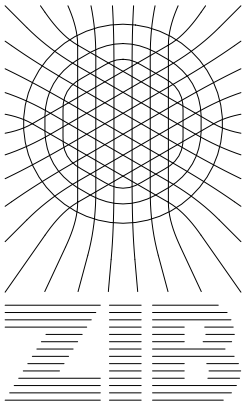
Konrad-Zuse-Zentrum für Informationstechnik Berlin



Andreas Hohmann

An Implementation of Extrapolation Codes in C++

TR 93-8 (November 1993)

An Implementation of Extrapolation Codes in C++

Andreas Hohmann*

Abstract

This report describes the new object oriented implementation of extrapolation codes (Eulex, Eulsim, Difex) for ordinary differential equations. The resulting C++ class library provides a simple and flexible interface to these methods and incorporates advanced features like continuous output and order-stepsize freezing. The interface of the ODE classes allows in particular a user-defined solver for the linear systems occuring in the linearly implicit discretization scheme. The library also provides some classes for numerical objects such as vectors and (full) matrices. Due to the underlying data-view concept it is possible to access substructures without copying. In addition, we included several utility classes such as a timer and a minimal command language that may be useful in other contexts, too.

Contents

*Konrad–Zuse–Zentrum Berlin, Heilbronner Str. 10, D-10711 Berlin-Wilmersdorf, Germany. hohmann@sc.zib-berlin.de

Introduction

This paper presents an update of the report 'Modular Design of Extrapolation Codes' [HW92] on the modular implementation of extrapolation codes for ordinary differential equations in C. Based on these codes, we have developed a C++ class library for ODEs and the associated solvers, which not only simplifies the usage of the whole package by its object oriented interfaces, but also incorporates new features such as continuous output [HO90], freezing of order-stepsize sequences, and other protocol facilities. Regarding the numerical ingredients, we simplified the stepsize restrictions for linearly implicit discretizations. The tests used in the former codes were replaced by a single residual oriented monotonicity test that increases the robustness of the codes for weak accuracy requirements.

The report is organized as follows. In the first part we recall the order and stepsize control for evolutional h-p methods (cf. Deuflhard [Deu83]) which is then applied to the standard extrapolation discretizations. Moreover, we describe in Section 3 a standard scaling strategy. The main purpose of this part is to collect the numerical methods and to introduce the notation used in the codes. If the reader is already familiar with these techniques or only wants to use the integrators without knowing the algorithmical details, he/she may skip this part.

The second part is devoted to the C++ class libraries. Starting with some conventions which we employed throughout the whole package, we describe the basic library for vectors and matrices of integers and real numbers. It provides the standard functionality like element access, arithmetic operators and linear solvers for full matrices. The following section concentrates on the ODE and integrator classes. Most classes are presented with some illustrative examples which are also included in the package. For a better understanding it might be helpful to study the header files while reading this part.

All classes are available via anonymous ftp from

elib.ZIB-Berlin.de (130.73.108.11)

in the directory pub/ode++. Here, the reader may also find more information about some implementational details and the latest version of this report.

I. Numerical Preliminaries

1 Order and Stepsize Control for Evolutional h-p Methods

H-p methods, which may be characterized as methods that make use of locally varying orders and stepsizes, occur in two rather different situations. On one hand, variable orders and stepsizes are exploited to construct global discretizations for *boundary value problems* via local refinement (or regridding) and order adjustment (see e.g. [GB86] for finite elements, [Hoh93] for collocation). On the other hand, h-p methods are used to solve *initial value problems* by successively choosing appropriate local orders and stepsizes for the next step of the integration process (see e.g. [HNW87], [Deu83]). We shall deal with the latter which may be called *evolutional h-p methods* to distinguish them from the former.

Let us consider an initial value problem

$$x' = f(x,t), \quad x(t_0) = x_0$$

of a first order ordinary differential equation. By $\Phi^{t,s}$ we denote the associated flow from s to t which we assume to exist for all $s,t \in \mathbb{R}$ in question. Our task is to compute

$$x_1 = \Phi^{t_1,t_0}(x_0)$$

up to a prescribed accuracy *tol*. We shall present the order and stepsize control according to Deuflhard [Deu83] in an abstract setting independent of extrapolation methods. To this end, we consider a family

