4 Multistep Methods And Numerical Stability

Read *sections 9.3.2 – 9.3.4, 9.3.8, 9.4.*
Review questions *9.14 – 9.16, 9.19, 9.22, 9.27, 9.29, 9.30, 9.33, 9.34, 9.37 9.38, 9.42.*

4.1 Adams Methods

Let us return to our initial value problem

$$y' = f(t,y), \quad y(t_0) = y_0.$$

All methods which we introduced earlier use only information at t_k (that is, y_k, h, and f) in order to compute y_{k+1}. One could expect that the inclusion of information from several previous steps $t_k, t_{k-1}, \ldots$ could make the approximation of $y(t_{k+1})$ even more efficient. Such methods are called *multistep methods*.

A large class of methods can be obtained by applying polynomial interpolation. The differential equation near t_k is rewritten as an equivalent integral equation,

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} y'(t)dt = \int_{t_k}^{t_{k+1}} f(t,y(t))dt.$$

We replace the integrand $f(t,y(t))$ by a polynomial $p(t)$:

$$y(t_{k+1}) - y(t_k) \approx \int_{t_k}^{t_{k+1}} p(t)dt.$$

It remains to choose the interpolation nodes. We assume that $y_k, y_{k-1}, \ldots, y_{k+1-m}$ are already computed (and approximate $y(t_{k+1-i})$ well). Let the grid be equidistant, $t_{k+1-i} - t_{k-i} \equiv h$. Then, we may choose the interpolation data

$$(t_i, f_i), \quad i = k, \ldots, k+1-m,$$

such that

$$f_i = f(t_i, y_i) \quad \big(\approx f(t_i, y(t_i))\big).$$

Since we have m interpolation points, p is a polynomial of degree $m-1$. This polynomial can be represented in such a way that the integral $\int_{t_k}^{t_{k+1}} p(t)dt$ is easily computable. One uses either Lagrange or Newton interpolation. This leads to different representations of the same numerical method. While in practical realizations of the resulting multistep methods the representation by Newton interpolation is preferable, we will present the result of the Lagrange ansatz:

$$\begin{aligned} y_{k+1} - y_k &= h(\beta_1 f_k + \beta_2 f_{k-1} + \cdots + \beta_m f_{k+1-m}) \\ &= h \sum_{i=1}^{m} \beta_i f_{k+1-i}. \end{aligned}$$

Note that the coefficients depend on m even if this is not denoted explicitly. The methods constructed are known as *Adams-Bashforth methods*. They were first published in 1883 although they were developed by J.C. Adams in 1855. The coefficients β_i are precomputed and can be found in special tables. Table 1 below contains some coefficients. Note that, for $m = 1$, the resulting Adams-Bashforth method is exactly Euler's method. So Euler's method can be considered as the simplest multistep method as well as the simplest Runge-Kutta method.

m	β_1	β_2	β_3	β_4
1	1			
2	$\frac{3}{2}$	$-\frac{1}{2}$		
3	$\frac{23}{12}$	$-\frac{4}{3}$	$\frac{5}{12}$	
4	$\frac{55}{24}$	$-\frac{59}{24}$	$\frac{37}{24}$	$-\frac{3}{8}$

Table 1: Some coefficients for Adams-Bashforth methods

When applying multistep methods one needs approximations $y_{-1}, y_{-2}, \ldots$ at $k = 0$ which are neither available nor even defined. In order to start the computation, two alternatives are available:

- One computes $y_1, \ldots, y_{m-1}$ with the aid of a one-step method, e.g., a Runge-Kutta method.
- One computes y_1 using a one-step method, then one uses a two-step method for computing y_2, and so on. Finally, one can apply the m-step method which one is interested in.

