A Midpoint Stress Test:

Catalin Trenchea, Wenlong Pei, John Burkardt

Secret Planning Conference
28 October 2023
Denver, Colorado
https://people.sc.fsu.edu/~jburkardt/presentations/midpoint_2023_denver.pdf

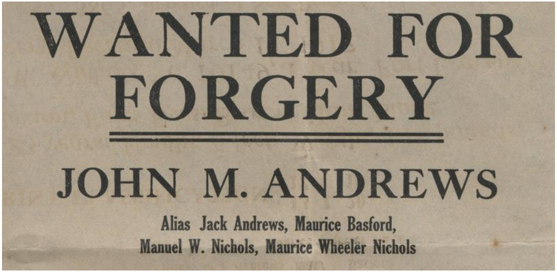
References

Catalin Trenchea, John Burkardt,
Refactorization of the midpoint rule,
Applied Mathematics Letters,Volume 107, September 2020,

John Burkardt, Wenlong Pei, Catalin Trenchea,
A stress test for the midpoint time-stepping method,
International Journal of Numerical Analysis and Modeling,
Volume 19, Number 2-3, pages 299-314, 2022.

William Layton, Wenlong Pei, Catalin Trenchea,
Time step adaptivity in the method of Dahlquist, Liniger, and Nevanlinna,
Advances in Computational Science and Engineering,
Volume 1, Number 3, pages 320-350, September 2023.

Which midpoint method are we talking about?



WANTED FOR FORGERY

JOHN M. ANDREWS

Alias Jack Andrews, Maurice Basford,
Manuel W. Nichols, Maurice Wheeler Nichols

	f(mid)	$\frac{1}{2}$(f(left)+f(right))
explicit:	explicit midpoint method	Heun's method or Improved Euler method
implicit:	implicit Runge-Kutta 2 or **implicit midpoint method**	trapezoidal method

For linear problems, explicit and implicit are the same.

The original Crank-Nicolson uses implicit midpoint time stepping.

Here we consider only the *implicit midpoint method.*

Two Ways to Look at It

The implicit midpoint method can be seen as a single step:

$$y_{n+1} = y_n + \tau_n f(t_{n+1/2}, y_{n+1/2})$$

or as Backward Euler and Forward Euler (BEFE) steps of size $\frac{\tau_n}{2}$:

$$y_{n+1/2} = y_n + \frac{\tau_n}{2} f(t_{n+1/2}, y_{n+1/2}) \quad \text{(Backward)}$$

$$y_{n+1} = y_{n+1/2} + \frac{\tau_n}{2} f(t_{n+1/2}, y_{n+1/2}) \quad \text{(Forward)}$$

The second step can be rewritten simply as:

$$y_{n+1} = 2\, y_{n+1/2} - y_n \quad \text{(Forward)}$$

so the implicit problem only needs to be solved once, in BE.

Sketch of an implementation

Using *n* steps of size *dt* starting at t_0, we invoke a “magic” function *SOLVER(variable,equation)*, which determines a value for y_h:

```
for i from 1 to n
{
  to = t(i)
  yo = y(i,:)

  th = to + dt / 2
  yh = SOLVER ( yh,
    ( yh - yo ) / ( th - to) - f ( th, yh ) == 0 )

  t(i+1) = to + dt
  y(i+1,:) = 2 * yh - yo
}
```

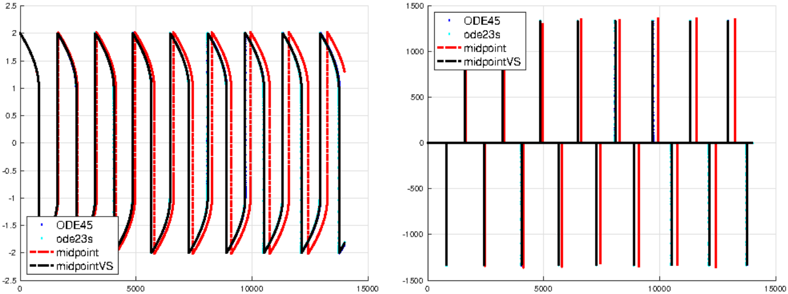
In MATLAB, Octave, Python and R, the magic function is *fsolve()*.

