</td>
</tr>
<tr>
<td></td>
<td>( u''_2 = v_2 )</td>
</tr>
<tr>
<td></td>
<td>( v''_2 = \frac{((m_1+m_2)((l_1\gamma_1^2+g\cos(u_1))+l_2m_2\cos(u_1-u_2))\sin(u_1-u_2))}{l_2(m_1+m_2-m_2\cos(u_1-u_2)^2)} )</td>
</tr>
</tbody>
</table>

Table 4.1: Definitions of the test differential equations.

5. Sample Problem Data. Aside from the differential equations, each test problem has a typical interval over which it is solved, initial conditions for the unknowns, and one or more suggested values for the problem parameters.
6. Conservation. As systems evolve over time, there may be certain properties whose values stay fixed. These are known as invariants or conserved quantities. Mechanical systems in particular may undergo violent changes and yet conserve mass, momentum, and energy. When the behavior of such a system is to be approximated by an ODE solver, the true solution is generally unknown. Then monitoring the conservation behavior may be a primary means of determining the quality of an approximate solution. An approximation that does not satisfy conservation might be regarded as physically dubious or even meaningless. Common conservation constraints involve the state variables linearly, as in mass conservation, or quadratically, as in energy conservation (for example, in magnetohydrodynamics, there are several quadratic conserved Hamiltonians - the kinetic energy, the magnetic helicity and cross helicity [32,33,52]); more complicated relations are less common. The constraints are sometimes thought of as defining the surface of a manifold, upon which any exact solution curve must lie.

Although the solutions returned by a differential equation solver are approximations, they may nonetheless closely or exactly satisfy the conservation constraints. The behavior of many physical systems can be described in terms of Hamilton’s equations. These equations are written in terms of a Hamiltonian function, which can be regarded as an energy measure. In the absence of outside influences or dissipation, the system’s Hamiltonian function should have a constant value over time. In many cases, it is possible to construct ODE solvers which exactly conserve a slightly perturbed value of the Hamiltonian function over all future time. Such solvers are known as symplectic integrators. They are of great use in the study of mechanics, planetary motion, molecular dynamics, and quantum physics. The symplectic Euler method and the Verlet method are two popular versions of this approach.

Symplectic methods are restricted to a particular class of differential equations. However, the ability to conserve certain system quantities for a wider range of ODEs is highly desirable. A number of integration methods have been identified which do preserve certain invariants. Such an ODE solver is known as a geometric integrator [23].

Many ODE solvers can satisfy a linear conservation constraint. However, the midpoint method is the only single step method which also satisfies quadratic conservation constraints [5]. This greater power is notable, since most expressions of energy involve quadratic terms.

It is worth repeating that for typical real-life physical systems, if no exact solution is known, conservation constraints are a useful check on the solution. Moreover, a sort of stress test can be performed on any ODE solver by observing the extent to which it can preserve quantities that should be conserved. Exactly this sort of test will be reported for the midpoint method, as well as Euler, backward Euler, RK4, and and several of the standard MATLAB ODE solvers [48]. Several of the sample problems have an associated conserved quantity, as listed in Table 6.1. These cases will be used to explore the conservation properties of the midpoint method, and to compare its performance to some other ODE solvers.

Because the rigid body ODE constraint is quadratic, we expect the midpoint method to do an excellent con-