Another class of multistep methods can be constructed from other interpolation data. Instead of only using informations from previous steps, namely $(t_k, f_k), \ldots, (t_{k+1-m}, f_{k+1-m})$, let us additionally use the point (t_{k+1}, f_{k+1}). Since we have now $m+1$ interpolation points, the resulting polynomial has degree m. Carrying out the integration as above we obtain, by using the Lagrange ansatz,

$$y_{k+1} - y_k = h\Big(\bar{\beta}_0 f_{k+1} + \sum_{i=1}^{m} \bar{\beta}_i f_{k+1-i}\Big).$$

The second term on the right-hand side is exactly as before, but there is one additional term,

$$h\bar{\beta}_0 f_{k+1} = h\bar{\beta}_0 f(t_{k+1}, y_{k+1}).$$

This gives rise to a problem, because the new value y_{k+1} to be computed appears on both sides of the equation. This is why such a method is called an *implicit* one. In order to compute y_{k+1}, one needs to solve a nonlinear (system of) equation(s). The methods constructed this way are known as *Adams-Moulton methods*. They were invented by J.C. Adams in 1855 but published only in 1926 by the French mathematician F. Moulton.

Some coefficients of the Adams-Moulton methods are presented in Table 2. Note that, in practice, a representation using the Newton ansatz is preferred. Because of its similarity to Euler's method, the first formula for $m = 0$ is known as *implicit Euler's method*. In order to

m	β_0	β_1	β_2	β_3
0	1			
1	$\frac{1}{2}$	$\frac{1}{2}$		
2	$\frac{5}{12}$	$\frac{2}{3}$	$-\frac{12}{12}$	
3	$\frac{3}{8}$	$\frac{19}{24}$	$-\frac{5}{24}$	$\frac{1}{24}$

Table 2: Some coefficients for Adams-Moulton methods

distinguish between these to methods, the former method carries usually the attribute "explicit". The second row describes another very important method which is called the *trapezoidal rule*.

In order to avoid the difficulties with an implicit method, one uses the following ideas for handling Adams-Moulton methods:

(i) Compute $y_{k+1}^{(p)}$ by using the m-step (explicit) Adams-Bashforth method,

$$y_{k+1}^{(p)} = y_k + h\sum_{i=1}^{m} \beta_i f_{k+1-i}.$$

(ii) Compute a preliminary value for the derivative,

$$f_{k+1}^{(p)} = f(t_{k+1}, y_{k+1}^{(p)}).$$

(iii) Replace f_{k+1} by the predicted value $f_{k+1}^{(p)}$ in the m-step Adams-Moulton method,

$$y_{k+1} = y_k + h\left(\bar{\beta}_0 f_{k+1}^{(p)} + \sum_{i=1}^{m} \bar{\beta}_i f_{k+1-i}\right).$$

The resulting method is completely explicit. It is called an *Adams predictor-corrector method* or shorter an Adams PC method. There are very efficient implementations of this class of methods available, among the MATLAB's function `ode113`. Practically, it is not necessary that both methods have the same number of steps. Very often, an m-step Adams-Bashforth method is combined with an $m-1$-step Adams-Moulton method. The reason for this pairing will become clear when the accuracy of the different methods is investigated.

4.2 Consistency, Convergence, Stability

The main question for a finite difference method is the behavior of the global error

$$e_k = y_k - y(t_k).$$

More precisely, we are interested in knowing how fast this error goes to zero if the step size h goes to zero. Even more basic is the question if it goes to zero at all. As we have seen in the

case of one-step methods, the global error is not available immediately. The same holds true for multistep methods. In contrast, it is relatively easy to obtain an estimate of the local error,

$$l_{k+1} = y_{k+1} - y(t_{k+1}),$$

which appears when computing y_{k+1} if the previous values are assumed to be exact values on the solution curve, i.e., $y_k = y(t_k), \ldots, y_{k+1-m} = y(t_{k+1-m})$.

