Extrapolation Methods in Mathematica

Mark Sofroniou
Wolfram Research, Champaign, Illinois, USA.

Giulia Spaletta
Mathematics Department, Bologna University, Italy.

Received 22 December, 2007; accepted in revised form 28 January, 2008

Abstract: This article outlines design and implementation details of the framework for one step methods for solving ordinary differential equations in Mathematica. The solver breaks up the solution into three main phases for equation processing and classification, numerical solution and processing of results. One of the distinguishing features of the framework is the hierarchical nature of the method invocation which allows for simple construction of composed integration schemes. A plug-in facility for user defined schemes is also provided. Highly accurate reference solutions can also be obtained by making use of arbitrary precision software arithmetic. Issues relating to appropriate formulation and efficient implementation will also be discussed, together with strategies for automatic method, order and parameter selection.

Keywords: Ordinary differential equations; initial value problems; numerical differential equations; numerical integration; extrapolation methods; rounding error accumulation; NDSolve.

Mathematics Subject Classification: 65L05, 65L06, 65P10, 68W30.

PACS: 02.60Lj, 02.70Wz, 05.45Pq, 45.20Jj, 45.40Cc.

1 Introduction

A new framework for the numerical solution of differential equations was introduced in version 5 of Mathematica. The goal was to provide a variety of methods in a uniform, efficient and interactive environment. The design builds on a number of ideas found in existing software and can be summarized as follows.

- Each method has its own data structure encapsulating necessary information such as coefficients, workspaces and so on. This is a generalisation of ideas found in LSODA [23, 29].
- A method construction and initialisation phase is carried out before the numerical integration is attempted.

1Published electronically March 31, 2008
2marks@wolfram.com
3giulia@cs.unibo.it
The framework includes a plug-in mechanism for the specification of method coefficients. This generalises the ideas in RKSUITE [4].

Users can write their own plug-in methods as modules which can be incorporated into the solving environment.

A communication mechanism enables the specification of method properties such as order and symmetry.

A method hierarchy allows one method to call a number of others. This is a generalisation of the ideas used in the Generic ODE Solving System (GODESS) which is implemented in C++ (see [28] and the references therein).

Supporting a large number of numerical integration methods for differential equations is a lot of work. In order to cut down on maintenance and duplication of code, common components are shared between methods. This approach also allows code optimization to be carried out in just a few central routines. The principal features of the NDSolve framework are as follows.

- Uniform design and interface.
- Code reuse (common code base).
- Object orientation (method property specification and communication).
- Data hiding. Each method has its own data object that contains information that is needed for invocation. This includes, but is not limited to, coefficients, workspaces, step-size control parameters, step-size acceptance/rejection information, and Jacobian matrices.
- Dynamic memory allocation means that there are no hard-coded restrictions on dimension and order (only limited by the amount of available memory).
- Separation of method initialization phase and run-time computation.
- Hierarchical numerical methods.
- Uniform treatment of rounding errors.
- Vectorized framework based on a generalization of the BLAS model [1] using optimized in-place arithmetic to reduce repeated allocation overhead.
- Array-based framework that allows families of methods to share one implementation.
- Type and precision is dynamic for all methods.
- Plug-in capabilities that allow user extensibility and prototyping.
- Specialized data structures.

NDSolve is broken up into three components, each of which can be called individually.

- NDSolveProcessEquation parses the differential system and classifies it as a Partial Differential Equation, Initial Value Problem, Differential Algebraic Equation or Boundary Value Problem. It usually constructs a function which represents the right-hand side of the equations in normal form.
- NDSolveIterate initializes the integration method(s), invokes them to advance the solution from the initial conditions and stores the solution data.
Extrapolation Methods in Mathematica

- **NDSolve’ProcessSolutions** takes the stored data and constructs **InterpolatingFunction** objects to represent the solutions.

In this article we discuss some of the design aspects in some detail with particular emphasis on the method of extrapolation, which is particularly advantageous when highly accurate solutions are being sought [43, 19, 21, 7]. The article is organised as follows. In §2 we briefly introduce the ideas involved in extrapolation. In §3 we outline some of the design issues that have been considered in the incorporation of extrapolation in the solving environment. Specific implementation issues are discussed in §4. Examples that illustrate the solver framework and the desirable qualitative behaviour of extrapolation methods are given in §5.

