Numerical Simulation of Dynamic Systems VIII

Prof. Dr. François E. Cellier
Department of Computer Science
ETH Zurich

March 19, 2013

Symbolic Method Coefficients

You may have noticed that I presented the coefficients of the linear multi-step methods always as *rational numbers*. Yet, if I use **Matlab** to compute the coefficient vector, e.g. with the formula:

$$\mathbf{coef} = \mathrm{sum}(\mathrm{inv}(\mathbf{M}));$$

Matlab will return *real-valued numbers* truncated to a given number of digits after the comma.

How did I compute the rational numbers? A first idea might be to remember that both the determinant and the adjugate matrix of a matrix with integer coefficients are integer-valued. This means, we might try to compute:

$$\begin{aligned} \mathit{Mdet} &= \mathrm{round}(\mathrm{det}(\mathbf{M})); \\ \mathbf{Madj} &= \mathrm{round}(\mathit{Mdet} * \mathrm{inv}(\mathbf{M})); \\ \mathbf{coef\_num} &= \mathrm{sum}(\mathbf{Madj}); \end{aligned}$$

Unfortunately, this approach will only work for low-order methods, because the determinant is growing so fast that we quickly exhaust the range of integer numbers.

Symbolic Method Coefficients II

A better approach makes use of **Matlab's symbolic toolbox**, an implementation of a **Maple** kernel accessible using a Matlab interface.

We can write:

$$\textbf{coef} = \mathrm{sum}(\mathrm{inv}(\mathrm{sym}(\mathbf{M})));$$

This is how all of the coefficients of the linear multi-step methods offered in the previous presentation were computed. The sym-function converts the coefficients of the **M**-matrix to text strings. The inv- and sum-functions have been overloaded to operate on *symbolic data structures* as well as on *numerical data structures*.

Newton Iteration

We noticed that many of the most interesting linear multi-step methods, such as the BDF algorithms, ar implicit algorithms. Consequently, we need to iterate during each step using Newton iteration.

We can write the BDF methods in the form:

$$\mathbf{x}_{k+1} = \alpha_i \; h \cdot \mathbf{f}_{k+1} + \sum_{j=1}^{i} \beta_{ij} \; \mathbf{x}_{k-j+1}$$

Plugging in the linear homogeneous problem and solving for $\mathbf{x}_{\mathbf{k+1}}$, we find:

$$\mathbf{x}_{\mathbf{k+1}} = -\left[\alpha_i \cdot (\mathbf{A} \cdot h) - \mathbf{I}^{(\mathbf{n})}\right]^{-1} \cdot \sum_{j=1}^{i} \beta_{ij} \; \mathbf{x}_{k-j+1}$$

On a non-linear problem, we cannot apply matrix inversion. Instead, we formulate a zero-crossing function:

$$\mathcal{F}(\mathbf{x}_{k+1}) = \alpha_i \; h \cdot \mathbf{f}(\mathbf{x}_{k+1}, t_{k+1}) - \mathbf{x}_{k+1} + \sum_{j=1}^{i} \beta_{ij} \; \mathbf{x}_{k-j+1} = 0.0$$

Newton Iteration II

We apply Newton iteration to the zero-crossing function, $\mathcal{F}(\mathbf{x}_{k+1})$, iterating on the unknown $\mathbf{x}_{k+1}$:

$$\mathbf{x}_{k+1}^{\ell+1} = \mathbf{x}_{k+1}^{\ell} - [\mathcal{H}^{\ell}]^{-1} \cdot [\mathcal{F}^{\ell}]$$

where: $\mathcal{H}$ is the *Hessian matrix of the vector function*, $\mathcal{F}$:

$$\mathcal{H} = \alpha_i \cdot (\mathcal{J} \cdot h) - \mathbf{I}^{(\mathbf{n})}$$

with $\mathcal{J}$ being the *Jacobian matrix of the system*.