<table>
<thead>
<tr>
<th>Title</th>
<th>Interval</th>
<th>Initial</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential:</td>
<td>$0 \leq t \leq 5$</td>
<td>$u(0) = 1$</td>
<td>$\lambda = -1, -10, -100$</td>
</tr>
<tr>
<td>Stiff:</td>
<td>$0 \leq t \leq 1$</td>
<td>$u(0) = 0$</td>
<td>$\lambda = 1, 10, 50$</td>
</tr>
<tr>
<td>Lotka-Volterra:</td>
<td>$0 \leq t \leq 10$</td>
<td>$u(0) = 5000$</td>
<td>$v(0) = 100$</td>
</tr>
<tr>
<td>Rigid body:</td>
<td>$0 \leq t \leq 50$</td>
<td>$u(0) = \cos(0.9)$</td>
<td>$v(0) = 0$</td>
</tr>
<tr>
<td>Vanderpol:</td>
<td>$0 \leq t \leq 20$</td>
<td>$u(0) = 0.05$</td>
<td>$v(0) = 0.05$</td>
</tr>
<tr>
<td>Nonlinear pendulum</td>
<td>$0 \leq t \leq 50$</td>
<td>$u(0) = 0.99\pi$</td>
<td>$v(0) = 0$</td>
</tr>
<tr>
<td>Double pendulum</td>
<td>$0 \leq t \leq 50$</td>
<td>$u_1(0) = 0.25$</td>
<td>$v_1(0) = 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$u_2(0) = 0$</td>
<td>$v_2(0) = 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$l_1 = 2$</td>
<td>$l_2 = 1$</td>
</tr>
</tbody>
</table>

Table 5.1: Data associated with the test problems.
Conserved Quantity

<table>
<thead>
<tr>
<th>Title</th>
<th>Conserved Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rigid Body:</td>
<td>$h_1(u,v,w) = u^2 + v^2 + w^2$</td>
</tr>
<tr>
<td></td>
<td>$h_2(u,v,w) = u^2/a + v^2/b + w^2/c$</td>
</tr>
<tr>
<td>Lotka-Volterra:</td>
<td>$h(u,v) = \delta u - \gamma \log(u) + \beta v - \alpha \log(v)$</td>
</tr>
<tr>
<td>Nonlinear Pendulum:</td>
<td>$h(u,v) = \frac{2m}{l^2}(1 - \cos(u)) + \frac{m^2}{l^2}$</td>
</tr>
<tr>
<td>Double Pendulum:</td>
<td>$h(u_1, v_1, u_2, v_2) = 0.5(m_1 + m_2)l_1^2v_1^2 + 0.5m_1l_2^2v_2^2 + m_2l_1l_2v_1v_2 \cos(u_1 - u_2) - (m_1 + m_2)gl_1 \cos(u_1) - m_2gl_2 \cos(u_2)$</td>
</tr>
</tbody>
</table>

The conserved quantities for some sample problems.

6.1. Rigid Body Conservation. The rigid body problem [40, 47, 50] represents the Euler equations for a rigid body, with given moments of inertia, undergoing rotation. There are two invariants, $h_1$ and $h_2$. Because these two are quite similar in formula and behavior, we will focus solely on $h_1$. The initial condition is chosen to correspond to $h_1 = 1$. The left hand plot in Figure 6.1 shows our first conservation test, which runs the evolution over the relatively short time interval $0 \leq t \leq 50$. We run backward_euler() and euler() with $n = 1000$ steps, midpoint() with $n = 100, n = 1000$ steps, midpoint() with $n = 100$, and let ode45() adaptively chooses $n = 137$ steps.

We repeat the test over the more challenging interval $0 \leq t \leq 10,000$. Based on their poor performance in the first test, we eliminate the backward Euler and Euler “contestants”, and bring in another MATLAB solver ode23() and rk4(), a fixed stepsize fourth order Runge Kutta solver. The results of this comparison are on the right hand plot in Figure 6.1.

The two rigid body conservation plots show that, even over a relatively short time interval, both backward_euler() and euler() do a poor conservation job. In fact, these methods have a reputation for “losing” or “gaining” energy, respectively, which is confirmed by the plot. Over the longer time interval, both ode23() and ode45() fail the conservation test significantly. The rk4() procedure has a small but visible deviation from the constant value. On the other hand, midpoint() seems to do an excellent job of conservation. We note that midpoint() and rk4() were assigned a fixed number of steps $n = 20,000$, while ode23() and ode45() adaptively chose $n = 21,266$ and $n = 24,909$ steps of varying size.