We write down a multistep method a little bit more general than before:

$$y_{k+1} = \sum_{i=1}^{m} \alpha_i y_{k+1-i} + h \sum_{i=0}^{m} \beta_i f_{k+1-i}.$$

For all Adams methods, we have for example $\alpha_1 = 1$ and $\alpha_2 = \cdots = \alpha_m = 0$. Moreover, an explicit method has $\beta_0 = 0$. Then we define

$$\begin{aligned} l_{k+1} &= y_{k+1} - y(t_{k+1}) \\ &= \sum_{i=1}^{m} \alpha_i y(t_{k+1-i}) + h \sum_{i=0}^{m} \beta_i f(t_{k+1-i}, y(t_{k+1-i})) - y(t_{k+1}). \end{aligned}$$

A multistep method has the *order of accuracy* (or *order of consistency*, or simply *order*) p if

$$l_k = O(h^{p+1}).$$

Example 4.1. We compute the order of the two-step Adams-Bashforth method

$$y_{k+1} = y_k + \frac{h}{2}(3f_k - f_{k-1}).$$

The definition of the local error yields

$$\begin{aligned} l_{k+1} &= y(t_k) + \frac{h}{2}(3f(t_k, y(t_k)) - f(t_{k-1}, y(t_{k-1}))) - y(t_{k+1}) \\ &= y(t_k) + \frac{h}{2}(3y'(t_k) - y'(t_{k-1})) - y(t_{k+1}). \end{aligned}$$

Using $h = t_{k+1} - t_k = t_k - t_{k-1}$ we may write down the following Taylor expansions:

$$\begin{aligned} y(t_{k+1}) &= y(t_k) + hy'(t_k) + \frac{h^2}{2}y''(t_k) + O(h^3), \\ y'(t_{k-1}) &= y'(t_k) - hy''(t_k) + O(h^2). \end{aligned}$$

Introduce these Taylor expansions into the representation of the local error:

$$\begin{aligned} l_{k+1} &= y(t_k) + \frac{h}{2}\big(3y'(t_k) - y'(t_k) + hy''(t_k) + O(h^2)\big) - y(t_k) - hy'(t_k) - \frac{h^2}{2}y''(t_k) + O(h^3) \\ &= O(h^3). \end{aligned}$$

This means that this method has order 2. □

As we have seen earlier, Euler's method is equivalent to the Adams-Bashforth method with $m = 1$. Moreover, Euler's method has the order one. More generally, one can show that

- the m-step Adams-Bashforth method has order $p = m$;
- the m-step Adams-Moulton method has order $p = m+1$.

In the case of Runge-Kutta methods, that the global error has the estimate $e_k = O(h^p)$ for a pth order method. Our wish is to have a similar estimate for multistep methods. Unfortunately, this is not true in general.

Example 4.2. This example is a very famous one provided by G. Dahlquist. Let us consider two-step methods, $m = 2$. We know already that a high order method can save a lot of computation time. Therefore, we try to construct a method which has the highest order among all explicit two-step methods. By using Taylor expansions as above one obtains the following method with the highest possible order:

$$y_{k+1} = -4y_k + 5y_{k-1} + h(4f_k + 2f_{k-1}).$$

The order of this method is 3. We apply this method for solving the initial value problem

$$y' = -y, \quad y(0) = 1.$$

The exact solution is $y(t) = e^{-t}$. In order to avoid the problems concerned with the start computations we apply the exact initial values

$$y_0 = 1, \quad y_1 = e^{-h},$$

where h denotes the step size. We compute 100 steps with a step size of $h = 0.01$. The resulting MATLAB code is as follows:

```
clear
h = 0.01;
n = 101;
y = zeros(n,1);
y(1) = 1;
y(2) = exp(-h);
for j = 3:n
   y(j) = -4*y(j-1)+5*y(j-2)-h*(4*y(j-1)+2*y(j-2));
end
t = (0:n-1)'*h;
format short e
[(0:n-1)',y]
```

The following table contains extracts from the results.

k	y_k
0	1.0000e+00
1	9.9005e-01
2	9.8020e-01
3	9.7045e-01
4	9.6079e-01
5	9.5123e-01
⋮	⋮
96	-1.0235e+57
97	5.1485e+57
98	-2.5897e+58
99	1.3026e+59
100	-6.5524e+59

Since $t_{100} = 1$, the exact solution is $y(1) = 1/e \approx 0.367879$. The numerical solution explodes. It has nothing to do with the analytical solution. □

What we have seen in the previous example is a typical stability problem for a numerical algorithm: Small perturbations (in our case the local error) lead to huge errors in the final result. How can one decide if a multistep method is numerically stable? The answer was given by G. Dahlquist.