Newton Iteration III

- Most professional multi-step codes do not reevaluate the Jacobian during each step.
- They use the error estimate of the method as an indicator when the Jacobian needs to be reevaluated.
- As long as the error estimate remains approximately constant, the Jacobian is still acceptable. However, as soon as the error estimate starts to grow, indicating the need for a change in step size, this is a clear indication that a new Jacobian computation is in order.
- Only if a reevaluation of the Jacobian doesn't get the error estimate back to where it was before, will the step size of the method be adjusted.
- Even the Hessian is not reevaluated frequently. The Hessian needs to be recomputed either if the Jacobian has been reevaluated, or if the step size has just been modified.

Newton Iteration IV

Most professional codes offer several options for how to treat the Jacobian. The user can choose between:

1. providing an analytical expression for the Jacobian,
2. having the full Jacobian evaluated by means of a numerical approximation, and
3. having only the diagonal elements of the Jacobian evaluated by means of a numerical approximation ignoring the off-diagonal elements altogether.

Both the convergence speed and the convergence range of the Newton iteration scheme are strongly influenced by the quality of the Jacobian.

- A diagonal approximation is cheap, but leads to a heavy increase in the number of iterations necessary for the algorithm to converge, and necessitates more frequent Jacobian evaluations as well. In our experience, it hardly ever pays off to consider this option.
- The full Jacobian is usually determined by first-order approximations. The i^{th} state variable, x_i, is perturbed by a small value, Δx_i. The state derivative vector is then reevaluated using the modified state value. We find:

$$\frac{\partial \mathbf{f}(\mathbf{x}, t)}{\partial x_i} \approx \frac{\mathbf{f}_{\mathbf{pert}} - \mathbf{f}}{\Delta x_i}$$

Newton Iteration V

- Computing an estimate of the full Jacobian calls for n additional function evaluations, one for each state variable, x_i.
- Yet, even by spending these additional n function evaluations, we gain only a first-order approximation of the Jacobian.
- Using the first-order approximation, we usually need three to four iterations in order to get the *iteration error* down to a value below the *integration error*.
- For these reasons, we strongly advocate the analytical option. An n^{th}-order model calls for n^2 additional equations in order to analytically describe its Jacobian.
- The additional equations can, however, be generated automatically by the model compiler. There is no need to burden the user with this task.

Cost vs. Accuracy of AB Algorithms

Let us simulate a linear fifth-order system using a variety of AB methods with differing fixed step sizes. We compare the global errors of the simulation. For completeness, we also compare with the RK4 algorithm.

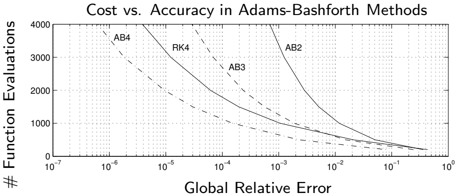

Figure: Cost versus accuracy for different ABi algorithms

Cost vs. Accuracy of AB Algorithms II

We notice that:

- For the same number of function evaluations (i.e., for the same "price" of the simulation), we obtain approximately an improvement in accuracy by a factor of 10, if we increase the order of the algorithm by one.

- Using a method of order n, we can economically obtain an accuracy of 10^{-n}.

- In this example, AB4 is more economical than RK4, as we are dealing with a linear system without discontinuities. However, if we also take into consideration the step-size control, the situation changes, because step-size control is much cheaper for single-step than for multi-step methods.

Order Control

- Order control is very cheap in linear multi-step methods.
- Proceeding from one integration step to the next, we simply throw away the oldest support value in the tail.
- If we wish to *increase the order* of the method by one, we keep the oldest support value for the next step.
- If we wish to *decrease the order* of the method by one, we throw the two oldest support values in the tail away.
- For this reason, order control is very fashionable in multi-step ODE solvers.
- Most professional multi-step codes start the simulation with one step of order one and then increase the order in each subsequent step by one, until the error estimate of the method stops decreasing.

Step-size Control

The linear multi-step algorithms were designed under the assumption of *equidistant sampling*. If we *change the length of the integration step-size* from one step to the next, this assumption no longer holds.