6.2. Lotka Volterra Conservation. The Lotka Volterra system is a predator-prey model, with a stable equilibrium point [16, 41]. Depending on the initial condition, the resulting solution follows an orbit about the equilibrium point. There can be sharp curves in the orbit shape when one of the variables is near zero; an ODE solver that does
not round such a curve accurately can gradually wander away from the correct orbit. The conserved quantity is not simply the total population, but is expressed in a more complicated formula involving logarithms (see Table 6.1). Hence it is not of the quadratic type for which the midpoint method can promise exact conservation.

We begin with a short time interval, $0 \leq t \leq 10$. The backward euler() and euler() methods use $n = 1000$ steps, midpoint() with $n = 100$, and ode45() adaptively chooses $n = 137$ steps. The left hand plot in Figure 6.2 illustrates this test.

We repeat the test over the more challenging interval $0 \leq t \leq 2,000$. The MATLAB solver ode23() was originally one of the “contestants”, but could not complete the calculation in a reasonable time. Therefore, it was replaced by ode23s(), a related low-order method that is suitable for stiff problems. The results of this comparison are on the middle plot in Figure 6.2.

As in the previous conservation test, backward euler() and euler() do poorly, with the first “losing” and the second “gaining” energy, while midpoint() and ode45() seem to be on target. The longer time interval shows that this problem is more challenging. All the methods tested show deviation, with ode23s() far out. Although it is scarcely perceptible in the plot, midpoint(), ode45() and rkf() all seem to be losing energy, at roughly the same rate. This becomes clear in a close up of the results, displayed in the right hand plot in Figure 6.2.

The thickness of the line for the midpoint() results actually indicates that the computed result aside from its gradual decline, also has an oscillation in value whose amplitude is too low to be clearly seen. Meanwhile, rk4() seems to be on a corresponding gradual increasing energy curve. The ode45() results deviate, after a few sudden jolts, and then seem to level off. Overall, these results show both that the problem is difficult (sharp turns, non-quadratic conservation law) and that the better ODE methods perform relatively well, though not perfectly.

6.3. Nonlinear Pendulum Conservation. While the conservation quantity for the linear pendulum is quadratic in the unknowns, the more nonlinear pendulum [4, 42] involves the cosine function. Moreover, the behavior of the nonlinear system deviates markedly from the linear system as its energy is increased; in particular, if the energy approaches the amount sufficient for the pendulum to reach the full upright position, the period of the system increases without limit. For our tests, we chose to challenge the system by imposing an initial energy that was 0.9998% of the flip-over energy.

We begin with a short time interval, $0 \leq t \leq 50$. The backward euler() and euler() methods use $n = 5000$ steps, midpoint() with $n = 700$, and ode45() adaptively chooses $n = 709$ steps. The results are plotted in the left hand of Figure 6.3.

We repeat the test over the more challenging interval $0 \leq t \leq 2,000$, replacing backward euler() and euler() with ode23s() and rk4(), with the results in the right hand of Figure 6.3.

For the short time interval test, the Euler routines again diverge up and down, while midpoint() and ode45() are indistinguishable from the exact constant line. The long time test, however, shows ode45() taking a steep path to a spurious, seemingly stable solution of much higher energy, while midpoint() remains fixed on the exact energy. The rk4() solver shows a gradual decrease in energy, while ode23s() takes a strong drop and then levels off.

The conservation quantity for this problem is not quadratic. Hence the midpoint() method cannot expect perfect conservation, and yet it seems to do a remarkably good job, particularly in comparison to the other solvers.

6.4. Double Pendulum Conservation. The double pendulum problem [1] raises the difficulty level by modeling a system in which one nonlinear pendulum is attached to another, and the system is given an initial motion.
The mass and lengths of both pendulums are parameters that can be varied. At low energy, the system is somewhat similar to the nonlinear pendulum, but given sufficient energy, a variety of “flip overs” can be observed, and the behavior of the system approaches chaos. Nonetheless, the system conserves energy, so even if the exact solution is not known, an ODE solver can be judged at least by how well it keeps the energy constant.

Using our more reliable solvers, we will consider three tests, involving an increasing initial energy. These results are displayed in Figure 6.4.

