Numerical methods for complex ODE / DAE Systems

Introduction

We will further elaborate on numerical methods used for simulation of engineering systems described by ordinary differential equations, either explicit (ODE) or implicit (DAE).

It is crucial to consider that numerical integration is an *approximation* of the real integration process. Besides errors due to the time discretization, errors are generated because numerical integration formulas are approximations of the real integration function.

To compute the variables in the model as a time function on a digital computer, time will be discretized. Otherwise, we have an infinite amount of values of time between $t0$ and te, since t is a real variable. Only at discrete points tk the model is computed. The result is that the model variables are available as a discrete series of values, and not as a continuous function.

The time distance between two adjacent points is called the *time step*, hk, or
integration interval. This hk need not necessarily be constant. *integration interval*

The model suitable for simulation consists of a set of *Ordinary Differential Equations*, and is called a *simulation model*. These differential equations describe the system as a set of *state equations*, where the initial values of the state variables (also called *initial conditions*) are known. In numerical mathematics, this is called an *Initial Value Problem* (IVP)*:*

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t)$$
$$\mathbf{y} = \mathbf{g}(\mathbf{x}, \mathbf{u}, t)$$

where x is the array of *state variables*, also called the state vector; x(0), the initial values, which are known. u is the array of input variables. f are the state equations. y is the array of *output variables*, computed from the states and inputs using the functions g.

Numerical integration methods for ODE

A numerical integration method specifies how $\mathbf{x}_{k+1}$ is approximated in terms of $\mathbf{x}$ itself and its derivatives, both at previous, current or future moments of time. Practical numerical methods consist of a linear combination of these terms $\mathbf{x}^{(i)}_{k+1-j}$ using parameters α_{ij}, n and r ($\dot{x} = x^{(1)}$):

$$\sum_{i=0}^{n} \sum_{j=0}^{r} \alpha_{ij} h^{i}_{k+1-j} \mathbf{x}^{(i)}_{k+1-j} = 0 \qquad (3)$$

The method is explicit if all $\alpha_{i0}=0$ for i=1...n, otherwise the method is implicit.

Equation (3) is an approximation, so errors are made:

· *Truncation errors*, due to the non-idealness of the approximation formula.
· *Round-off errors*, due to the limited machine precision.
Round-off errors

At smaller timesteps, the truncation error usually becomes smaller, whereas the round-off errors become larger.

For a useful approximation, the time step hk depends on both the model and the integration method.

The form of the numerical integration method, either explicit or implicit, the number of steps and the order of derivatives involved, significantly influence numerical stability and effective applicability of the method.

Explicit numerical integration methods have less computational load than implicit methods, but unfortunately they also have a small stability region, and thus relatively small time steps hk are necessary. The disadvantage of implicit numerical integration methods is that iterative calculations involving matrix inversion

Most numerical integration methods have been developed and evaluated for linear equations or for nonlinear equations only in the vicinity of the point of linearization

Basically there are 3 general classes of integration methods, categorized by the number of steps and order of derivatives:

a) Multistep, high-order derivative methods
the full version of equation (3).

b) Multistep, first-order derivatives methods
n=1, only **x** and its derivative are used. Higher order methods take more values of the past, until **x**k+1-n.

c) One-step higher-order derivatives methods
r=1, only values of **x**k and **x**(i)
k, the ith derivative of **x**k, are used. These
methods are Runge-Kutta methods. No values of previous moments are used.

Multistep, first-order derivative methods

For multistep, first-order derivative methods, the parameter n is equal to one Equation (3) can be written as:

$$\mathbf{x}_{k+1} = a_{10} h_{k+1} \dot{\mathbf{x}}_{k+1} + \sum_{j=1}^{r} a_{0j} \mathbf{x}_{k+1-j} + \sum_{j=1}^{r} a_{1j} h_{k+1-j} \dot{\mathbf{x}}_{k+1-j} \qquad (4)$$