$$\{(T_k, E_k) \mid \ k_{\min} \le k \le k_{\max}\}$$

of *discretization schemes* $T_k^{t+H,t}$ approximating the flow $\Phi^{t+H,t}$ and corresponding *error estimators* $E_k^{t+H,t}$ approximating the error $T_k^{t+H,t} - \Phi^{t+H,t}$. We call (T_k, E_k) an *h-p scheme of basic order* $p \in \mathbb{N}$, if T_k and E_k are of order pk and $pk+1$, respectively. More precisely, we assume that the error of T_k behaves according to

$$\varepsilon_k^{t+H,t}(x) := \|\Phi^{t+H,t}(x) - T_k^{t+H,t}(x)\| \doteq \gamma(x,t) H^{pk+1} \tag{1.1}$$

for some proportionality factor $\gamma(x,t)$ depending on x and t only, and the error estimator E_k satisfies

$$E_k^{t+H,t}(x) = \Phi^{t+H,t}(x) - T_k^{t+H,t}(x) + O(H^{pk+2}) \ . \tag{1.2}$$

In what follows, we will call k the *order* of the approximation, although this is not true but for the basic order $p = 1$. For ease of notation, we shall often drop the arguments t and x in the local situation and use the stepsize H as the main argument. So, we write for example $T_k(H)$ instead of $T_k^{t+H,t}(x)$.

In the extrapolation context the approximations T_k are the subdiagonal entries $T_k = T_{k+1,k}$ of the extrapolation tableau $\{T_{ik}\}$ corresponding to the subproblem $\Phi^{t+H,t}(x)$ with the outer stepsize H (see [Deu83] and Section 2 below). The error estimator is the subdiagonal error criterion, i.e., we compare the subdiagonal entries with the diagonal ones, which are of higher order and get $E_k = T_{k+1,k+1} - T_{k+1,k}$.

For the final approximation for a particular k, we may use the error estimator to refine the approximation T_k and take $T_k + E_k$ as the subproblem's solution. In an extrapolation method this is just the diagonal entry $T_{k+1,k+1}$.

1.1 Prediction of Optimal Orders and Stepsizes

It is the task of an adaptive order and stepsize control to divide the main problem $\Phi^{t_1,t_0}(x_0)$ into appropriate subproblems $\Phi^{t+H,t}(x)$ with (varying) stepsizes H and to choose the orders k in order to solve the main problem up to the prescribed local accuracy tol, i.e.

$$\varepsilon_k^{t+H,t}(x) \leq \text{tol} \ .$$

Substituting the norm of the error estimator E_k for the unavailable error ε_k, we require

$$\|E_k^{t+H,t}(x)\| \leq \text{tol} \ . \tag{1.3}$$

If this error criterion is fulfilled, we say that the method *converged* for the subproblem $\Phi^{t+H,t}(x)$ and call k the *convergence order*. Of course, this should be achieved with the least possible effort. To this end, we have to minimize the work per unit step in each step from t to $t+H$. Since we have to decide in advance which order and stepsize to take, we need a priori information about the stepsizes H_k required for the accuracy tol employing the discretization

T_k. Using the formula (1.1) for the approximation error of the discretizations T_k and the error ε_k, we know a posteriori which stepsize H_k would have been optimal in the sense that $\varepsilon_k(H_k) = \text{tol}$:

$$\varepsilon_k(H_k) = \text{tol} \quad \Longrightarrow \quad H_k = \sqrt[p_{k+1}]{\frac{\text{tol}}{\varepsilon_k(H)}}\, H$$

Since in reality we only have the error estimates E_k at hand instead of the true error ε_k, we multiply the accuracy tol with a safety factor $\rho := 1/4$ and define

$$H_k := \sqrt[p_{k+1}]{\frac{\rho\,\text{tol}}{\|E_k(H)\|}}\, H \tag{1.4}$$

as the expected *optimal stepsize* for T_k. Assuming that the proportionality coefficient $\gamma(x,t)$ does not vary much, it is feasible to use the *a posteriori* stepsize estimate H_k also as an *a priori* estimate for the next step.

If A_k measures the amount of work to compute the approximation T_k together with its error estimator E_k, we now have to minimize the work per unit step

$$W_k = A_k/H_k\ , \quad A_k = \text{amount of work to compute } T_k \text{ and } E_k.$$

Hence, the predicted optimal order k_{opt} for the next step has to satisfy

$$W_{k_{\text{opt}}} = \min_k W_k$$

and we obtain the corresponding predicted optimal stepsize as $H_{\text{opt}} := H_{k_{\text{opt}}}$.