## 2 Extrapolation

Extrapolation generalizes the idea of Richardson’s extrapolation to a sequence of refinements. It provides classes of arbitrary order methods with automatic order and step size control. The error estimate comes from computing a solution over an interval, using the same method with a varying number of steps and using extrapolation on the polynomial that fits through the computed solutions, giving a composite higher order method [3]. At the same time, the polynomials give a means of error estimation. Typically, for low precision, extrapolation methods have not been competitive with Runge-Kutta-type methods. For high precision, however, the arbitrary order means that they can be arbitrarily faster than fixed order methods for very precise tolerances. The order and step size control for the implementation in **NDSolve** is roughly based on an amalgamation of the strategies for the codes **odex.f** and **seulex.f** described in [21, 19] (see also [5, 6, 33]).

Consider a differential system

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

Let \( H > 0 \) be a basic step size and choose a monotonically increasing sequence of positive integers \( n_1 < n_2 < n_3 < \cdots < n_k \) and define the corresponding step sizes \( h_1 > h_2 > h_3 > \cdots > h_k \) by \( h_i = H/n_i, i = 1, 2, \ldots, k \). Choose a numerical method of order \( p \) and compute the solution of the initial value problem (1) by carrying out \( n_i \) steps with step size \( h_i \) to obtain:

\[ T_{i,1} = y_{n_i}(t_0 + H), i = 1, 2, \ldots, k. \]

Extrapolation is performed using the Aitken-Neville algorithm by building up a table of values:

\[ T_{i,j} = T_{i,j-1} + \frac{T_{i,j-1} - T_{i-1,j-1}}{n_i^{n_i-1/j-1}} - 1, \quad i = 2, \ldots, k, \quad j = 2, \ldots, i. \]  

where \( w \) is either 1 or 2 depending on whether the base method is symmetric under extrapolation. A dependency graph of the values in (2) illustrates the relationship:

\[
\begin{array}{ccccccc}
T_{11} & & & & & & \\
\downarrow & & & & & & \\
T_{21} & \leftarrow & T_{22} & & & & \\
\downarrow & & & & & & \\
T_{31} & \leftarrow & T_{32} & \leftarrow & T_{33} & & \\
\downarrow & & & & & & \\
T_{41} & \leftarrow & T_{42} & \leftarrow & T_{43} & \leftarrow & T_{44} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

\[ c 2008 \text{ European Society of Computational Methods in Sciences and Engineering (ESCMSE)} \]
Considering \( k = 2, n_1 = 1, n_2 = 2 \) is equivalent to Richardson’s extrapolation. For non-stiff problems the order of \( T_{k,k} \) in (2) is \( p + (k - 1)w \). For stiff problems the analysis is more complicated and involves the investigation of perturbation terms that arise in singular perturbation problems [16, 17].

### 2.1 Extrapolation sequences

Any extrapolation sequence can be specified in our implementation. Some common choices are as follows.

- The Harmonic sequence, \( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots \).
- The sub-Harmonic sequence, \( 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, \ldots \).
- The Romberg sequence, \( 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, \ldots \).
- The Bulirsch sequence, \( 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, \ldots \).

The sequence with the lowest cost is the Harmonic sequence, but this is not without problems since rounding errors are not damped. A sequence that satisfies \( (n_i/n_{i-1})^w \geq 2 \) has the effect of minimizing the roundoff errors for an order-\( p \) base integration method. For a base method with \( w = 1 \) the optimal sequence is the Romberg sequence. For a base method with \( w = 2 \) the optimal sequence is: \( 1, 2, 3, 5, 8, 12, 17, 25, 36, 51, \ldots \).

### 2.2 Rounding error accumulation

For high order extrapolation an important consideration is the accumulation of rounding errors in the Aitken-Neville algorithm (2). As an example consider [21, Exercise 5, Section II.9]. Suppose that the entries \( T_{11}, T_{21}, T_{31}, \ldots \) are disturbed with rounding errors \( \epsilon, -\epsilon, \epsilon, \ldots \) and compute the propagation of these errors into the extrapolation table.

Due to the linearity of the extrapolation process (2), suppose that the \( T_{i,j} \) are equal to zero and take \( \epsilon = 1 \). This shows the evolution of the Aitken-Neville algorithm (2) on the initial data using the Harmonic sequence and a symmetric integration method \( w = 2 \) of order \( p = 2 \).

\[
\begin{array}{cccccc}
1. & 1. & 1.66667 & 1 & 2.6 & 3.13333 \\
1 & -1 & -3.57143 & -5.62857 & -6.2127 & 4.55556 \\
\end{array}
\]

Hence, for an order-sixteen method approximately two decimal digits are lost due to rounding error accumulation. This model is somewhat crude because it is more likely that rounding errors are made in \( T_{i+1,1} \) than in \( T_{i,1} \) for \( i \geq 1 \).