To every multistep method, we assign a polynomial which will be derived from its coefficients. More precisely, let

$$y_{k+1} = \sum_{i=1}^{m} \alpha_i y_{k+1-i} + h \sum_{i=0}^{m} \beta_i f_{k+1-i}$$

be the given multistep method. Then, let

$$\rho(\zeta) = \xi^m - \alpha_1 \zeta^{m-1} - \alpha_2 \zeta^{m-2} - \cdots - \alpha_m.$$

ρ is a polynomial of ζ. The multistep method is called *stable* if all solutions to $\rho(\zeta) = 0$ lie within the unit circle $|\zeta| \leq 1$, and those with $|\zeta| = 1$ are simple.

The final result is: If a method is stable and if it has the order of accuracy p, then the global error can be estimated by

$$e_k = O(h^p)$$

provided the step size is sufficiently small.

Example 4.3. **One-step methods** Here, it holds $m = 1$ and $\alpha_1 = 1$. This gives the polynomial $\rho(\zeta) = \zeta - 1$. The only root is $\zeta_1 = 1$. Hence, one-step methods are stable.

Adams methods For an m-step Adams method, we have $\alpha_1 = 1$, $\alpha_2 = \cdots = \alpha_m = 0$. Therefore, the polynomial is $\rho(\zeta) = \zeta^m - \zeta^{m-1}$. The roots are $\zeta_1 = 1$ and $\zeta_2 = \cdots = \zeta_m = 0$. Hence, all Adams methods are stable.

□

Our considerations so far are valid under the assumption that the step size is "sufficiently small". Strictly speaking, they are only valid asymptotically if $h \to 0$. Therefore, this stability concept is often called 0-stability because it is only valid if $h \approx 0$. In reality one is interested in using step sizes as large as possible in order to reduce the computational costs. So it is dubious to rely too much on 0-stability. It is even possible that large step sizes lead to instabilities.

Example 4.4. Consider the differential equation

$$y' = \lambda y, \quad y(0) = 1,$$

where $\lambda < 0$. Apply Euler's method which is 0-stable according to our previous derivations. One obtains

$$\begin{aligned} y_{k+1} &= y_k + h\lambda y_k = (1 + h\lambda) y_k \\ &= (1 + h\lambda)^2 y_{k-1} = \cdots \\ &= (1 + h\lambda)^{k+1} y_0. \end{aligned}$$

The analytical solution of the differential equation is $y(t) = e^{\lambda t}$ which fulfills $y(t) \to 0$ $(t \to 0)$. On the other hand, if h is so large that $|1 + h\lambda| > 1$ (equivalently, $h > -2/\lambda$), then

$$|y_k| \longrightarrow \infty \quad (k \longrightarrow \infty).$$

Hence, the method is numerically unstable. □

4.3 Stiff Differential Equations

The instability which we have seen just before needs some contemplation. Our intention with choosing the step size has been to approximate the differential equation as accurate as desired. The example indicates that the step size can even have an influence on the stability properties. This is something that the step size is not intended for. Does this give rise to practical consequences?

Example 4.5. Let us consider the problem

$$y' = -100y + 100, \quad y(0) = y_0.$$

The exact solution is given by

$$y(t) = (y_0 - 1)e^{-100t} + 1.$$

Assume that we perturb the initial value a little bit. Instead of y_0 we will use the initial value $y_0 + \varepsilon$. The solution is now

$$\begin{aligned} y_\varepsilon(t) &= (y_0 + \varepsilon - 1)e^{-100t} + 1 \\ &= y(t) + \varepsilon e^{-100t}. \end{aligned}$$

Because of $|\varepsilon e^{-100t}| \leq \varepsilon$ for all $t \geq 0$, the problem is well-conditioned with respect to perturbations of the initial values:

$$|y_\varepsilon(t) - y(t)| \leq \varepsilon, \quad t \geq 0.$$

In the present context, this type of well-conditioning is called *Lyapunov stability* or simpler *stability*. Note that this is a property of the continuous problem. It has nothing to do with any numerical method.

