Comparison of a semi-implicit Runge-Kutta method with MATLAB solvers for stiff Ordinary Differential Equations

06-606: Computational Methods for Large Scale Process Design

Course Project

Jay Shah - jrshah@andrew.cmu.edu

1 Stiff ODE systems

Some mathematically well behaved systems of Ordinary Differential Equations cannot be solved with traditional algorithms without the use of extremely small step sizes. Such problems are characterized by the presence of terms that can lead to rapid variation in the solutions i.e. terms that are on widely varying time scales. Such problems occur frequently is areas such as reactor kinetics, distillation column modeling and atmospheric modeling. Equation systems exhibiting such behavior are called "stiff".

To solve a stiff system with reasonable accuracy, the initial value algorithm used must exhibit A-stable behavior or strongly A-stable behavior. Thus explicit RK methods cannot be used because of their restricted absolute stability regions. Here, a semi-implicit Runge-Kutta method developed by Michelsen [1] is described. The method is used to solve various well-known systems of stiff ODEs and the performance of the algorithm is compared to in-built stiff and non-stiff solvers available in MATLAB (`ode15s`, `ode23s` and `ode45`). All calculations were done on a Windows PC with an Intel i7 processor (2.90 GHz).

2 The basic semi-implicit Runge-Kutta method

The algorithm described here can be used to solve problems of the form:

$$\frac{d}{dt}y = f(y) \tag{1}$$

with initial vector $y = y_0$.

Michelsen [1] modified a third order method proposed by Caillaud and Padmanabhan [2].

The algorithm is as follows:

$$
\begin{aligned}
k_1 &= h[I - haJ_k]^{-1} f(y_k) \\
k_2 &= h[I - haJ_k]^{-1} f(y_k + b_2 k_1) \\
k_3 &= [I - haJ_k]^{-1} [b_{31} k_1 + b_{32} k_2] \\
y_{k+1} &= y_k + R_1 k_1 + R_2 k_2 + R_3 k_3
\end{aligned}
\tag{2}
$$

where h = time step, I = identity matrix and J_k = Jacobian matrix of f with respect to y_k. The values of the parameters in Equations 2 are given by the following equations:

$$
\begin{aligned}
& a^3 - 3a^2 + \frac{3}{2}a - \frac{1}{6} = 0, a = 0.435867 \\
& b_2 = 0.75 \\
& b_{31} = \frac{-1}{6a}(8a^2 - 2a + 1) \\
& b_{32} = \frac{2}{9a}(6a^2 - 6a + 1) \\
& R_1 = \frac{11}{27} - b_{31} \\
& R_2 = \frac{16}{27} - b_{32}
\end{aligned}
$$

2.1 Step-size adjustement

Michelsen [1] also proposed a step-size adjustment algorithm based on the Richardson extrapolation method. The algorithm follows:

For an initial step length h_1, a one-step integration is carried out from t_k to t_{k+1} to give $y_{k+1,1}$. Then a two-step integration is carried out from t_k to t_{k+1} at $h_2 = h_1/2$ to give $y_{k+1,2}$. Let

$$
g = max_i \left| \left(\frac{y_{k+1,2} - y_{k+1,1}}{\epsilon} \right) \right| \tag{3}
$$

where ϵ is a prescribed vector of tolerance. If $g \leq 1$, the integrated result is accepted and the truncation error E is given by:

$$
E = \frac{8}{7}(y_{k+1,2} - y_{k+1,1}) \tag{4}
$$

The solution value $y_{k+1}*$ is computed by:

$$y_{k+1}* = y_{k+1,2} + \frac{E}{8} \tag{5}$$

If the value of g calculated by 3 is greater than 1, the result is not accepted and the integration from t_k to t_{k+1} is repeated with $h_3 = h_2/2$. Once a step has been accepted, the proposed step size for the next time step is selected as follows:

$$h_{k+1} = h_k \cdot \min\left[(4g)^{-0.25}, 3\right] \tag{6}$$

permitting an increase in h when $q < 0.25$ and possibly an acceleration by a factor of 3 when $q << 1$. The factor of 4 and the empirical restriction on maximum increase by a factor of 3 in Eq. 6 were proposed by Michelsen [1] as safety margins to avoid selection of too large steps that might be rejected.