### 2.3 Rounding error reduction

It seems worthwhile to look for approaches that can reduce the effect of rounding errors in high order extrapolation. Selecting a faster growing step sequence to diminish rounding errors is one approach, although the drawback is that the number of integration steps increases when computing the \( T_{i,1} \) for the first column of the extrapolation table which requires more work.
Some codes, such as STEP, take active measures to reduce the effect of rounding errors for stringent tolerances [34]. An alternative strategy, which does not appear to have received a great deal of attention in the context of extrapolation, is to modify the base integration method in order to reduce the magnitude of the rounding errors in floating-point operations. This approach, based on ideas that date back to [11] and used to good effect for the two body problem in [9], are explained in the next section. For more background see also [24, 26, 27, 44].

3 Design description

In the NDSolve framework the method Extrapolation is a controller method. It requires the specification of a base method, using the Method option, to actually carry out the numerical integration. The following methods are the most common choices for base integrators in extrapolation.

- ExplicitEuler explicit Euler method
- ExplicitMidpoint explicit midpoint method
- ExplicitModifiedMidpoint includes Gragg smoothing step (11)
- LinearlyImplicitEuler linearly implicit Euler method
- LinearlyImplicitMidpoint linearly implicit midpoint method (Bader-Deuflhard formulation [2])
- LinearlyImplicitModifiedMidpoint includes smoothing step (18)

For efficiency, these have been implemented in the Mathematica kernel and are also callable via the Method option as individual methods. The implementation of these methods has a special interpretation for multiple substeps within Extrapolation.

The NDSolve framework for one step methods uses a formulation that returns the increment or update to the solution. This is advantageous in the context of geometric numerical integration where numerical errors are not damped over long time integrations. It also allows the application of efficient correction strategies such as compensated summation (see [20, 22, 36] and the references therein for more details). It will be shown in the sequel that this formulation is also useful in the context of extrapolation. The base methods are now described together with the increment reformulation that is used to reduce rounding error accumulation.

3.1 Multiple Euler steps

Given \( t_0, y_0 \) and \( H \), consider a succession of \( n = n_k \) integration steps with step size \( h = H/n \) carried out using Euler’s method:

\[
y_i = y_{i-1} + h f(t_{i-1}, y_{i-1}), \quad i = 1, \ldots, n, \tag{3}
\]

where \( t_i = t_0 + i h \). It is well-known that, for certain base integration schemes, the entries \( T_{i,j} \) in the extrapolation table produced from (2) correspond to explicit Runge-Kutta methods. For example, (3) is equivalent to an \( n \)-stage explicit Runge-Kutta method:

\[
k_i = f(t_0 + c_i H, y_0 + H \sum_{j=1}^{n} a_{i,j} k_j), \quad i = 1, \ldots, n,
\]

\[
y_n = y_0 + H \sum_{i=1}^{n} b_i k_i
\]

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)
Figure 1: Addition of two fixed precision floating point registers $y_i$ and $\Delta y_i$ which differ in magnitude by $\delta$ bits. The corresponding $\delta$ bits that are truncated in $\Delta y_i$ during addition are highlighted in gray.

where the coefficients are represented by the Butcher table:

\[
\begin{array}{c|cccc}
0 & 1/n & 1/n & & \\
1/n & 1/n & & & \\
\vdots & \vdots & \ddots & & \\
(n-1)/n & 1/n & \cdots & 1/n & \\
(n-1)/n & 1/n & \cdots & 1/n & 1/n \\
\end{array}
\]  

(5)

For $i \geq 0$ define:

$$\Delta y_i = y_{i+1} - y_i,$$

$$\Sigma \Delta y_i = \Sigma \Delta y_{i-1} + \Delta y_{i-1} = \sum_{j=0}^{i-1} \Delta y_j \quad \text{and} \quad \Sigma \Delta y_0 = 0.$$

Then the integration procedure (3) can be rewritten to reflect the correspondence with an explicit Runge-Kutta method (4)-(5) as:

$$\Delta y_{i-1} = h f(t_{i-1}, y_0 + \Sigma \Delta y_{i-1}), \quad i = 1, \ldots, n.$$  

(6)

where terms in the right-hand side of (6) are now considered as departures from the same value $y_0$. The result at the end of the step is recovered as:

$$y_n = y_0 + \Sigma \Delta y_n.$$  

(7)

The $\Delta y_{i-1}$ in (6) correspond to $hk_i$ in (4)-(5). Mathematically the formulations (3) and (6)-(7) are equivalent. For $n > 1$, however, the computations in (6) have the advantage of accumulating a sum of smaller $O(h)$ quantities, or increments, which reduces rounding error accumulation in finite-precision floating-point arithmetic. Graphical illustration of the process is given in Figure 1.