As above, consider now Euler's method. This gives

$$y_{k+1} = y_k + h(-100y_k + 100) = (1 - 100h)y_k + 100h.$$

This recursion can be solved explicitly. We obtain

$$y_k = (y_0 - 1)(1 - 100h)^k + 1.$$

Choose for simplicity a special initial value, say $y_0 = 2$. This gives

$$\begin{aligned} y(t) &= e^{-100t} + 1, \\ y_k &= (1 - 100h)^k + 1. \end{aligned}$$

The function $y(t)$ converges very fast toward its limit 1. At $t = 0.1$, it holds already $y(0.1) \approx 1 + 5 \cdot 10^{-5}$. What should we expect of a good method?

- In the beginning around $t = 0$, we will expect very small steps in order to be able to reproduce the fast changes in the analytical solution.
- If t_k has become a little bit larger, say $t \geq 0.1$, it should be possible to use large step sizes and still approximate the solution sufficiently accurately, because the solution is an almost constant function.

Unfortunately, Euler's method does not behave in this way. As we have seen before, the method is only stable if $|1 - 100h| < 1$ (the last example with $\lambda = -100$). This implies a step size of $h < 0.02$ even if the local error is very small. □

A problem where the step size is determined by the stability but not by the accuracy when applying Euler's method, is called *stiff*. This type of problems is frequently met in practice.

Even if the definition of stiffness depends explicitly on Euler's method, the qualitative behavior can be seen with any explicit method. This includes even predictor-corrector methods because they are explicit in nature.

Example 4.6. A chemical reaction between three species can be described by a system of three differential equations

$$\begin{aligned} y_1' &= -k_1 y_1 + k_2 y_2 y_3, \\ y_2' &= k_1 y_1 - k_2 y_2 y_3 - k_3 y_2^2, \quad t \in [0, 1000], \\ y_3' &= k_3 y_2^2, \end{aligned}$$

subject to the initial conditions

$$y(0) = (1,0,0)^T.$$

The variable $y_i(t)$ describes the concentration of species i at time t. The constants in the system are given by

$$k_1 = 0.04, \quad k_2 = 10^4, \quad k_3 = 3 \cdot 10^7.$$

The problem was introduced by Robertson. This is a three-dimensional example of a Lotka-Volterra system. We want to solve the system with the aid of MATLAB. In order to play around with different solvers we use the following function:

```
function res = robbytest(method,Tend)
    kparm = [0.04;1e4;3e7];
    options = odeset('Stats','on','Refine',1,'Reltol',1e-6);
    res = feval(meth,@robertson,[0,Tend],[1;0;0],options,kparm);
%
------------------------------------------------------------
function yp = robertson(t,y,kparm)
    yp = zeros(3,1);
    yp(1) = -kparm(1)*y(1)+kparm(2)*y(2)*y(3);
    yp(2) = kparm(1)*y(1)-kparm(2)*y(2)*y(3)-kparm(3)*y(2)^2;
    yp(3) = kparm(3)*y(2)^2;
```

We start with `ode45` and call the function

```
res = robbytest(@ode45,1000).
```

The function `ode45` returns only meaningless values (`NaN`)! Therefore, we try `ode23`. The computer works with full power but we do not see any result. We will continue our experiments a little bit more cautiously. Instead of using the complete interval we try to run the codes with $t \in [0, \texttt{Tend}]$ where $\texttt{Tend} = 1,2,3,4,10$. The number of steps in every attempt is reported below:

code \ Tend	1	2	3	4	10
ode45	679	1359	2049	2753	crash
ode23	868	1739	2623	3524	9316
ode113	1419	2802	4198	5631	14849