1.2 A Priori Error Estimates

While successively computing the approximation $T_1, T_2, \ldots$ and error estimates $E_1, E_2, \ldots$, we would like to control whether the method behaves as expected or not. Therefore, we need a *convergence monitor* for the family of approximations, or, more precisely, a model of comparison α_k for the error ε_k. By means of this model we may check our approximation by

$$\|E_k(H)\| \ \leq \ \alpha_k\ .$$

Deuflhard proposed in [Deu83] a simple model based on Shannon's information theory, which works in a rather general context (not only extrapolation

methods). We regard the approximations T_k as encoding machines transferring function values into an approximation of the solution of the initial value problem. The input entropy $E_k^{(\mathrm{in})}$ is supposed to be proportional to the number B_k of function evaluations needed for T_k,

$$E_k^{(\mathrm{in})} = \alpha B_k, \quad B_k = \text{number of } f \text{ evaluations for } T_k,$$

while the output entropy $E_k^{(\mathrm{out})}$ is the number of significant binary digits

$$E_k^{(\mathrm{out})} = -\log_2 \varepsilon_k(H)$$

of the approximation T_k. To obtain estimates α_k for the errors ε_k, we assume that there is a linear relationship

$$E_k^{(\mathrm{out})} = \beta E_k^{(\mathrm{in})}$$

for some proportionality factor β, or, equivalently,

$$-\log_2 \varepsilon_k(H) = cB_k \tag{1.5}$$

for some constant c. Now, assume that we know a single error ε_q for some q. Then we may use (1.5) to define estimates $\alpha_k^{(q)}$ for all errors ε_k by

$$\alpha_q^{(q)} = \varepsilon_q \quad \text{and} \quad -\log_2 \alpha_k^{(q)} = cB_k$$

for some constant c. Eliminating c, we obtain

$$\alpha_k^{(q)} := \alpha_k^{(q)}(\varepsilon_q) := \varepsilon_q^{B_k/B_q} \ .$$

These a priori estimates $\alpha_k^{(q)}$ constitute the base of the convergence monitor.

1.3 Convergence Monitor

Suppose that we have predicted an optimal order k_{opt} and the corresponding stepsize H_{opt}, forecasting

$$\varepsilon_{k_{\mathrm{opt}}}(H_{\mathrm{opt}}) = \rho\,\mathrm{tol} \ . \tag{1.6}$$

Using the information theoretic model, we obtain from (1.6) the (a priori) error estimates

$$\varepsilon_k(H_{\mathrm{opt}}) = \alpha_k^{(k_{\mathrm{opt}})}(\rho\,\mathrm{tol}) \tag{1.7}$$

for arbitrary k. As already mentioned above, we will use these a priori estimates to control the convergence behaviour. We require the error estimate E_k to be at least smaller than the estimates $\alpha_k^{(k_{\text{opt}}+1)}(\rho\,\text{tol})$ that we would expect for convergence order $k_{\text{opt}}+1$, i.e.,

$$\|E_k(H_{\text{opt}})\| \;\leq\; \alpha_k^{(k_{\text{opt}}+1)}(\rho\,\text{tol})\;. \tag{1.8}$$

For an easy formulation (and realization) of the stepsize mechanism, we do not use the error estimates and stepsizes directly, but the corresponding stepsize factors. We define the stepsize factor $\beta_k(\varepsilon)$ for a given error ε as

$$\beta_k(\varepsilon) := \sqrt[pk+1]{\frac{\rho\,\text{tol}}{\varepsilon}}\;.$$

If we apply β_k to the a posteriori error estimates $\varepsilon = \|E_k(H)\|$ and the a priori estimates $\varepsilon = \alpha_k^{(q)}(\rho\,\text{tol})$, we obtain the stepsize factors

$$\lambda(k,H) := \beta_k(\|E_k(H)\|) \;\text{ and }\; \alpha(k,q) := \beta_k(\alpha_k^{(q)}(\rho\,\text{tol}))\;.$$