### 3.2 Multiple explicit midpoint steps

Expansions in even powers of $h$ are extremely important for an efficient implementation of Richardson’s extrapolation, and an elegant proof is given in [42]. Consider a succession of integration steps $n = 2n_k$ with step size $h = H/n$ using one Euler step followed by multiple explicit midpoint steps:

$$y_1 = y_0 + h f(t_0, y_0),$$

$$y_i = y_{i-2} + 2h f(t_{i-1}, y_{i-1}), \quad i = 2, \ldots, n.$$  

(8)

When (8) is computed with $2n_k - 1$ midpoint steps, then the method has a symmetric error expansion [12, 42]. Some implementations of (8) require that a doubled sequence of values be
specified, such as the double-Harmonic sequence 2, 4, 6, 8, ... For reasons of modularity, in the \texttt{NDSolve} framework the sequence is just specified as Harmonic and the values are doubled internally in the method as required. Reformulation of (8) can be accomplished in terms of increments as:

\[
\begin{align*}
\Delta y_0 &= h f(t_0, y_0), \\
\Delta y_{i-1} &= 2 h f(t_{i-1}, y_0 + \Delta y_{i-1}) - \Delta y_{i-2}, \quad i = 2, \ldots, n.
\end{align*}
\]  

(9)

The smoothing step of Gragg has its historical origins in the weak stability of the explicit midpoint rule:

\[
S y_h(n) = 1/4(y_{n-1} + 2y_n + y_{n+1})
\]  

(10)

In order to make use of (10), 2 \(n_k\) midpoint steps are taken in (8). This has the advantage of increasing the stability domain and evaluating the function at the end of the basic step [21]. Because of the construction in (9), a sum of increments is available at the end of the algorithm together with two consecutive increments. This leads to the following formulation:

\[
S \Delta y_h(n) = S y_h(n) - y_0 = \Sigma \Delta y_n + 1/4(\Delta y_n - \Delta y_{n-1}).
\]  

(11)

Moreover (11) has an advantage over (10) in finite-precision arithmetic because the values \(y_i\), which typically have a larger magnitude than the increments \(\Delta y_i\), do not contribute to the computation. Gragg’s smoothing step is not of great importance if the method is followed by extrapolation, and Shampine proposes an alternative smoothing procedure that is slightly more efficient [35].

3.3 Implicit differential equations

A generalization of the differential system (1) arises in many situations such as the spatial discretization of parabolic partial differential equations:

\[
M y'(t) = f(t, y(t)), \quad y(t_0) = y_0,
\]  

(12)

where \(M\) is a constant matrix that is often referred to as the mass matrix. Base methods in extrapolation that involve the solution of linear systems of equations can be easily modified to solve problems of the form (12).

3.4 Multiple linearly implicit Euler steps

Increments arise naturally in the description of many semi-implicit and implicit methods. Consider a succession of integration steps carried out using the linearly implicit Euler method for the system (12) with \(n = n_k\) and \(h = H/n\).

\[
(M - h J) \Delta y_{i-1} = h f(t_{i-1}, y_{i-1}), \quad y_i = y_{i-1} + \Delta y_{i-1}, \quad i = 1, \ldots, n.
\]  

(13)

Here \(M\) denotes the mass matrix and \(J\) denotes the Jacobian of \(f\):

\[
J = \frac{\partial f}{\partial y}(t_0, y_0).
\]

The solution of the equations for the increments in (13) is accomplished using a single LU decomposition of the matrix \(M - h J\) followed by the solution of triangular linear systems for each right-hand side. Reformulation in terms of increments as departures from \(y_0\) is now:

\[
(M - h J) \Delta y_{i-1} = h f(t_{i-1}, y_0 + \Sigma \Delta y_{i-1}), \quad i = 1, \ldots, n.
\]  

(14)

Notice that (14) and (6) are equivalent when \(J = 0, M = I\).