If we like to make use of multi-step ODE solvers with step-size control, we need to employ a trick. Once again, our Newton-Gregory polynomials will save the day.

We start with a backward Newton-Gregory polynomial of the state vector:

$$\mathbf{x}(t) = \mathbf{x}_\mathbf{k} + s\nabla\mathbf{x}_k + \left(\frac{s^2}{2} + \frac{s}{2}\right)\nabla^2\mathbf{x}_k + \left(\frac{s^3}{6} + \frac{s^2}{2} + \frac{s}{3}\right)\nabla^3\mathbf{x}_k + \ldots$$

We differentiate with respect to time:

$$\dot{\mathbf{x}}(t) = \frac{1}{h}\left[\nabla\mathbf{x}_k + \left(s + \frac{1}{2}\right)\nabla^2\mathbf{x}_k + \left(\frac{s^2}{2} + s + \frac{1}{3}\right)\nabla^3\mathbf{x}_k + \ldots\right]$$

The second derivative can be written as follows:

$$\ddot{\mathbf{x}}(t) = \frac{1}{h^2}\left[\nabla^2\mathbf{x}_k + (s+1)\nabla^3\mathbf{x}_k + \ldots\right]$$

etc.

Step-size Control II

Truncating after the cubic term and evaluating for $t = t_k$, i.e., $s = 0.0$, we obtain:

$$\begin{pmatrix} x_k \\ h \cdot \dot{x}_k \\ \frac{h^2}{2} \cdot \ddot{x}_k \\ \frac{h^3}{6} \cdot x_k^{(\text{iii})} \end{pmatrix} = \frac{1}{6} \cdot \begin{pmatrix} 6 & 0 & 0 & 0 \\ 11 & -18 & 9 & -2 \\ 6 & -15 & 12 & -3 \\ 1 & -3 & 3 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_k \\ x_{k-1} \\ x_{k-2} \\ x_{k-3} \end{pmatrix}$$

The vector to the left of the equal sign is called *Nordsieck vector*, here of third order.

If we have n equidistantly spaced values of a state variable, we can calculate *without loss of precision* in its place the value of the state variable itself together with values of its first $n - 1$ time derivatives at the time instant t_k.

Step-size Control III

It is now easy to change the step size:

$$\begin{pmatrix} x_k \\ h_{\text{new}} \cdot \dot{x}_k \\ \frac{h_{\text{new}}^2}{2} \cdot \ddot{x}_k \\ \frac{h_{\text{new}}^3}{6} \cdot x_k^{(\text{iii})} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{h_{\text{new}}}{h_{\text{old}}} & 0 & 0 \\ 0 & 0 & \left(\frac{h_{\text{new}}}{h_{\text{old}}}\right)^2 & 0 \\ 0 & 0 & 0 & \left(\frac{h_{\text{new}}}{h_{\text{old}}}\right)^3 \end{pmatrix} \cdot \begin{pmatrix} x_k \\ h_{\text{old}} \cdot \dot{x}_k \\ \frac{h_{\text{old}}^2}{2} \cdot \ddot{x}_k \\ \frac{h_{\text{old}}^3}{6} \cdot x_k^{(\text{iii})} \end{pmatrix}$$

The Nordsieck vector is meanwhile expressed in the new step size, h_{new}.

Consequently, we can write:

$$\begin{pmatrix} x_k \\ x_{k-1} \\ x_{k-2} \\ x_{k-3} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & -2 & 4 & -8 \\ 1 & -3 & 9 & -27 \end{pmatrix} \cdot \begin{pmatrix} x_k \\ h_{\text{new}} \cdot \dot{x}_k \\ \frac{h_{\text{new}}^2}{2} \cdot \ddot{x}_k \\ \frac{h_{\text{new}}^3}{6} \cdot x_k^{(\text{iii})} \end{pmatrix}$$

We obtain another equidistantly spaced sequence of state values expressed in the new step size, h_{new}.