Stress Tests

Exp:	$u' = \lambda u$
Stiff:	$u' = \lambda(\cos(t) - u)$
Lotka:	$u' = 2u - 0.001uv, \quad v' = -10v + 0.002uv$
Rigid:	$u' = (1/c - 1/b)vw$ $v' = (1/a - 1/c)uw$ $w' = (1/b - 1/a)uv$
VDP:	$u' = v, \quad v' = \mu(1 - u^2)v - u$
Pend:	$u' = v, \quad v' = -\frac{g}{l}\sin(u)$
Double pend:	$u_1' = v_1$ $v_1' = \frac{g(2m_1+m_2)\sin(u_1)+m_2(g\sin(u_1-2u_2)+2(l_2+v_2^2+l_1v_1^2\cos(u_1-u_2)\sin(u_1-u_2)}{2l_1(m_1+m_2-m_2\cos(u_1-u_2)^2}$ $u_2' = v_2$ $v_2' = \frac{((m_1+m_2)(l_1v_1^2+g\cos(u_1))+l_2m_2v_2^2\cos(u_1-u_2))\sin(u_1-u_2)}{l_2(m_1+m_2-m_2\cos(u_1-u_2)^2)}$
Lindberg	$y_1' = 10^4(y_1y_3 + y_2y_4)$ $y_2' = -10^4(y_1y_4 + y_2y_3)$ $y_3' = 1 - y_3$ $y_4' = -0.5y_3 - y_4 + 0.5$

Very stiff Van der Pol ODE

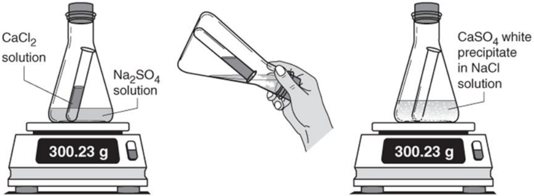
Compare ode45(), ode23s(), fixed step midpoint, adaptive step midpoint, on Van der Pol ODE with $\mu = 1000$.



Conservation

In practical problems, exact solutions are not known, but physical constraints may require conservation of certain quantities. An ODE solver's results can then be judged by how well conservation is modeled.

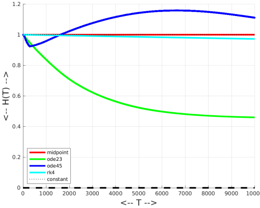
Conservation of mass

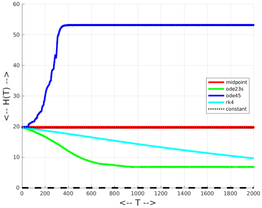


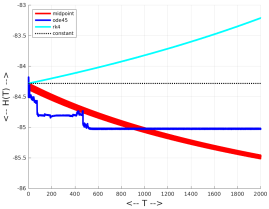
mass (g) of reactants = mass (g) of products

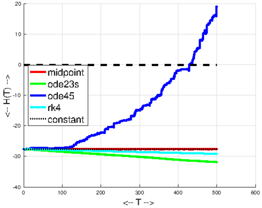
Energy conservation for four test cases

rigid body	nonlinear pendulum
lotka-volterra	double pendulum



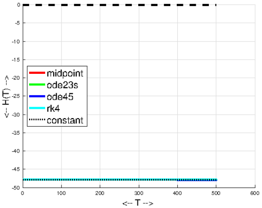


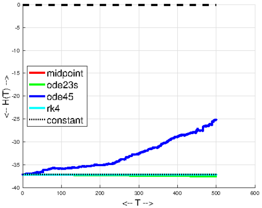


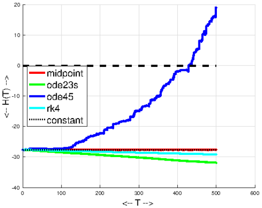


Review energy conservation for double pendulum

Energy conservation(?) for midpoint, ode23s(), ode45(), rk4(), on double pendulum, using increasing energy levels.







ode45() explodes, ode23s() and rk4() lose energy, midpoint() conserves.

Adaptive Stepsize for Midpoint

For a smooth solution $y(x)$, the local truncation error for the midpoint method is

$$T_{n+1} \equiv y(t_{n+1}) - y_{n+1} = \frac{1}{24}\,\tau_n^3\, y'''(t_n + 1/2) + \mathcal{O}(\tau_n^5)$$

For a given local error tolerance `tol`, propose the next time step as

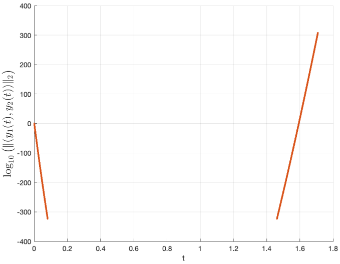
$$\tau_{n+1} = \kappa\, \tau_n \big(\frac{\text{tol}}{||\, T_{n+1}||}\big)^{\frac{1}{3}}$$

with the safety factor $\kappa \le 1$.

The Lindberg ODE: a Deadly Trap

- Lindberg proposed a system of 4 ODE's as a severe test for stiff ODE solvers.
- The system has been criticized as atypical, and "unfair"(!)
- Standard methods fail catastrophically with underflow or overflow.
- The eigenvalues of the Jacobian are large, and evolve over time from negative to positive values.
- This ODE was used to evaluate the DIFSUB and DIFSOL solvers.
- An exact solution is known.