If $a_{10} = 0$, the method is explicit, otherwise the method is implicit, as $\dot{\mathbf{x}}_{k+1}$ is the only term at $\mathbf{x}_{k+1}$ at the right-hand side. The first summation represents $\mathbf{x}$ at previous moments of time; the second summation represents the derivative of $\mathbf{x}$ at previous moments of time. If $r = 1$, then it is a *one*-step method: only $\mathbf{x}(t)$ and its derivative are used. At $r > 1$, it is a multistep method, which is not self-starting: at the beginning, old values of $\mathbf{x}_{k-1}$ etc. are

AB-1	Adams Bashforth = explicit Euler	$\mathbf{x}_{k+1} = \mathbf{x}_k + h_k \dot{\mathbf{x}}_k = \mathbf{x}_k + h_k \mathbf{f}_k$
AB-2	Adams Bashforth = trapezium rule	$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{3}{2} h_k \dot{\mathbf{x}}_k - \frac{1}{2} h_k \dot{\mathbf{x}}_{k-1}$
AM-1	Adams Moulton = implicit Euler	$\mathbf{x}_{k+1} = \mathbf{x}_k + h_{k+1} \dot{\mathbf{x}}_{k+1}$
AM-2	Adams-Moulton = trapezium rule	$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{1}{2} h_{k+1} \dot{\mathbf{x}}_{k+1} + \frac{1}{2} h_k \dot{\mathbf{x}}_k$

The Adams-Bashforth methods are explicit method, require one evaluation of the model per time step, and have a limited stability region (conditionally stable). The Adams-Moulton methods are implicit in which $\dot{\mathbf{x}}_{k+1}$ is substituted as a fraction of $\mathbf{x}_{k+1}$, and require one evaluation of the model. The second order method is A-stable (absolute stable, the stability region is the whole left half plane), and can therefore be used for stiff models. The local truncation error of these methods is of order $O(h^{r+1})$.

At a *predictor-corrector* method, first a *predictor* step is done (= a forecast) of $\mathbf{x}_{k+1}$ and thus of $\mathbf{f}(\mathbf{x}_{k+1})$. Then, this result is used to compute a *corrector* step (= a better approximation of $\mathbf{x}_{k+1}$). As predictor, an explicit method is used and as corrector an implicit one.

At *variable-step* methods, the step size h_{k+1} is adjusted every time step to the dynamics of the system, such that $h_k \lambda_{i,k}$ stays in the stability region of the numerical integration method used.

One-step higher-order derivative methods

Writing **x**k+1 as a Taylor's series, the coefficients of the numerical method, aij, can be determined. This way is useful for theoretical research, but for a practical algorithm, the aij's are approximated via extrapolation.

extrapolation. $\mathbf{x}$ is calculated at a number of points $t_k + h_i h_k$ where $0 \le h_i \le 1$ The values $\mathbf{x}(t_k + h_i h_k)$ and $\mathbf{x}(t_k)$ are used to match terms in the Taylor's series, which result in a specification of a numerical integration method

These methods are called *r-stage Runge-Kutta* methods. They need r model computations per time step. The general r-stage Runge-Kutta method is given by:

$$
\begin{aligned}
x_Q &= x_k + \sum_{j=1} b_{qj} \mathbf{z}_j \quad q = 1,2,...r \\
t_Q &= t_k + h_k \sum_{j=1}^{q} b_{qj} \\
\mathbf{z}_q &= h_k \mathbf{f}(\mathbf{x}_Q, \mathbf{t}_Q) \\
\mathbf{x}_{k+1} &= \mathbf{x}_k + \sum_{q=1}^{r} g_q \mathbf{z}_q
\end{aligned}
\tag{6}
$$

Equations for the parameters γ_q and β_{qj} are derived by matching terms with the Taylor's series. If $\beta_{qj} = 0$ for $j \ge q$, the method is explicit, otherwise implicit. As indicated above, there are more parameters to determine than equations available, so that a family of methods results. For explicit methods, the order is equal to the stage r for $r \le 4$. Otherwise, the order is less than the stage. The implicit methods can have a maximal order of 2r