The first test, at an energy level $h \approx -47.8$, shows all four solvers tracking the energy very well. In the second test, with $h \approx -37.1$, the $\text{ode23s()}$ energy begins to decay, but the $\text{ode45()}$ energy rises dramatically. For the final test, at $h \approx -27.6$. Initially, the $\text{ode45()}$ energy explodes, $\text{ode23s()}$ and $\text{rk4()}$ are both losing energy, but $\text{midpoint()}$ steadily tracks the original energy value.

6.5. Conservation Overview. Of course, the true goal of an ODE solver is direct estimation of the solution variable. The conservation quantity is only a projection of the solution; but it is a computable, testable value, whereas, except in academic problems, the exact solution to the ODE is unknown. A poor conservation result may be our principal warning that the ODE solution is untrustworthy. While higher degree ODE solvers may promise accurate solutions at a lower cost, we may have no independent checks to apply if we cannot rely on conservation. Hence, the midpoint method can be recommended as doing a generally trustworthy treatment of the more common conservation quantities.

7. Error Estimation. The fact that the midpoint method is (nonlinearly) B-stable and (linearly) A-stable is of paramount importance for accurate computations. As the midpoint method’s region of absolute stability is exactly the left half-plane, and the region of instability is the right half-plane, it is ideal for both stable and unstable problems. For example, in [54], Watanabe points out that a certain one-leg $k$-steps method, with a region of instability containing a small portion of the left half-plane, even with an adaptive algorithm, yields inaccurate solutions. Namely, for ‘a system where the eigenvalues have large imaginary components and small negative real components, the solution would grow rather than decay as it should.’
The other major issue mentioned in [54], in relation with stiff and unstable problems, is the importance of error estimators and tolerances: ‘the detection of positive eigenvalues is sensitive to the error tolerance and the details of the Newton iteration’. The refactorization of the midpoint method in [9] proved that the correct estimator for the local truncation error is the differentiation defect, and it does not involve the interpolation defect [14]. The midpoint method error constant was shown [9] to be \( \frac{1}{24} \), compared to the trapezoidal method’s error constant \( -\frac{1}{12} \), given by Dahlquist’s first barrier [13]. Moreover, the midpoint method’s error constant is the smallest among all variable-step G-stable one-leg second-order accurate multistep methods [36].

In [54] Watanabe compared the performance of two time-adaptive codes on the Lindberg problem [39] from the stability and error estimator viewpoints. ‘The stability properties of the one-leg formulas are thus not ideal, but these formulas are more likely to warn the user of possible difficulties when the eigenvalues approach the right half-plane’. Also, ‘we have observed that decreasing the error tolerance, evaluating the Jacobian by numerical differentiation, or increasing the number of Newton iterations can delay the growth of the computed solution’. The Lindberg example is a four component stiff ODE system, in which eigenvalues change from large negative values \( 10^{-4} \) to large positive values \( 10^{4} \) as \( t \) increases. Even for local error tolerances of order \( 1.0e-12 \), one of the algorithms in [54] failed to detect the presence of large positive eigenvalues and produced decaying solutions.

The precision of an error estimator is paramount for the stability and overall performance of adaptive algorithms [18, 19, 26]. We tested our error estimator and the adaptive midpoint method [6, 9, 36] on the Lindberg example. In Figure 7.1 we compare the solutions of \texttt{ode45()} and the stiff solver \texttt{ode23s()}, both with relative tolerance \( 10^{-10} \) and absolute tolerance \( 10^{-15} \), versus the adaptive midpoint method with tolerance \( 10^{-14} \), and the exact solution. Neither \texttt{ode45()} nor \texttt{ode23s()} diagnose the presence of large positive eigenvalues, giving the incorrect zero solutions. The adaptive midpoint with a comparable tolerance for the absolute error estimator notices the change in the eigenvalues and stays close to the exact solution.