The Lindberg ODE: $\log_{10}(y_1(t), y_2(t))$

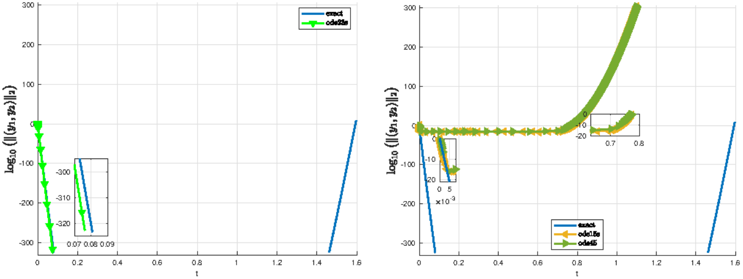


The norm of the exact solution, although nonzero, becomes too small to represent in 64 bit floating point.

We see an initial plunge in the solution, a gap in the range $0.1 \leq t \leq 1.5$, and a final unstable explosion.

An ODE solver will miss this final growth phase near $t = 1.5$ if it has set $(y_1, y_2) = (0, 0)$ by then.

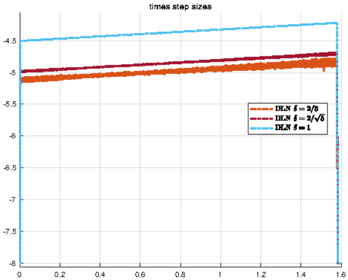
The Lindberg ODE: MATLAB solvers fail

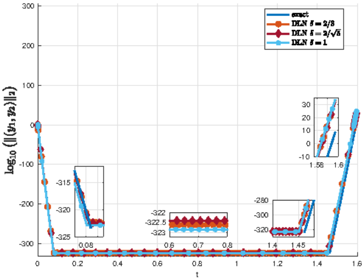


- Left: The ode23s() solver drops too quickly, and “dies” at $t = 1.1$.
- Right: The ode15s() and ode45() solvers drop only slightly, then proceed to about $t = 0.8$ and and explode.

Tolerances used were `abstol = 1.0E-15, reltol = 1.0E-11`.

The Lindberg ODE: Midpoint adaptivity crucial



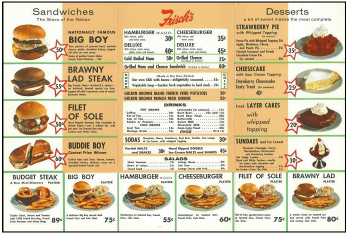


Focus on light blue lines with $\delta = 1$:

- Left: Stepsize drastically reduced at blowup time.
- Right: Adaptive code follows blowup.

Note that a nonadaptive midpoint method, using a constant stepsize, doesn't break down early as the MATLAB solvers do, but marches past the blowup point without detecting it.

A Menu of Implementations



Language	fsolve	adaptive
C	midpoint.c	
C++	midpoint.cpp	
Fortran77	midpoint.f	
Fortran90	midpoint.f90	
FreeFem++	midpoint.edp	
MATLAB	midpoint.m	midpoint_adaptive.m
Octave	midpoint.m	midpoint_adaptive.m
Python	midpoint.py	midpoint_adaptive.py
R	midpoint.R	

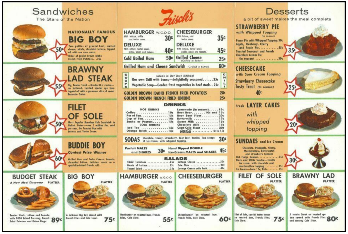
Professional Codes available

Or you may prefer your ODE to be handled by a professional!

Language	library	code
C	Gnu Scientific Library	gsl_odeiv2_step_rk2imp()
C++	Gnu Scientific Library	gsl_odeiv2_step_rk2imp()
Julia	ODE	Midpoint

You might also find an implementation of the midpoint method “hiding” in a standard library as an order 2 implicit Runge Kutta ODE solvers.

A Menu of Implementations



Language	fixed point	fsolve	adaptive
C	midpoint_fixed.c	midpoint.c	gsl_odeiv2_step_rk2imp()
C++	midpoint_fixed.cpp	midpoint.cpp	gsl_odeiv2_step_rk2imp()
Fortran77	midpoint_fixed.f	midpoint.f	
Fortran90	midpoint_fixed.f90	midpoint.f90	
FreeFem		midpoint.edp	
Julia			ODE:Midpoint()
MATLAB	midpoint_fixed.m	midpoint.m	midpoint_adaptive.m
Octave	midpoint_fixed.m	midpoint.m	midpoint_adaptive.m
Python	midpoint_fixed.py	midpoint.py	midpoint_adaptive.py
R	midpoint_fixed.R	midpoint.R	

The Story in a Nutshell

- Implicit backward step/2 + explicit forward step/2;
- It is second order accurate;
- It is absolutely stable and B-stable;
- It preserves linear and quadratic conservation quantities;
- It produces reliable error estimates;
- It predicts safe time-steps for adaptive solution;
- It upgrades Backward Euler codes with one new line;
- C, C++, Fortran, FreeFem++, Julia, MATLAB, Octave, Python, R.