Name	Coefficients	Algorithm
RK1, explicit Euler	$g_1 = 1$	$\mathbf{q}_1 = h_k \mathbf{f}(\mathbf{x}_k, t_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{q}_1$
RK2, Heun	$g_1 = 1/2$ $g_2 = 1/2 \quad b_{21} = 1/2$	$\mathbf{q}_1 = h_k \mathbf{f}(\mathbf{x}_k, t_k)$ $\mathbf{q}_2 = h_k \mathbf{f}(\mathbf{x}_k + 1/2\,\mathbf{q}_1, t_k + 1/2\,h_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + 1/2\,\mathbf{q}_1 + 1/2\,\mathbf{q}_2$
RK4, Classical	$g_1 = 1/6$ $g_2 = 1/3 \quad b_{21} = 1/2$ $g_3 = 1/3 \quad b_{32} = 1/2$ $g_4 = 1/2 \quad b_{43} = 1$	$\mathbf{q}_1 = h_k \mathbf{f}(\mathbf{x}_k, t_k)$ $\mathbf{q}_2 = h_k \mathbf{f}(\mathbf{x}_k + 1/2\,\mathbf{q}_1, t_k + 1/2\,h_k)$ $\mathbf{q}_3 = h_k \mathbf{f}(\mathbf{x}_k + 1/2\,\mathbf{q}_2, t_k + 1/2\,h_k)$ $\mathbf{q}_4 = h_k \mathbf{f}(\mathbf{x}_k + \mathbf{q}_3, t_k + h_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + 1/6\,\mathbf{q}_1 + 1/3\,\mathbf{q}_2 + 1/3\,\mathbf{q}_3 + 1/6\,\mathbf{q}_4$
RK1, implicit Euler	$g_1 = 1 \quad b_{11} = 1$	$\mathbf{q}_1 = h_k \mathbf{f}(\mathbf{x}_k + \mathbf{q}_1, t_k + h_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{q}_1$
RK2, trapezodial	$g_1 = 1/2 \quad b_{11} = 0$ $g_2 = 1/2 \quad b_{21} = 1/2$ $b_{22} = 1/2$	$\mathbf{q}_1 = h_k \mathbf{f}(\mathbf{x}_k, t_k)$ $\mathbf{q}_2 = h_k \mathbf{f}(\mathbf{x}_k + 1/2\,\mathbf{q}_1 + 1/2\,\mathbf{q}_2, t_k + h_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + 1/2\,\mathbf{q}_1 + 1/2\,\mathbf{q}_2$

Table 2 Runge-Kutta methods

Variable time step Runge-Kutta's do exist. Their time step is controlled by calculating each time step twice by either using two Runge-Kutta . For the first method, computations are minimized by choosing the coefficients such, that some intermediate results are common to both algorithms (Gear, 1971) . Two algorithms which are base on the multiple use of intermediate results, are often used: the Runge-Kutta-Merson algorithm and the Runge-Kutta- Fehlberg algorithm.

General remarks on integration formulas

Some remarks on the order of derivatives, the number of steps, and the coefficients of the numerical integration methods:

1. Order of derivatives

The higher the order of derivatives used in the formula, the more accurate and more flexible the formula is. But at each time step, an accompanying heavy computation load exists, i.e. the computation of the high order derivatives of **f**(**x**,t) must be done.

2. Number of steps
The larger the number of steps, the more accurate and more flexible the formula is. But estimates of the unknown initial values of **x**k are needed.

3. Implicit formula versus explicit formula
Implicit formulas always require the computation of an inverse of a matrix or its equivalent. Explicit formulas are easy to apply to complicated nonlinear differential equations, and do not involve the calculation of a matrix inverse, but have a smaller stability region.

Stiffness

Stiffness is a subtle, difficult, and important concept in the numerical solution of ordinary diferential equations. It depends on the diferential equation, the initial conditions, and the numerical method.