8. Adaptive Stepsize. We compare the solutions obtained with the constant stepsize (‘midpoint’) versus the adaptive midpoint (‘midpointVS’) methods, for the Van Der Pol’s equation \( x' = y, y' = \mu(1-x^2)y - x \), where \( x(0) = 2, y(0) = 0 \), and \( \mu = 1000 \). This problem is considered to be very stiff (see e.g., [3, 11, 23, 53]) when the parameter \( \mu \) is this big. Both solutions are computed with the same number of time steps, and the adaptive algorithm is the one described in [9], with an absolute tolerance of \( 1.0e-6 \). The local truncation error was evaluated by the differentiation defect [9], by comparing the midpoint and the Adams-Bashforth 2 solutions. The plots in Figure 8.1 also show the solutions obtained with the \texttt{ode45()} and \texttt{ode23s()} algorithms, which overlap with the adaptive midpoint solution. The number of \texttt{ode45()} steps is of the same order, only slightly larger than the number of the midpoint steps. Although the constant step algorithm computes correctly the amplitude of the solution, there is an increasing error in the phase.
9. Stability. Dahlquist et al. used in [49] a simple ODE example, from Stetter [15, pp. 181-182], to argue in favor of one-leg multistep methods versus linear multistep methods, when considering variable time steps. The equation is non-autonomous $y' = \lambda(t)y$, with $\Re(\lambda(t)) \leq 0$, and therefore is stable. Nonetheless, with the particular choice of time steps $h_{2m} = 7, h_{2m+1} = \frac{1}{2}$, and with the values of $\lambda(t)$ being such that $\lambda(t_{2m}) = 0, \lambda(t_{2m+1}) = -1$, the trapezoidal method is unstable resulting in $y_{2m} = (-2)^m y_0$, while the midpoint method, with the same choice of time steps is stable $|y_{m+1}| \leq |y_m|$. This is illustrated in Figure 9.1.

10. Conclusion. In [9], a number of claims were made about the implicit midpoint method. This discussion provides concrete implementations, example problems, and numerical results that back up those claims.

One important assertion in [9] was on the unconditional A- and B-stability of the midpoint method, which is a critical ingredient in the convergence of any numerical method. This means that the midpoint method, with constant or with variable time steps, is both linearly and nonlinearly stable. In Section 9 we showed on the Stetter example [15] that the variable step trapezoidal method is unstable, while the midpoint method is stable.

The midpoint method is a symplectic method for general Hamiltonian systems, conserving all quadratic Hamiltonians. In Section 6 we compared the performance of the constant step midpoint method versus the backward- and forward-Euler, Runge-Kutta 4 method, and also Matlab’s adaptive algorithms ode45, ode23, and the stiff solver ode23s. The test problems were the rigid body problem, the Lotka-Volterra predator-prey model, the nonlinear simple pendulum, and the double pendulum.

The refactorization of the midpoint method in [9] proved that the correct estimator for the local truncation error
is only the differentiation defect, and does not involve the interpolation defect [14]. In Section 7 we evaluated the importance of a performant error estimator for adaptive algorithms [18,19,26]. We tested the variable-step midpoint method with tolerance $10^{-14}$ versus ode45 and ode23s, both with relative tolerance $10^{-15}$ and absolute tolerance $10^{-15}$ on the Lindberg example [39]. A comparison of constant versus adaptive step versions of the midpoint method is provided in Section 8, on a stiff case of the Van Der Pol’s equation.

11. Source Code. Source code for the example problems and some of the tests presented is available in subdirectories of https://people.sc.fsu.edu/~jburkardt/. In particular, going to the indicated web page will enable you to download the source code for the midpoint method, and to navigate to associated test codes, results, and plots:

- Freefem++: /freefem_src/midpoint/midpoint.html
- MATLAB: /m_src/midpoint/midpoint.html
- Octave: /octave_src/midpoint/midpoint.html
- Python: /py_src/midpoint/midpoint.html
- R: /r_src/midpoint/midpoint.html

The “free software” GNU Scientific Library (GSL) [20] includes an implementation of the implicit midpoint method, which they term the “implicit Gaussian second order Runge Kutta method”. The algorithm is accessible under the name gsl_odeiv2_step_rk2imp. An example applied to the van der Pol equation is available at /c_src/midpoint_gsl_test/midpoint_gsl_test.html.

Acknowledgments. Wenlong Pei was partially supported by the NSF grant 2110379.

REFERENCES