2.2 Analysis

For each integration, the Jacobian J is evaluated only once and can be stored for further calculations. The main effort is thus that of solving a set of linear equations with three different right hand sides.

To investigate the numerical stability of this method, we apply it to the test problem

$$y' = \lambda y \tag{7}$$

Let the solution be of the form $y_{n+1} = \mu y_n$. Here the Jacobian is simply $J = \lambda$. We have:

$$y_{n+1} = y_n + R_1 k_1 + R_2 k_2 + R_3 k_3 \tag{8}$$

$$\begin{aligned} k_1 &= \frac{h\lambda}{1 - ah\lambda} y_n \\ k_2 &= \frac{h\lambda}{1 - ah\lambda}\left[y_n + b_2 \frac{h\lambda}{1 - ah\lambda} y_n\right] \\ &= \frac{h\lambda}{1 - ah\lambda} y_n + b_2 \left(\frac{h\lambda}{1 - ah\lambda}\right)^2 y_n \end{aligned}$$

$$k_3 = \frac{1}{1-ah\lambda}\left[b_{31}\frac{h\lambda}{1-ah\lambda}y_n + b_{32}\frac{h\lambda}{1-ah\lambda}y_n + b_{32}b_2\left(\frac{h\lambda}{1-ah\lambda}y_n\right)^2 y_n\right]$$
$$= \frac{h\lambda}{(1-ah\lambda)^2}(b_{31}+b_{32})y_n + b_{32}b_2\frac{(h\lambda)^2}{(1-ah\lambda)^3}y_n$$

Substituting values of k_1, k_2 and k_3 in Eq. 8 and rewriting the constants in terms of a gives

$$\frac{y_{n+1}}{y_n} = \mu = \frac{1+(1-3a)h\lambda + (3a^2-3a+\frac{1}{2})(h\lambda)^2}{(1-ah\lambda)^3} \tag{9}$$

$Re(\mu)$ is less than 1 for all $h\lambda$ with negative real part. Hence this method is strongly A stable and thus a suitable stiff solver.

3 MATLAB's inbuilt solvers

3.1 `ode15s` and `ode23s`

Both `ode15s` and `ode23s` integrate the system of differential equations of the form $y' = f(t, y)$ with initial conditions for y.

The `ode15s` code is a quasi-constant step size implementation of the Numerical Differentiation Formulas in terms of backward differences [3]. `ode23s` is an alternative to `ode15s` and is especially effective at crude tolerances. It is a simple structure fixed order method; the overhead is low except for linear algebra calculations which are relatively fast in MATLAB. Both these codes require calculation of the Jacobian matrix J. They include an option to input the analytical Jacobian if one is available, else it is calculated numerically.

3.2 `ode45`

`ode45` solves non-stiff differential equations of the form $y' = f(t, y)$. The code is an implementation of the explicit Runge-Kutta (4,5) pair of Dormand and Price [4]. Unlike the stiff solvers, it does not require calculation of the Jacobian.

In most cases, when looking to solve systems of ODEs, `ode45` is the code that is tried first. If there is reason to believe that the problem is stiff, or if the problem turns out to be too difficult for `ode45`, then `ode15s` is used.

4 Implementation

The described ODE solver was implemented in MATLAB as `ode_Mic` and used to solve various stiff ODEs descried in literature. The code also include options to include an analytical Jacobian function if one is available. If no function is provided, the Jacobian is calculated numerically using finite differences (code `jacobianest`). The tolerance vector was modified each time according to the specific problem being solved.

5 Test examples

5.1 Example 1

We use the method described above to solve a set of rate equations described by Seinfeld et al. [5]:

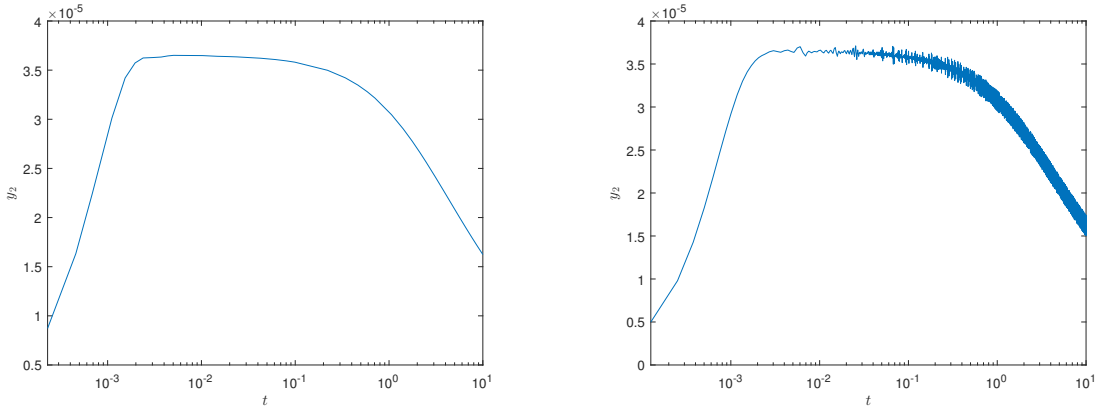
$$\begin{aligned} \frac{dy_1}{dt} &= -0.04y_1 + 10^4 y_2 y_3 \\ \frac{dy_2}{dt} &= 0.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2 \\ \frac{dy_3}{dt} &= 3 \cdot 10^7 y_2^2 \\ y_1(0) = 1, y_2(0) &= 0, y_3(0) = 0 \end{aligned} \tag{10}$$

The analytical jacobian matrix for this system is given by:

$$J = \begin{bmatrix} -0.04 & 10^4 y_3 & 10^4 y_2 \\ 0.04 & -10^4 y_3 - 6 \cdot 10^7 y_2 & -10^4 y_2 \\ 0 & 6 \cdot 10^7 y_2 & 0 \end{bmatrix}$$

The tolerance vector was taken as $\epsilon = [10^{-3}, 10^{-7}, 10^{-3}]$. The integration was carried out in the range [0,10] with an initial step size of 10^{-4}. The same problem was also solved using `ode15s`, `ode23s` and `ode45`.

Table 1 shows the statistics involved in computing the solution for the algorithm under consideration and inbuilt MATLAB solvers. All the three stiff solvers show markedly better performance across all parameters as compared to `ode45`. The performance of `ode_Mic` with an analytical Jacobian is comparable to MATLAB's inbuilt stiff solvers. Figure 1 shows the plot of y_2 vs time for stiff and non-stiff solutions. We see that the solution using `ode45` undergoes oscillations at higher time steps.



(a) Stiff solver (`ode_Mic`, `ode15s` or `ode23s`) (b) Non-stiff solver (`ode45`)

Figure 1: Comparision of solutions for non-stiff vs. stiff solvers (Example 1)

Table 1: Solver statistics for Michelsen's algorithm compared with inbuilt MATLAB solvers (Example 1)

	Running time (s)	Function evaluations	Steps
		`ode_Mic`	
Analytical Jacobian	0.00963	168	29
Numerical Jacobian	0.14337	8960	29
		`ode15s`	
Analytical Jacobian	0.03540	81	41
Numerical Jacobian	0.05238	89	41
		`ode23s`	
Analytical Jacobian	0.00851	60	19
Numerical Jacobian	0.01168	115	19
		`ode45`	
	1.20454	52201	29109

5.2 Example 2

The second test example is a model for a fluid bed reactor [6]:

$$\begin{aligned}
\frac{dy_1}{dt} &= 1.30(y_3 - y_1) + 1.04 \cdot 10^4 k y_2 \\
\frac{dy_2}{dt} &= 1.88 \cdot 10^3 (y_4 - y_2(1+k)) \\
\frac{dy_3}{dt} &= 1752 + 266.7 y_1 - 269.3 y_3 \\
\frac{dy_4}{dt} &= 0.1 + 320 y_2 - 321 y_4 \\
y_1(0) &= 759.167, y_2(0) = 0, y_3(0) = 600, y_4(0) = 0.1
\end{aligned} \tag{11}$$

where $k = 0.0006 \exp(20.7 - 15000/y_1)$.

The tolerance vector was taken as $\epsilon = [1, 1, 0.1, 0.1]$. The integration was carried out in the range [0,500] with an initial step size of 10^{-4}. Similar to Example 1, the running time of `ode_Mic` with a Numerical Jacobian was comparable to that of `ode15s` and `ode23s` (Table 2). The performance of `ode45` is several orders of magnitude worse than the stiff solvers.