"A problem is stiff if the solution being sought is varying slowly, but there are nearby solutions that vary rapidly, so the numerical method must take small steps to obtain satisfactory results"

This systems are given by a differential equation in the general form

$$\mathbf{M(t)\ dy/dt = f\ (t,y)} \qquad \mathbf{y(to)=yo}$$

with a mass matrix M(t) that is non-singular and usually sparse

A model of flame propagation provides an example. If you light a match, the ball of flame grows rapidly until it reaches a critical size. Then it remains at that size because the amount of oxygen being consumed by the combustion in the interior of the ball balances the amount available through the surface.

The simple model is

$$\begin{aligned} \dot{y} &= y^2 - y^3 \\ y(0) &= \delta \\ 0 \le t &\le 2/\delta \end{aligned}$$

The scalar variable $y(t)$ represents the radius of the ball. The y^2 and y^3 terms come from the surface area and the volume. The critical parameter is the initial radius, δ, which is "small." We seek the solution over a length of time that is inversely proportional to δ.

```
delta = 0.0001;
ode45(F,[0 2/delta],delta,opts);
```

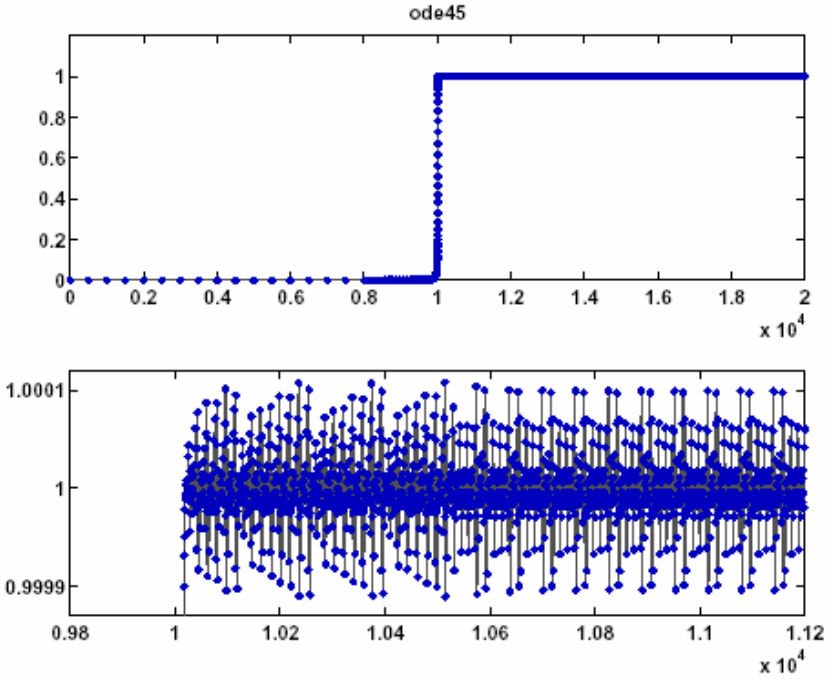


Figure 7.4. *Stiff behavior of* `ode45`

What can be done about stiff problems? You don't want to change the dif-differential equation or the initial conditions, so you have to change the numerical method. Methods intended to solve stiff problems efficiently do more work per step, but can take much bigger steps. Stiff methods are *implicit*. At each step they use Matlab matrix operations to solve a system of simultaneous linear equations that helps predict the evolution of the solution.