Using this notation, the predicted optimal stepsize (1.4) for order k reads

$$H_k = \lambda(k,H)\;H\;. \tag{1.9}$$

Moreover, the convergence monitor (1.8) is equivalent to

$$\lambda(k,H) \geq \alpha(k,k_{\text{opt}}+1)\;. \tag{1.10}$$

If (1.10) is violated for order k, we have to reduce the stepsize by a factor $\lambda_{\text{red}} < 1$ using the latest available information. According to (1.7) we want the new stepsize $H_{\text{red}} = \lambda_{\text{red}}\,H$ to meet the condition

$$\|E_k(H_{\text{red}})\| = \alpha_k^{(k_{\text{opt}})}(\rho\,\text{tol})\;,$$

or equivalently

$$\lambda(k,H_{\text{red}}) \;=\; \alpha(k,k_{\text{opt}}) \tag{1.11}$$

in order to achieve convergence for order k_{opt}. Since for any scalar $c \neq 0$

$$\lambda(k,cH) = c^{-1}\lambda(k,H)\;,$$

we derive from (1.11) the reduction factor

$$\lambda_{\text{red}} = \alpha(k,k_{\text{opt}})\,/\,\lambda(k,H_{\text{opt}})\;.$$

1.4 Possible Order Increase

So far we choose the optimal order k_{opt} by minimizing the work per unit step in the range from $k_{\min}$ to the convergence order k_{conv} of the last step. To check whether a higher order $k > k_{\text{conv}}$ would be cheaper, we employ again the a priori estimates as derived from the information theoretic model: We are looking for the stepsize H_k such that convergence is achieved for order k, that is, $\varepsilon_k(H_k) = \rho\,\text{tol}$. Therefore we expect

$$\|E_{k_{\text{conv}}}(H_k)\| = \alpha^{(k)}_{k_{\text{conv}}}(\rho\,\text{tol}) \ ,$$

or, equivalently,

$$\lambda(k_{\text{conv}}, H_k) = \alpha(k_{\text{conv}}, k) \ ,$$

Since for the optimal stepsize $H_{k_{\text{conv}}}$ for order k_{conv} we have $\lambda(k_{conv}, H_{k_{\text{conv}}}) = 1$, we obtain for the stepsize quotient $H_{k_{\text{conv}}}/H_k$

$$\frac{H_{k_{\text{conv}}}}{H_k} = \frac{H_{k_{\text{conv}}}}{H_k}\,\lambda(k_{\text{conv}}, H_{k_{\text{conv}}}) = \lambda(k_{\text{conv}}, H_k) = \alpha(k_{\text{conv}}, k) \ .$$

Thus, regarding the work per unit step, we may test an order $k > k_{\text{conv}}$ by

$$W_k < W_{k_{\text{conv}}} \Longleftrightarrow A_k\ \alpha(k_{\text{conv}}, k) < A_{k_{\text{conv}}} \ .$$

1.5 Order Window

Both, the convergence monitor as well as the possible order increase should not be applied to all orders k. First, the information theoretic model is only valid for an optimal code, i.e., at most around the optimal order and stepsize. Second, it is a good numerical practice not to change the order too rapidly to get a "smooth" behaviour of the algorithm. Therefore, we introduce a so-called *order window*

$$\{k \in \mathbb{N} \mid \ \max(k_{\min}, k_{\text{opt}} - 1) \le k \le \min(k_{\max}, k_{\text{opt}} + 1)\}$$

around the predicted optimal order k_{opt}. The accuracy check (1.3) as well as the convergence monitor (1.10) are only applied inside this range. Moreover, we choose the next optimal order from this set.

If the accuracy check (1.3) failed for all orders k inside the order window, we reject the given stepsize H_{opt}, even if the convergence monitor was not violated. Again using the latest information, i.e., the error estimate $E_k(t, H_{\text{opt}})$, and the formula (1.9) for the optimal stepsize for order k_{opt}, we take

$$H_{\text{red}} = \lambda(k, H_{\text{opt}})\, H_{\text{opt}}$$

as the new stepsize in order to meet $\|E_{k_{\text{opt}}}(H_{\text{red}})\| = \rho\,\text{tol}$.

1.6 Determination of a Feasible Maximal Order

Based on the information theoretic model, we can determine a maximal order $k_{\text{fea}} \leq k_{\max}$ for which we expect computational profit. More precisely, we compute the smallest order k such that using the next order would be more expensive, that is, $W_{k+1} > W_k$, or, equivalently,

$$\frac{A_{k+1}}{A_k} > \frac{H_{k+1}}{H_k}.$$

Obviously, we have no stepsizes H_k at hand, but we can derive an estimate of the quotient H_{k+1}/H_k. If H_{k+1} is the stepsize to obtain

$$\varepsilon_{k+1}(t, H_{k+1}) = \rho\,\text{tol},$$

then, using the information theoretic model, we expect

$$\varepsilon_k(t, H_{k+1}) = \alpha_k^{(k+1)}(\rho\,\text{tol}).$$

The optimal stepsize H_k for convergence order k can be computed by

$$H_k = \beta_k(\varepsilon_k(t, H_{k+1}))H_{k+1} = \beta_k(\alpha_k^{(k+1)}(\rho\,\text{tol}))H_{k+1} = \alpha(k, k+1)H_{k+1}$$