Table 2: Solver statistics for Michelsen's algorithm compared with inbuilt MATLAB solvers (Example 2)

	Running time (s)	Function evaluations	Steps
		`ode_Mic`	
Analytical Jacobian	0.014103	252	43
Numerical Jacobian	0.341434	16112	39
		`ode15s`	
Analytical Jacobian	0.190613	2354	764
Numerical Jacobian	0.02084	126	85
		`ode23s`	
Analytical Jacobian	0.018893	341	114
Numerical Jacobian	0.014007	331	48
		`ode45`	
	55.881972	2123530	1327201

5.3 Example 3 - Oregonator reaction

The oscillatory reaction between $HBrO_2$, Br^- and Ce(IV) is known as the Oregonator reaction [7]. The rate expressions are given by:

$$\begin{aligned}
\frac{dy_1}{dt} &= 77.27(y_2 + y_1(1 - 8.375 \times 10^{-6} y_1 - y_2)) \\
\frac{dy_2}{dt} &= \frac{1}{77.27}(y_3 - (1 + y_1)y_2) \\
\frac{dy_3}{dt} &= 0.161(y_1 - y_3) \\
y_1(0) = 1, y_2(0) &= 2, y_3(0) = 3
\end{aligned} \tag{12}$$

In this system, we observe that at smaller integration limits (e.g. [0,10]) the problem is

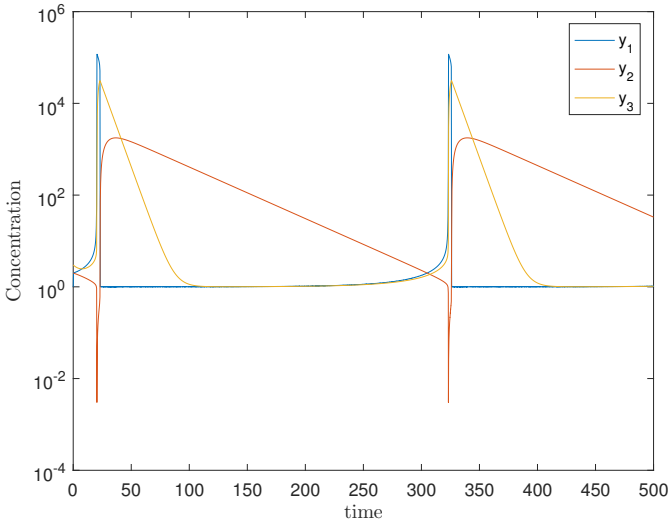


Figure 2: Oregonator reaction plot

non-stiff and both stiff and non-stiff solvers integrate the system effectively. However, at larger intervals (once oscillations start taking place), the stiff solvers show markedly better performance than `ode45` (Table 3).

Table 3: Solver statistics for Michelsen's algorithm compared with inbuilt MATLAB solvers (Example 3 - Oregonator reaction)

	Running time	Function evaluations	Steps
		`ode_Mic`	
Analytical Jacobian	0.15919	5538	923
Numerical Jacobian	4.46163	295203	923
		`ode15s`	
Analytical Jacobian	0.11093	1566	583
Numerical Jacobian	0.11796	1845	583
		`ode23s`	
Analytical Jacobian	0.06939	2142	709
Numerical Jacobian	0.14239	4266	709
		`ode45`	
	584.78864	28648200	17905633

5.4 Example 4 - Photochemical smog model

A kinetic model Photochemical smog formation presented by Gelinas [8] is solved. The original model given in ref. [8] involves 29 species and over 60 equations with rate constants varying by 20 orders of magnitude. A simplified form of the model as described in ref. [9] with 12 species and 22 reactions is used and shown in Tables 4 & 5.

The system was solved using `ode_Mic`, `ode15s`, `ode23s` and `ode45` for increasing integration limits. As expected (especially for this large system), `ode45` is orders of magnitude worse than any of the stiff solvers for all integration limits. Interestingly, `ode45` varies as $\mathcal{O}(n)$ whereas `ode_Mic` varies as $\mathcal{O}(n^{0.5})$ (where n is max integration limit).