\footnote{The capability to solve systems of the form (12) was added in version 6 of \textit{Mathematica}.}
3.5 Multiple linearly implicit midpoint steps

Consider one step of the linearly implicit Euler method followed by multiple linearly implicit midpoint steps with \( n = 2^k \), \( h = H/n \), using the formulation of Bader and Deuflhard [2]:

\[
\begin{align*}
(M - h J)\Delta y_0 &= h f(t_0, y_0), \\
y_1 &= y_0 + \Delta y_0, \\
(M - h J)(\Delta y_{i-1} - \Delta y_{i-2}) &= 2(h f(t_{i-1}, y_{i-1}) - \Delta y_{i-2}), \quad i = 2, \ldots, n. \\
y_i &= y_{i-1} + \Delta y_{i-1},
\end{align*}
\]

If (15) is computed for \( 2^k - 1 \) linearly implicit midpoint steps, then the method has a symmetric error expansion [2]. Reformulation of (15) in terms of increments can be accomplished as:

\[
\begin{align*}
(M - h J)\Delta y_0 &= h f(t_0, y_0), \\
(M - h J)(\Delta y_{i-1} - \Delta y_{i-2}) &= 2(h f(t_{i-1}, y_{i-1} + 2\Delta y_{i-1}) - \Delta y_{i-2}), \quad i = 2, \ldots, n. \\
\end{align*}
\]

An appropriate smoothing step for the linearly implicit midpoint rule is [2]:

\[
S_y h(n) = \frac{1}{2}(y_{n-1} + y_{n+1}).
\]

The smoothing step (17) rewritten in terms of increments becomes:

\[
S\Delta y_h(n) = S_y h(n) - y_0 = \Sigma\Delta y_n + \frac{1}{2}(\Delta y_n - \Delta y_{n-1}).
\]

The required quantities are obtained when (16) is run with \( 2^k \) linearly implicit midpoint steps. The smoothing step for the linearly implicit midpoint rule has a different role to Gragg’s smoothing for the explicit midpoint rule [2, 35]. Since there is no weakly stable term to eliminate, the aim is to improve the asymptotic stability.

3.6 Polynomial extrapolation with increments

It has been shown how to modify \( T_{i,1} \), the entries in the first column of the extrapolation table, in terms of increments. However, for certain base integration methods, each of the \( T_{i,j} \) corresponds to an explicit Runge-Kutta method. Therefore, it appears that the correspondence has not yet been fully exploited and further refinement is possible. Since the Aitken-Neville algorithm (2) involves linear differences, the entire extrapolation process can be carried out using increments. This leads to the following modification of the Aitken-Neville algorithm:

\[
\Delta T_{i,j} = \Delta T_{i-1,j-1} + \frac{\Delta T_{i,j-1} - \Delta T_{i-1,j-1}}{\frac{n_{i,j}}{n_{i-1,j+1}} - 1}, \quad i = 2, \ldots, k, \quad j = 2, \ldots, i.
\]

The quantities \( \Delta T_{i,j} = T_{i,j} - y_0 \) in (19) can be computed iteratively, starting from the initial quantities \( \Delta T_{1,1} \) that are obtained from the modified base integration schemes without adding the contribution from \( y_0 \). The final desired value \( T_{k,k} \) can be recovered as \( y_0 + \Delta T_{k,k} \). The advantage is that the extrapolation table is built up using smaller quantities, and so the effect of rounding errors from subtractive cancellation is reduced. The situation is similar to the process outlined in Figure 1, with \( y_i \) replaced by the constant value \( y_0 \) and \( \Delta y_i \) replaced by \( \Delta T_{i,j} \). The digits of \( \Delta T_{i,j} \) which would be truncated to form \( T_{i,j} \) in (2) are preserved in (19).

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)
3.7 Stability check

Extrapolation methods use a large basic step size that can give rise to some difficulties. For example, issues with numerical overflow when solving the van der Pol equation were experienced in [33]. Two forms of stability check are used for the linearly implicit base schemes (for further discussion, see [19]). One check is performed during the extrapolation process. Let \( \text{err}_j = \|T_{j,j-1} - T_{j,j}\| \).

If \( \text{err}_j \geq \text{err}_{j-1} \) for some \( j \geq 3 \), then recompute the step with \( H = H/2 \).

In order to interrupt the method after the computation of \( T_{1,1} \), Deuflhard suggests checking if the Newton iteration applied to a fully implicit scheme would converge. For the implicit Euler method this leads to consideration of quantities \( \Delta_0 \) and \( \Delta_1 \) which satisfy:

\[
(M - h J)\Delta_0 = h f(t_0, y_0), \quad (M - h J)\Delta_1 = h f(t_0, y_0 + \Delta_0) - \Delta_0.
\]

Notice that (20) differs from (13) only in the second equation. It requires finding the solution for a different right-hand side but no extra function evaluation. For the implicit midpoint method an appropriate test is:

\[
\Delta_0 = \Delta y_0 \quad \text{and} \quad \Delta_1 = 1/2(\Delta y_1 - \Delta y_0),
\]

which simply requires a few basic arithmetic operations. If \( \|\Delta_1\| \geq \|\Delta_0\| \) then the implicit iteration diverges, so the step should be recomputed with \( H = H/2 \). Increments are a more accurate formulation for the implementation of both forms of stability check.