No precision was lost in the process. The new "state history vector" is accurate to the same order of approximation accuracy as the previous one.

Step-size Control IV

- To change the integration step size, the *state history vector* of each state variable must be multiplied consecutively by three matrices.
- For this reason, changing the step size during the simulation when using a multi-step ODE solver is *quite expensive*.
- A *step-size control algorithm* like the one proposed by Gustafsson cannot be economically implemented for multi-step algorithms.
- We prefer to use a *more conservative step size* to avoid that the step size has to be changed frequently.
- For all these reasons, the *Adams algorithms aren't competitive* for the simulation of non-linear engineering systems.
- In contrast, the *BDF algorithms are competitive* for the simulation of engineering systems due to their ability to deal with *stiff systems*.

The Startup Problem

As we already explained, higher-order multi-step methods cannot be used during the initial steps of the simulation due to their need for support values of the past. They rely on a *state history vector*.

One easy way to tackle this problem, that is being used frequently in practice, is to *change the order* of the algorithm during the simulation.

- We start with an algorithm of order 1. For example, we may start the simulation with one step of BDF1. After the first step, we already have one history point.
- During the second step, we use BDF2. This step generates a second history point.
- During the third step, we use BDF3, etc., until we reach the order of the algorithm that is most appropriate, i.e., that minimizes the error estimate.

This technique has an important disadvantage. In order to obtain results with acceptable accuracy during the first low-order steps, we need to start with a very small step size. Once we have reached the desired method order, we then will have to increase the step size in order to improve the economy of the algorithm.

The Startup Problem II

- Another technique, that has been proposed in the literature is to use a single-step algorithm during the startup phase of the simulation, e.g. an RK algorithm. Since single-step methods are self-starting, we can start the simulation immediately with a method of higher order.

- For example, if we would like to simulate using the BDF4 algorithm, we start with three steps of RK4. After the initial RK4 steps, we have the state history information available that we require to switch over to the corresponding BDF4 algorithm.

- Unfortunately, this technique has the same disadvantage as the previous one, because the problem to be simulated is presumably stiff, since otherwise we wouldn't use a BDF algorithm, and consequently, we'll again have to start out with a very small step size, because the RK4 starter method is a non-stiff ODE solver that is numerically unstable for larger step sizes when dealing with a stiff system.

- RK starters will work well for simulations employing Adams-Moulton, but we already know that Adams-Moulton is not a competitive algorithm for simulating non-stiff non-linear models of engineering systems.

The Startup Problem III

If we decide to use an RK starter, how do we determine the correct step size to use by the multi-step technique after the startup period? Remember that we should hit the correct step size at once, because each new change in the step size is expensive.

Let us redraw the relationship between cost and accuracy of different algorithms, this time using a double-logarithmic scale.

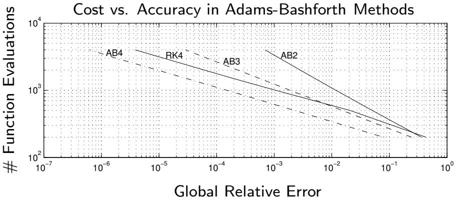

Figure: Cost versus accuracy for different ABi algorithms

We notice that the logarithm of the step size is, for all practical purposes, linear in the logarithm of the accuracy.

The Startup Problem IV

After the startup phase, we can perform one step of the multi-step technique using the step size h_1 from the RK starter, and obtain an error estimate ε_1. We then reduce the step size to $h_2 = h_1/2$ and repeat the step. We obtain a new error estimate ε_2.

We now place a linear curve through the two points, and interpolate (extrapolate) with the desired accuracy ε_{des} to obtain a good value for the true step size to be used by the multi-step algorithm:

$$\begin{pmatrix} \ln(h_1) \\ \ln(h_2) \end{pmatrix} = \begin{pmatrix} \ln(\varepsilon_1) & 1 \\ \ln(\varepsilon_2) & 1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

$$\ln(h_{\text{des}}) = a_1 \cdot \ln(\varepsilon_{\text{des}}) + a_2$$

$$h_{\text{new}} = 0.8 \cdot h_{\text{des}}$$

The step size h_{des} should be fairly close to the optimal step size. I introduced a fudge factor of 0.8 to be on the safe side, i.e., I prefer to use a step size that is slightly smaller than the optimal one to avoid having to change the step size rapidly again.