It is also interesting to have a look at the average step size $h = \texttt{Tend}/(\text{number of steps})$:

code \ Tend	1	2	3	4	10
ode45	1.5×10^{-3}	1.5×10^{-3}	1.5×10^{-3}	1.5×10^{-3}	crash
ode23	1.2×10^{-3}	1.2×10^{-3}	1.1×10^{-3}	1.1×10^{-3}	1.1×10^{-3}
ode113	7.0×10^{-4}	7.1×10^{-4}	7.1×10^{-4}	7.1×10^{-4}	6.7×10^{-4}

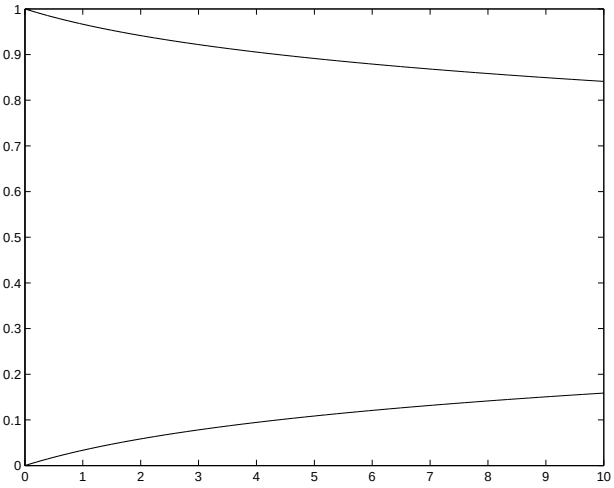

Figure 1: Solution of Robertson's problem

So, the step size is very small all the time. If ones plots the solution obtained in this way (see Figure 1), it looks very friendly and smooth. This is a clear indication of stiffness. MATLAB contains also special methods for stiff systems: `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. We apply those methods to the example above. The corresponding table looks much better now! All these methods are orders of magnitude faster.

code \ `Tend`	1	2	3	4	10	1000
`ode15s`	30	34	38	40	52	135
`ode23s`	16	18	19	20	23	61
`ode23t`	27	37	46	53	80	266
`ode23tb`	18	29	33	39	64	222

□

Note that, from a practical point of view, it is not the number of steps which is decisive for the computing time. The number of function evaluations per step is very different between the different classes of methods, and the evaluation of the right-hand side is where most of the computation time is spent. A much better measure for the efficiency of a code is, therefore, the number of function evaluations of the right-hand side. The codes included in MATLAB provide these numbers, too.

In order to solve stiff equations one must apply *implicit* methods. Let us take the simplest example. The Adams-Moulton method with $m = 0$ is

$$y_{k+1} = y_k + h f_{k+1}.$$

This is Euler's implicit method. In our example we obtain

$$y_{k+1} = y_k + h(-100y_{k+1} + 100),$$

or, more explicitly,

$$y_{k+1} = (1+100h)^{-1}(y_k + 100h).$$

This recursion has the exact solution

$$y_k = (y_0 - 1)(1+100h)^{-k} + 1.$$

For our initial value $y_0 = 2$, this yields

$$y_k = \frac{1}{(1+100h)^k} + 1.$$

Since $\left|\frac{1}{1+100h}\right| < 1$ for all $h > 0$, this method is stable independent of the chosen step size h.

Even the second order Adams-Moulton method ($m = 1$) has the same property for our example. This method (the trapezoidal rule) is the basis for MATLAB's `ode23t` solver. In contrast to that, higher order Adams-Moulton methods are *not* appropriate for stiff problems.

A special class of multistep methods has been constructed which is very well-suited for stiff problems. This class consists of the so-called *BDF* (backward differentiation formulas) methods. They are used in MATLAB's `ode15s`. The construction of these methods goes as follows. Let us write down the differential equation at $t = t_{k+1}$,

$$y'(t_{k+1}) = f(t_{k+1}, y(t_{k+1})).$$

As usual, we assume that the previous approximations $y_k, \ldots, y_{k+1-m}$ are already computed. In order to find an approximation y_{k+1} for $y(t_{k+1})$ we form the interpolation polynomial for $y(t)$ (and *not* for $f(t, y(t))$ as in the case of Adams methods) based on the data