4 Implementation issues

There are a number of important implementation issues that should be considered, some of which are mentioned here.

4.1 Method property communication

During the method initialization phase in NDSolve base methods can communicate properties to calling methods. This includes the method order and whether the method is symmetric under extrapolation. Methods which are able to solve (12) communicate it to Extrapolation using the MassMatrixQ method property.

4.2 Jacobian reuse

The Jacobian is evaluated only once for all entries \( T_{i,1} \) at each time step by storing it together with the associated time at which it is evaluated. This also has the advantage that the Jacobian does not need to be recomputed for rejected steps.

4.3 Linear algebra and arbitrary precision

For dense systems LAPACK routines are used [1]. The solvers are usually invoked through a vendor specific optimized library such as Intel’s Math Kernel Library. Sparse matrices and solvers are also available and are used when SparseArray data objects are encountered.

Since extrapolation provides highly accurate solutions, it is ideal as an arbitrary precision reference solver such as for benchmarking. Mathematica has two forms of arbitrary precision software floating point arithmetic. The first model is an extension of significance arithmetic which provides estimates of the propagation of errors [39]. The second model, which is used in NDSolve,
uses arbitrary but fixed precision. The reason is that the methods used in NDSolve contain precise
estimates of the way that errors are measured and propagated. For efficiency, arbitrary precision
computations use low level operations from the GNU Multiple Precision library [13]. The library
contains optimized assembly routines for most architectures. High precision dot products use a
specially written routine to make efficient use of the GMP library, which is accomplished using the
mpn layer. Many practical situations will involve computations in the range of quadruple precision.
Linear system solving in this range can be computed efficiently using double precision solvers
coupled with high precision iterative refinement [10]. More detail on how this is accomplished
automatically from a user perspective can be found in [41].

4.4 Stiffness detection

Extrapolation based on explicit base methods includes a form of stiffness detection that is also used
in ExplicitRungeKutta [30, 31, 32, 37]. The idea is to directly estimate the dominant eigenvalue $\lambda$
of the Jacobian $J$ of a problem. If $t_n$ denotes the current integration time and $h_n$ the current step
size being used, then a suitable value which approximates $\lambda$ can be obtained from:

$$\rho = \frac{\|f(t_n + h_n, T_{2,2}) - f(t_n + h_n, T_{1,1})\|}{\|\Delta T_{2,2} - \Delta T_{1,1}\|}. \quad (21)$$

The denominator in (21) corresponds to the power method applied to $hJ$, which yields a good
approximation to the eigenvector corresponding to the leading eigenvalue $\lambda$. The increment form
is used in the denominator in (21) in order to avoid subtractive cancellation of $y_n$:

Let $j_{\partial S}$ denote the boundary of the linear stability function. Whenever $h_{\partial S}$, then stability rather than local
accuracy is restricting the choice of step size and the problem is considered to be stiff.

4.5 Adaptive order and work estimation

In order to adaptively change the order of the extrapolation throughout the integration, it is important
to have a measure of the amount of work required by the base scheme and extrapolation
sequence. A measure of the relative cost of function evaluations is also advantageous. The dimen-
sion of the system, preferably with a weighting according to structure, needs to be incorporated
for linearly implicit schemes in order to take account of the expense of solving each linear system.

4.6 Fine-tuning

A variety of method options can be used to specify parameters in order to fine-tune the Extrapolation
method.

As with most methods, there is a balance between taking too small a step and trying to take
too big a step that will be frequently rejected. The option StepSizeSafetyFactors specifies safety
factors $\{s_1, s_2\}$ which constrain the choice of step size as follows. The step size chosen by the
method for order $p$ satisfies:

$$h_{n+1} = h_n s_1 \left( s_2 \frac{Tol}{\|err_n\|} \right)^{1/(p+1)} . \quad (22)$$

This includes both an order-dependent factor and an order-independent factor.

The option StepSizeRatioBounds specifies bounds $\{sr_{\text{min}}, sr_{\text{max}}\}$ on the next step size to take
such that:

$$sr_{\text{min}} \leq \left| \frac{h_{n+1}}{h_n} \right| \leq sr_{\text{max}} .$$

An important aspect in Extrapolation is the choice of order. Each extrapolation step $k$ has an
associated work estimate $A_k$. The work estimate for explicit base methods is based on the number
of function evaluations and the step sequence used. The work estimate for linearly implicit base methods also includes an estimate of the cost of evaluating the Jacobian, the cost of an LU decomposition, and the cost of back solving the linear equations.