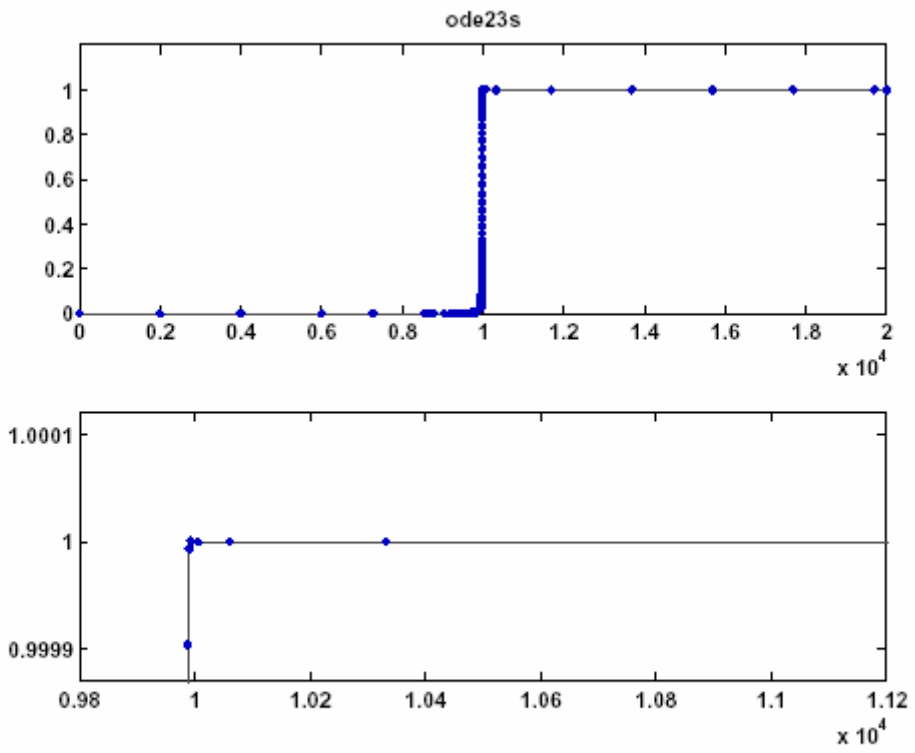


Figure 7.5. *Stiff behavior of* `ode23s`

Figure 7.5 shows the computed solution and the zoom detail. You can see that ode23s takes many fewer steps than ode45. This is actually an easy problem for a stiff solver. In fact, ode23s takes only 99 steps and uses just 412 function evaluations, while ode45 takes 3040 steps and uses 20179 function evaluations. Stiffness even affects graphical output. The print files for the ode45 figures are much larger than those for the ode23s figures.

Systems of Equations

Many mathematical models involve more than one unknown function, and second and higher order derivatives. These models can be handled by making $y(t)$ a vector-valued function of t. Each component is either one of the unknown functions or one of its derivatives. The MATLAB vector notation is particularly convenient here.

A slightly more complicated example, the *two-body problem*, describes the orbit of one body under the gravitational attraction of a much heavier body. Using Cartesian coordinates, $u(t)$ and $v(t)$, centered in the heavy body, the equations are

$$\ddot{u}(t) = -u(t)/r(t)^3$$
$$\ddot{v}(t) = -v(t)/r(t)^3$$

where

$$r(t) = \sqrt{u(t)^2 + v(t)^2}$$

The vector $y(t)$ has four components,

$$y(t) = \begin{bmatrix} u(t) \\ v(t) \\ \dot{u}(t) \\ \dot{v}(t) \end{bmatrix}$$

The differential equation is

$$\dot{y}(t) = \begin{bmatrix} \dot{u}(t) \\ \dot{v}(t) \\ -u(t)/r(t)^3 \\ -v(t)/r(t)^3 \end{bmatrix}$$

Numerical integration methods for DAE

Differential algebraic equations (DAEs) are systems of differential equations

$$F(x', x, t) = 0 \tag{1}$$

with $\partial F/\partial x'$ identically singular. The name arises since often (1) is a mix of differential and algebraic equations. DAEs arise naturally in many areas [1], [15]. The algebraic equations can arise because of physical constraints, desired behavior, or when discretizing spatial operators during the method of lines solution of a PDE. Depending on the area of application, DAEs are also called implicit, descriptor or singular.

2.1 Analytical

Consider the following simple example of (1):

$$x_2' = x_1 \tag{2a}$$
$$x_2 = t + \alpha(t)x_3 \tag{2b}$$
$$x_3' = x_3 + 1 \tag{2c}$$

Here $\alpha(t)$ is a nonzero coefficient. The solution of (2) is

$$x = \begin{bmatrix} 1 + \alpha'(t)(-1 + ce^t) + c\alpha(t)e^t \\ t + \alpha(t)(-1 + ce^t) \\ -1 + ce^t \end{bmatrix}, \quad c \text{ an arbitrary constant} \tag{3}$$

From this example we can make several observations about how DAEs differ from ordinary differential equations.