$$(t_{k+1}, y_{k+1}), (t_k, y_k), \ldots, (t_{k+1-m}, y_{k+1-m}).$$

Let this polynomial be $p(t)$. Then we use the approximation

$$p'(t_{k+1}) = f_{k+1}.$$

Obviously, the resulting method is implicit. Representations of these methods can be derived by using Lagrange or Newton interpolation. The result of Lagrange interpolation is a method

$$\sum_{i=0}^{m} \alpha_i y_{k+1-i} = h f_{k+1}.$$

The coefficients for the methods for $m \leq 6$ are given in Table 3. For $m > 6$, the BDF methods are no longer 0-stable, so they cannot be used. Usually, one does not even apply the BDF method with $m = 6$ because it has other unfavorable stability properties. Having a look at the table, we see that, for $m = 1$, Euler's implicit method is derived once again. The order of accuracy of the m-step BDF method is m.

m	α_0	α_1	α_2	α_3	α_4	α_5	α_6
1	1	-1					
2	$\frac{3}{2}$	-2	$\frac{1}{2}$				
3	$\frac{11}{6}$	-3	$\frac{3}{2}$	$-\frac{1}{3}$			
4	$\frac{25}{12}$	-4	3	$-\frac{4}{3}$	$\frac{1}{4}$		
5	$\frac{137}{60}$	-5	5	$-\frac{10}{3}$	$\frac{5}{4}$	$-\frac{1}{5}$	
6	$\frac{49}{20}$	-6	$\frac{15}{12}$	$-\frac{20}{3}$	$\frac{15}{4}$	$-\frac{6}{5}$	$\frac{1}{6}$

Table 3: Coefficients for the BDF methods

4.4 Differential-Algebraic Equations

In the first labwork we applied a method for the computation of electrical circuits, namely the MNA method. At that place, we were only interested in the stationary case. If we are interested in the in-stationary (or, transient) behavior of a circuit, the MNA equations form a system which consists of differential equations as well as equations which did not contain any derivatives of the unknown functions. This is of fundamental difference to the problems which we considered in the present section were all independent functions appear differentiated on the left-hand side. In order to reduce the problem to the form we considered here, one could be tempted to eliminate all variables which do not belong to differential equations. This is the way taken in the course on Analog Electronics. Another idea could be to differentiate the algebraic equations such that all variables appear differentiated finally. Both approaches are hard to realize in practice where the number of unknowns may reach some millions. Therefore, one tries to construct numerical methods which can be applied to solve these mixed systems which are often called *differential-algebraic equations*. It turns out that the BDF methods are among the methods of choice even for such systems. In MATLAB, `ode15s` is the first choice.

Example 4.7. Two typical examples illustrate the appearance of differential-algebraic equations.

MNA equations As we have seen before, the MNA equations can be written down as

$$\begin{pmatrix} A_C C A_C' & 0 & 0 \\ 0 & L & 0 \\ 0 & 0 & 0 \end{pmatrix} \frac{d}{dt} \begin{pmatrix} e \\ i_L \\ i_V \end{pmatrix} + \begin{pmatrix} A_R G A_R' & A_L & A_V \\ -A_L' & 0 & 0 \\ A_V' & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ i_L \\ i_V \end{pmatrix} = \begin{pmatrix} -A_I I \\ 0 \\ V \end{pmatrix}.$$

It is obvious that there are no differential equations for i_V.

Constrained mechanical systems Roughly speaking, Newton's law leads to differential equations while holonomic constraints give rise to algebraic relations. As a simple example we consider the mathematical planar pendulum in Cartesian coordinates. We assume that the pendulum with length l is fixed at the origin $x = y = 0$. The Lagrange formalism leads to

the following Euler-Lagrange equations:

$$\begin{aligned} m\ddot{x} &= -2x\lambda, \\ m\ddot{y} &= -mg - 2y\lambda, \\ 0 &= x^2 + y^2 - l^2. \end{aligned}$$

Here, m is the mass and g is the gravity constant. The Lagrange parameter λ has a physical interpretation: It describes the constraint forces in the string.

□