Estimates for the work per unit step are formed from the work estimate $A_k$ and the expected new step size to take for a method of order $k$, computed from (22): $W_k = A_k/h^k_{n+1}$. Comparing consecutive estimates, $W_k$ allows a decision about when a different order method will be more efficient. The option \texttt{OrderSafetyFactors} specifies safety factors $\{f_1, f_2\}$ to be included in the comparison of estimates $W_k$.

- An order decrease is made when $W_{k-1} < f_1 W_k$.
- An order increase is made when $W_{k+1} < f_2 W_k$.

There are some additional restrictions, such as when the maximal order increase per step is one (two for symmetric methods), and when an increase in order is prevented immediately after a rejected step.

## 5 Examples

A number of numerical examples are now given in order to illustrate various aspects of the implementation. All computations were carried out using \textit{Mathematica} version 6.0.2 running on a Pentium 4 PC at 3.2 Ghz with 2GB RAM and the Fedora Core 7 Linux release. The following code loads packages that contain some predefined problems and some utility functions for plotting step size sequences.

\begin{verbatim}
In[1] := Needs["DifferentialEquations\NDSolveProblems"];
In[2] := Needs["DifferentialEquations\NDSolveUtilities"];
\end{verbatim}

### 5.1 Increment formulation comparison

To compare different formulations of extrapolation schemes consider an example from [19]:

$$y'(t) = (-y(t) \sin(t) + 2 \tan(t)) y(t), \quad y(\pi/6) = 2/\sqrt{3}.$$  

The exact solution is given $y(t) = 1/\cos(t)$. The following example uses extrapolation of the explicit Euler method with the Harmonic sequence and results are given for a single step of size $H = 1/10$.

The figure displays the work (function evaluations) vs error on a logarithmic scale for values of $T_{i,j}$, $i = 1, \ldots, 8$, $j = 1, \ldots, i$. The comparison is between:

- the standard formulation (3)
- the increment formulation (6) followed by standard extrapolation (2)
- the increment formulation (6) with extrapolation carried out on the increments (19)

The results for $T_{1,1}$ are indistinguishable graphically. Approximately one digit is gained in $T_{8,8}$ using the increment formulation for the base integrator. Furthermore, an additional decimal digit of accuracy is gained in computing $T_{8,8}$ by using the increment-based formulation in the extrapolation process. Moreover the gains are achieved using only minimal additional storage and some elementary arithmetic operations, so that the cost in terms of function evaluations is identical.

\copyright 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)
5.2 Method comparison

This example compares the explicit extrapolation methods with the explicit Runge-Kutta methods in NDSolve. The example used is the Arenstorf system, which is a restricted three-body problem [21].

    system = GetNDSolveProblem["Arenstorf"];

The reference solution is computed using the Extrapolation method with arbitrary precision arithmetic corresponding to approximately 48 decimal digits. The local relative and absolute error goals are set to half of the working precision. This gives a solution at the end of the integration which is correct to all digits when converted to a double precision representation.

    Timing[ sol = NDSolve[system, Method -> "Extrapolation", AccuracyGoal -> 24, PrecisionGoal -> 24, WorkingPrecision -> 48];
    refsol = First[FinalSolutions[system, sol]]; ]

This sets up a list of methods and options to use in the comparison. The Extrapolation method uses ExplicitModifiedMidpoint as the base integration scheme by default.

    methods = {"ExplicitRungeKutta", "StiffnessTest" -> False},
           {"ExplicitRungeKutta", "StiffnessTest" -> True},
           {"Extrapolation", "StiffnessTest" -> False},
           {"Extrapolation", "StiffnessTest" -> True};

The accuracy of the solution using each method will be computed along with the work to compute the solution at a variety of local error tolerances. The PrecisionGoal and AccuracyGoal options are used to specify the relative or absolute local error $10^{-k}, k = 4, 4.5, 5, \ldots, 14$. The details of determining the cost are handled by the NDSolveUtilities.m package function CompareMethods which uses the EvaluationMonitor feature to count function evaluations during the numerical integration.

    agpg = "Table[i, {i, 4, 14, 1/2}]";

    Timing[data = Table[CompareMethods[system, refsol, methods, AccuracyGoal -> i, PrecisionGoal -> i, {i, agpg}]; ]

The time to compute the reference solution can be seen to be only a small fraction of the time needed to compute the solutions with various methods at the given local tolerances. The following figure illustrates the results comparing error to work on a log-log scale.
The explicit Runge-Kutta method selects the (fixed) order according to the problem, local tolerances and initial step size estimate. When no stiffness detection is selected (ERK) then a standard I-step controller is used and this is the most efficient method at low tolerances. When stiffness detection is enabled (ERK-S) then a PI-step controller is used [14, 15], which generally results in slightly smaller step sizes being taken. Stiffness detection for these methods only costs some additional storage and elementary arithmetic operations.