1. The solution x of (1) can depend on derivatives of the defining equations F. Note the $\alpha'(t)$ term that appears in the solution (3).
2. Only some initial conditions will admit smooth solutions. These are called *consistent initial conditions.*
3. There can be hidden constraints. The solutions of (2) satisfy not only the constraint (2b) but also the constraint $\alpha'(t)x_3 + \alpha(t)(1 + x_3) - x_1 = 0$.
4. The best that can be hoped for is that the solutions form a smooth manifold, called the *solution manifold.* It is parameterized by t, c in the example above.

Suppose the DAE (1) is a system of n equations in the n dimensional state variable x and that F is sufficiently differentiable in the variables (x', x, t) so that all needed differentiations can be carried out. Suppose also that the solutions form a smooth manifold and that a (consistent) initial value on the manifold uniquely determines a solution. Such a DAE is sometimes called *solvable.* As noted, the solution x of (1) will depend, in general, on derivatives of F. If the ith equation of (1) is differentiated r_i times with respect to t for $i = 1, \ldots, n$, we get the $m = n + \sum_{i=1}^{n} r_i$ *derivative array equations*

$$(4) \qquad G(x', w, x, t) = \begin{bmatrix} F(x', x, t) \\ \frac{d}{dt}F(x', x, t) \\ \vdots \\ \frac{d^r}{dt^r}F(x', x, t) \end{bmatrix} = \begin{bmatrix} F(x', x, t) \\ F_{x'}x'' + F_x x' + F_t \\ \vdots \\ F_{x'}x^{(r+1)} + \cdots \end{bmatrix} = 0$$

where $w = [x^{(2)}, \ldots, x^{(r+1)}]$

There are several versions of the *index* of a DAE and they are not equivalent for general nonlinear DAEs [6]. For our purposes, we shall define the index ν to be the least value of r for which the derivative array equations (4) uniquely determine x' given a consistent (x, t). This index is sometimes called the differentiation index. It should be noted that in a physical model the index is not determined just by the equations but also by which variables are considered known, such as inputs, and which are considered unknown.

In order to integrate a DAE, or for that matter an ODE, we need an estimate of x'. The index measures, in some sense, how hard it is to get x'. The index also measures the loss of smoothness in going from the coefficients and forcing functions to the solutions. Finally the index also affects the conditioning of the matrices which occur in some methods such as BDF (Backward differentiation formulas). Several numerical methods have been proposed for solving DAEs including BDF or implicit Runge-Kutta (IRK) methods [1],

For index one DAEs we have codes such as DASSL. There is no general code available for even index two problems.

To explain why it is difficult to solve high-index problems, let us briefly review the methods to solve ODEs. When solving an ODE on state space form the main task is to integrate. The derivatives are approximated by finite differences. A simple approximation is forward Euler, $\dot{x}_n \approx (x_{n+1} - x_n)/h$, where h is the step size. Applied to $\dot{x} = f(t, x)$, we obtain the explicit recursion $x_{n+1} = x_n + hf(t_n, x_n)$. We can also use backward Euler, $\dot{x}_{n+1} \approx (x_{n+1} - x_n)/h$ to obtain $x_{n+1} = x_n + hf(t_{n+1}, x_{n+1})$, which is an implicit recursion.

For example, backward Euler discretization transforms the differential algebraic problem, $F(t, x, \dot{x}) = 0$ into the recursion $F(t_n, x_n, (x_n - x_{n-1})/h) = 0$, which is readily solved when the index is 0 or 1. Backward differences of order upto five are used in the differential-algebraic solver DASSL developed by Petzold (see Brenan et al, 1989).