The Startup Problem V

- A better approach, not yet implemented in any professional BDF code, might be to use a BI algorithm, such as $BI4/5_{0.45}$, as a starter method.
- $BI4/5_{0.45}$ is a single-step algorithm, and therefore, this algorithm is self-starting.
- Yet, $BI4/5_{0.45}$ is also a stiff system solver, and consequently, we may be able to employ a larger step size already during the startup phase.

The Dense Output Problem

We would like to be able to obtain state (output) values at any point in time, also in between sampling instants. The values obtained should be accurate to the same order of approximation accuracy as the values at the sampling points. This feature is called *dense output*.

Since changing the step size of the simulation is expensive in a multi-step algorithm, we wish to obtain dense output without changing the step size.

To this end, we shall integrate across the communication instant to the next regular sampling instant, and then use *interpolation* to find the desired state (output) values at the communication instant.

Once again, the *Nordsieck vector* comes to our rescue. Once we passed the next *communication instant*, i.e., the next instant for which we wish to know the values of the states (outputs), we calculate the Nordsieck vector at the current sampling instant, and, using that information, employ a Taylor series expansion to determine the state (output) information at the communication instant. This can be done with the same accuracy as the integration itself.

After calculating the states (outputs) at the communication instant in this fashion, we continue with our simulation from the sampling instant using the same step size as before.

The Dense Output Problem II

Techniques for obtaining *dense output* were also developed for some *single-step algorithms*.

Unfortunately, we don't have access to the Nordsieck vector in single-step methods.

We cannot even use the intermediate computations (i.e., the predictor stages) to estimate the states (outputs) at the communication instant, because in most single-step methods, the states calculated by the predictor stages are not of the same order of approximation accuracy as those of the corrector stage.

One successful and economic method for computing dense output in single-step algorithms is to make use of *three embedded algorithms*, i.e., three methods of the same order of approximation accuracy that share as many stages with each other as possible.

Using the three approximations of the state variables at time t_{k+1}, we can obtain an approximation of the state vector at any time instant $t \in [t_k, t_{k+1}]$ that is accurate to the same order of approximation accuracy as the three individual algorithms themselves.

Conclusions

In this presentation, we looked at the issues of *step-size and order control* in multi-step algorithms. The problems surrounding step-size and order control are quite different in the case of multi-step algorithms than in the case of single-step algorithms. Whereas step-size control is easy and cheap in single-step methods, it is quite involved and expensive for multi-step methods. In contrast, order control is much cheaper and simpler for multi-step than for single-step algorithms.

We have also discussed the *startup problems* that are specific to multi-step algorithms. Contrary to the single-step algorithms, which are self-starting, multi-step algorithms require a separate startup procedure. This makes multi-step algorithms not suited for simulating systems with heavy discontinuities that call for a restart after each discontinuity.

Finally, we looked at the *readout problem*, i.e., how to get output at communication points that don't coincide with sampling instants. Once again, the methods used in multi-step algorithms are quite different from those used in single-step algorithms. Single-step algorithms will usually reduce the step size around communication points, whereas multi-step algorithms will simulate across communication points and interpolate back using the Nordsieck vector that is implicitly available for multi-step algorithms, but not for single-step methods.

Conclusions II

It turns out that the actual integration algorithm occupies not more than about 5% of the lines of code in a professional multi-step ODE solver. The remaining lines of code are for all of the secondary algorithms surrounding the actual integration method, such as step-size and order control, the startup procedure, the readout procedure, and the localization of discontinuities.

References

1. Hu, Luoan (1991), *DBDF: An Implicit Numerical Differentiation Algorithm for Integrated Circuit Simulation*, MS Thesis, Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, AZ.