Enabling stiffness detection (Extrapolation-S) as in (21) adds two function evaluations per step to the cost of the basic extrapolation method (Extrapolation). The extrapolation method without stiffness detection starts to become more efficient than the explicit Runge-Kutta method when the global error is around $10^{-8}$. This is in excellent agreement with the results for the FORTRAN codes dop853.f and odex.f in [21]. Furthermore, extrapolation using the increment formulation continues to give excellent results even for very stringent tolerances.

5.3 Brusselator PDE and stiffness detection

In NDSolve partial differential equations are currently discretized in space using the method of lines and methods for ordinary differential equations are then applied. Stiffness can arise in the translation of diffusion terms into a large system of ODEs. The Brusselator PDE in one spatial variable $x$ is given by:

$$\begin{align*}
\frac{\partial u}{\partial t} &= A + u^2 v - (B + 1) u + \alpha \frac{\partial^2 u}{\partial x^2}, \\
\frac{\partial v}{\partial t} &= Bu - u^2 v + \alpha \frac{\partial^2 v}{\partial x^2},
\end{align*}$$

with $0 \leq x \leq 1$, $A = 1$, $B = 3$, $\alpha = 1/50$ and boundary conditions:

$$
\begin{align*}
u(0, t) &= u(1, t) = 1, & v(0, t) = v(1, t) = 3, & u(x, 0) = 1 + \sin(2 \pi x), & v(x, 0) = 3.
\end{align*}
$$

This selects the Brusselator PDE problem with the time interval $[0, 10]$.

In[8] := system = GetNDSolveProblem["BrusselatorPDE"]; Attempting to solve the system with the default explicit extrapolation base method gives a message when stiffness is detected and the integration terminates. The default is to use the ExplicitModifiedMidpoint method with the Harmonic step sequence. The MaxStepFraction option is used to prevent the step size being restricted to a fraction of the relatively short integration range.

In[9] := NDSolve[system, Method -> "Extrapolation", MaxStepFraction -> 1];

© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)
NDSolve::nstf: At T == 1.545503525617818, system appears to be stiff. Methods Automatic, BDF or StiffnessSwitching may be more appropriate.

The StiffnessSwitching method is a controller method which acts as an interface to two methods, which can themselves be controller methods like Extrapolation. The first is appropriate for solving nonstiff problems and should be equipped with a stiffness detection device to raise an exception. The second method should be appropriate for solving stiff problems. Both of these assumptions are checked at run-time when the methods are initialized using method properties. Instead of issuing a message the StiffnessSwitching method traps the exception that is issued and it switches from extrapolation based on ExplicitModifiedMidpoint to extrapolation based on LinearlyImplicitEuler. The step sequence for LinearlyImplicitEuler defaults to the sub-Harmonic sequence.

In[10] := ssol = NDSolve[system, Method -> "StiffnessSwitching", MaxStepFraction -> 1];

Notice that the Jacobian matrix is computed automatically (user specifiable via either numerical differences or symbolic derivatives) and appropriate linear algebra routines are selected and invoked at run time.

A plot of the step sizes taken against time shows the large steps that are taken after the switch to a stiff solver around \( T = 1.5 \).


Out[11] =

6 Summary

We have described design considerations relevant for the incorporation of extrapolation methods into the differential equation solving environment in Mathematica. Providing a number of methods in a homogeneous fashion enables comparison on an equal footing. The plug-in mechanism of the new solver framework allows straightforward selection of alternative coefficient choices and the addition of user-defined special purpose methods. For details focusing on composition and splitting methods see [40]. A number of new methods have been derived and incorporated into the framework [37, 38]. For efficiency and simplicity, one step of Richardson’s extrapolation can be carried out using the separate method DoubleStep. This provides a simple, though not optimal, way of equipping any one-step integration method with a local error estimate. Useful functionality in the context of extrapolation, which has not yet been implemented, includes dense output [18] and extension to solving Differential Algebraic Equations [8, 17, 25].

References


© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)
SIAM (1999).


[27] O. Møller, Note on Quasi Double-Precision, BIT 5 (1965) 251–255.


© 2008 European Society of Computational Methods in Sciences and Engineering (ESCMSE)