Backward differences for order two is given by the recursive equation $F(t_n, x_n, (3x_n - 4x_{n-1} + x_{n-2})/h) = 0$**. In general BDF formulas are applied as follows**

$$F(t_n, x_n, \sum_{i=0}^{k} \alpha_i x_{n-i} \Big/ h) = 0$$

The index can be reduced by differentiating the equations a number of times, see e.g. Brenan et al. (1989, p. 33). The new problem obtained in this way is called the underlying ODE, or UODE. However, it is often less satisfactory to solve the UODE, because its set of solutions is larger.

To avoid such difficulties, one may try to obtain a low-index formulation, with a solution set identical to that of the original problem. This can be achieved by augmenting the system as the index reduction proceeds: all original equations, and their successive derivatives are retained in the process. The result is an overdetermined but consistent index 1 DAE.

For each *differentiated equation* appended to the original system, we introduced one "new" dependent variable. The introduced variables represent derivatives and we call them *dummy derivatives*. Dummy derivatives are purely algebraic variables and are not subject to discretization.

To find the differentiated index 1 problem we can use Pantelides's algorithm, cf. Pantelides (1988), intended for finding consistent initial values for a DAE. Pantelides's algorithm establishes the minimum number ν_i each equation has to be differentiated to make the differentiated problem $\mathcal{G}x = \mathcal{F}'x = 0$ structurally non-singular with respect to its

The MATLAB ODE and DAE Solvers

This section is derived from the Algorithms portion of the MATLAB Reference Manual page for the ODE solvers.

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a one-step solver. In computing $y(t_{n+1})$, it needs only the solution at the immediately preceding time point, $y(t_n)$. In general, `ode45` is the first function to try for most problems.

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It is often more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver.

`ode113` is a variable-order Adams-Bashforth-Moulton PECE solver. It is often more efficient than `ode45` at stringent tolerances and if the ODE file function is particularly expensive to evaluate. `ode113` is a multistep solver — it normally needs the solutions at several preceding time points to compute the current solution.

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable-order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` if `ode45` fails or is very inefficient and you suspect that the problem is stiff, or if solving a differential-algebraic problem.

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it is often more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective.

- `ode45`. Nonstiff problems, medium accuracy. Use most of the time. This should be the first solver you try.
- `ode23`. Nonstiff problems, low accuracy. Use for large error tolerances or moderately stiff problems.
- `ode113`. Nonstiff problems, low to high accuracy. Use for stringent error tolerances or computationally intensive ODE functions.
- `ode15s`. Stiff problems, low to medium accuracy. Use if `ode45` is slow (stiff systems) or there is a mass matrix.
- `ode23s`. Stiff problems, low accuracy, Use for large error tolerances with stiff systems or with a constant mass matrix.

TABLE OF MATLAB's NUMERICAL METHODS

The simplest system of differential algebraic equations (DAEs) has the folowing semiexplicit form:

$$u' = f(t,u,v) \qquad (1a)$$
$$0 = g(t,u,v) \qquad (1b)$$

In this notation, t is the independent variable (time); u stands for the differential variables, and v stands for the algebraic variables.

One idea for solving (1) could be to solve (1a) as an ODE. Evaluating the derivative u' for a given u would require solving the algebraic equation (1b) for the corresponding value of v. This is easy enough in principle, but can lead to initial value problems for which the evaluation of f is rather expensive.

MATLAB solvers use a different approach. To solve DAEs in MATLAB, first you need to combine the differential and algebraic part. Note that any semiexplicit system (1) can be written as

$$M * y' = F(t,y)$$

where M = [I 0; 0 0] and y = [u;v], F=[f;g].