Table 4: Components in the smog model

1. NO_2	2. NO	3. O
4. O_3	5. C_4H_8	6. $C_3H_7O_2$
7. HO_2	8. CH_3CO_3	9. CH_3O_2
10. HO	11. $C_4H_8OHO_2$	12. CH_2OHO_2

Table 5: Photochemical smog model (Example 3) - Reactions and rate constants.

1			$\longrightarrow$	2	+	3	6.7×10^{-3}
3			$\longrightarrow$	4			6.73×10^{4}
2	+	4	$\longrightarrow$	1			9.1×10^{-2}
3	+	5	$\longrightarrow$	6	+	7	3.8×10^{2}
6	+	2	$\longrightarrow$	1	+	7	3.0
4	+	5	$\longrightarrow$	7	+	8	4.9×10^{-5}
4	+	5	$\longrightarrow$	4			2.1×10^{-5}
2	+	8	$\longrightarrow$	1	+	9	3.0
1	+	8	$\longrightarrow$	inactive			0.1
5	+	8	$\longrightarrow$	9			1.0×10^{-4}
2	+	9	$\longrightarrow$	1	+	7	3.0
5	+	10	$\longrightarrow$	11			1.0×10^{2}
2	+	11	$\longrightarrow$	1	+	12	3.0
2	+	12	$\longrightarrow$	1	+	10	3.0
2	+	7	$\longrightarrow$	1	+	10	50
7	+	7	$\longrightarrow$	inactive			50
7	+	10	$\longrightarrow$	inactive			50
7	+	6	$\longrightarrow$	inactive			50
7	+	9	$\longrightarrow$	inactive			50
7	+	8	$\longrightarrow$	inactive			50
7	+	11	$\longrightarrow$	inactive			50
7	+	12	$\longrightarrow$	inactive			50

Table 6: Solver statistics for Michelsen's algorithm compared with inbuilt MATLAB solvers (Example 4 - smog model)

Time step (s)	Running time (s)	Function evaluations	Steps
	`ode_Mic`		
1	1.010029	15072	13
10	3.025674	51496	42
100	8.835829	158262	126
1000	25.085625	456571	362
	`ode15s`		
1	0.021004	61	26
10	0.060441	146	48
100	0.085993	233	75
1000	0.034693	284	94
	`ode23s`		
1	0.083637	252	17
10	0.057533	569	38
100	0.081612	914	61
1000	0.104977	1259	84
	`ode45`		
1	6.104102	137725	86245
10	67.269395	1379530	862373
100	653.744277	13792100	8620221

6 Conclusion

The integration algorithm proposed by Michelsen was implemented and found to be well suited for a variety of stiff problems arising in chemical engineering. Using the analytical Jacobian improves performance in each of the stiff solvers used. In all these cases, an arbitrary initial step size was chosen. A more systematic method may improve performance by reducing the number of step length adjustments required for the first step.

References

[1] Michael L. Michelsen. An efficient general purpose method for the intergration of stiff ordinary differential equations. *AIChE Journal*, 22(3):594–597, 1976.

[2] J.B. Caillaud and L. Padmanabhan. An improved semi-implicit runge-kutta method for stiff systems. *The Chemical Engineering Journal*, 2(4):227 – 232, 1971.

[3] Lawrence F. Shampine and Mark W. Reichelt. The matlab ode suite. *SIAM Journal on Scientific Computing*, 18(1):1–22, 1997.

[4] J.R. Dormand and P.J. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19 – 26, 1980.

[5] J. H. Seinfeld, Leon Lapidus, and Myungkyu Hwang. Review of numerical integration techniques for stiff ordinary differential equations. *Industrial & Engineering Chemistry Fundamentals*, 9(2):266–275, 1970.

[6] Richard C. Aiken and Leon Lapidus. An effective numerical integration method for typical stiff systems. *AIChE Journal*, 20(2):368–375, 1974.

[7] Richard J. Field and Richard M. Noyes. Oscillations in chemical systems. iv. limit cycle behavior in a model of a real chemical reaction. *The Journal of Chemical Physics*, 60(5):1877–1884, 1974.

[8] Robert J Gelinas. Stiff systems of kinetic equationsa practitioner's view. *Journal of Computational Physics*, 9(2):222 – 236, 1972.

[9] Michael L. Michelsen. Application of semi-implicit rungekutta methods for integration of ordinary and partial differential equations. *The Chemical Engineering Journal*, 14(2):107 – 112, 1977.