For differential algebraic equations, the mass matrix M is singular, but such systems can still be solved with ODE15S and ODE23T. In fact, the solvers can handle more general systems, with time- and state-dependent singular mass matrices

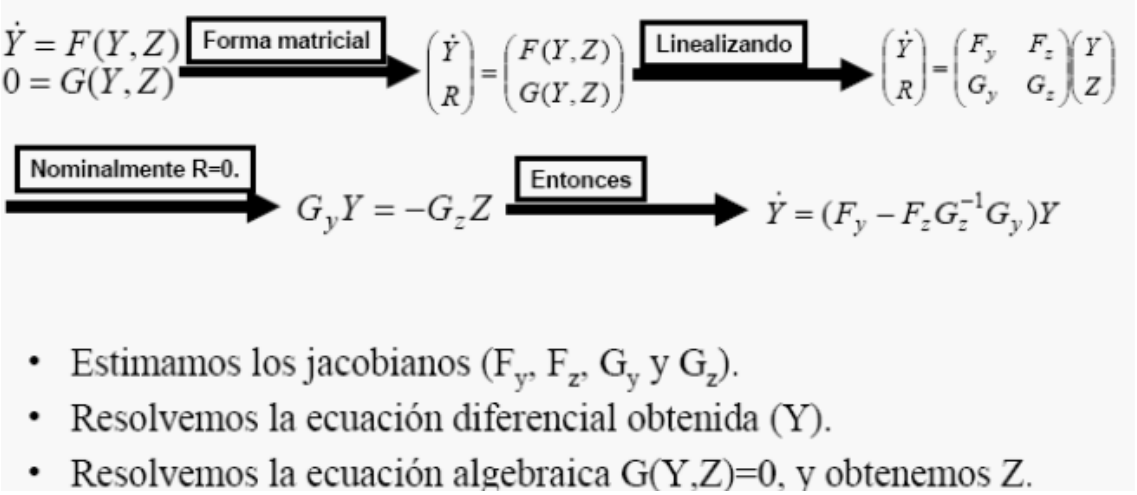
$$M(t,y) * y' = F(t,y)$$

The only restriction on DAEs solved in MATLAB is that they must be of index 1. Semi explicit DAEs are of index 1 when their matrix of partial derivatives dg/dv is non-singular

The MODELICA ODE and DAE Solvers

Hay dos formas de resolver las ecuaciones DAE en Modelica

1. Por linealización de la DAE:



Es necesario que G_z sea no singular

2. Por diferenciación de la DAE:

- Si el indice diferencial es 1, se realizará la integración por aplicación de métodos de integración implicitos, tales como Runge-Kutta Implicito (RADAU5) o por aplicación de formulas BFD de diferente orden (DASSL) sobre la nueva ecuación

  $$M(t,x_d) * x_d' = G(x_d,t)$$

  siendo x_d el nuevo vector de derivadas.

- Si el indice diferencial es superior a 1, no es posible obtener un sistema de ecuaciones en x_d de primer orden, por lo que no es posible aplicar los métodos de integración implicitos.

  Por ello se realizará la reducción de indice diferencial hasta obtener una sistema de indice diferencial 1 añadiendo variables mudas (dummy), resultando un sistema de mayor dimension, siendo resuelto como el caso anterior.

  Para la generación del sistema de ecuaciones en x_d así como la obtención de variables mudas se utilizará el algoritmo de Pantelides.

Modelica permite la integración tanto de ODE's como de DAE's a través de varios métodos de integración (ver tabla).

Method	Model Type	Order	Stiff	Root Find	Algorithm
DEABM	ODE	1-12	No	No	Adams/Bashforth/Moulton; reliable, maybe slow
LSODE1	ODE	1-12	No	No	Adams/Bashforth/Moulton; faster than DEABM
LSODE2	ODE	1-5	Yes	No	Backward Difference Formulae; Gear method
LSODAR	ODE	1-12, 1-5	Both	Yes	Adams/Bashforth/Moulton
DOPRI5	ODE	5	No	No	Runge-Kutta by Dormand and Prince
DOPRI8	ODE	8	No	No	Runge-Kutta by Dormand and Prince
GRK4T	ODE	4	Yes	No	Linearly-implicit Rosenbrock
DASSL	DAE	1-5	Yes	Yes	BDF; default choice in Dymola
ODASSL	ODAE	1-5	Yes	Yes	Modified DASSL for overdetermined DAEs
MEXX	ODAE	2-24	No	No	Special index-2 DAE; not for ODE!