Therefore we get an estimate for the stepsize quotient by

$$\frac{H_{k+1}}{H_k} = \alpha(k, k+1)^{-1}$$

Thus, we define the *feasible maximal order* k_{fea} as the smallest order $k_{\min} \leq k \leq k_{\max}$ satisfying

$$\frac{A_{k+1}}{A_k} > \alpha(k, k+1)^{-1}\,.$$

2 Application to Extrapolation Methods

In this section we shall apply the rather general order and stepsize control to extrapolation methods. The base of such an method is a discretization scheme depending on a stepsize h which allows an asymptotic expansion in h^p.

2.1 Discretization Schemes

To fix notation, we denote the result of n steps of such a *basic discretization scheme* by $D_n^{t+H,t}(x)$, where x and t are the initial value and time, and H the outer stepsize. In the following examples, $x_n = D_n^{t+H,t}(x)$ is recursively defined by a sequence $x_0, \ldots, x_n$. By h we denote the inner stepsize $h := H/n$.

EXAMPLE 1. *Explicit Euler discretization (code* Eulex*).*

a) $x_0 := x, \;\; t_0 := t$

b) $x_{i+1} := x_i + hf(x_i, t_i), \;\; t_{i+1} := t_i + h$ for $i = 0, \ldots, n-1$.

EXAMPLE 2. *Linearly implicit Euler discretization (code* Eulsim*).*

a) $x_0 := x, \;\; t_0 := t$

b) $x_{i+1} := x_i + h(I - hf_x(x,t))^{-1} x_i, \;\; t_{i+1} := t_i + h$ for $i = 0, \ldots, n-1$.

EXAMPLE 3. *Explicit mid-point rule (code* Difex*).* The explicit mid-point rule b) as a two step discretization is combined with an initial explicit Euler step a) and Gragg's final step c) to obtain the result $D_n^{t+H,t}(x) = x_n$.

a) Explicit Euler start step:

$$x_0 := x, \quad t_0 := t, \quad x_1 := x_0 + hf(x_0, t_0), \quad t_1 := t_0 + h,$$

b) Explicit mid-point rule:

$$x_{i+1} = x_{i-1} + 2hf(x_i, t_i), \quad t_{i+1} = t_i + h \quad \text{for} \quad i = 1, \ldots, n$$

c) Gragg's final step: $x_n = \frac{1}{4}(x_{n-1} + 2x_n + x_{n+1})$.

2.2 Order and Stepsize Control in the Extrapolation Context

As already mentioned in Section I the main application of the rather general order and stepsize control are extrapolation methods. We denote by $\{T_{ik}\}$ the extrapolation table corresponding to the discretization scheme $D_n^{t+H,t}(x)$. More precisely, let

$$0 < n_1 < n_2 < \cdots$$

be a subdivision sequence, $n_i \in \mathbb{N}$, and define $\{T_{ik}\}$ by the well-known formula for polynomial extrapolation with respect to h^p:

$$\begin{aligned} T_{i,1} &= D_{n_i}^{t+H,t}(x) \quad \text{for} \quad i = 1, \ldots \\ T_{i,k} &= T_{i,k-1} + \frac{T_{i,k-1} - T_{i-1,k-1}}{\left(\frac{n_i}{n_{i-k+1}}\right)^p - 1} \quad \text{for} \quad k = 2, \ldots, i \end{aligned}$$

Since the subdiagonal entries $T_{k+1,k}$ are of order pk, they constitute a family T_k of approximations as defined in section I. Moreover, the diagonal entries $T_{k+1,k+1}$ are of order $p(k+1)$ and may be used to define an error estimator, namely the *subdiagonal error criterion*. Thus, we have

$$T_k = T_{k+1,k} \quad \text{and} \quad E_k = T_{k+1,k+1} - T_{k+1,k} \; .$$

To use the order and stepsize control in this particular context, we have to compute the sequences $\{A_k\}$ and $\{B_k\}$ measuring the amount of work for T_k and E_k and the "information" employed for T_k, respectively. In the extrapolation framework, A_k is the amount of work to compute the extrapolation table up to row $k+1$. Therefore, neglecting the effort for the recursive computation of the T_{ik} for $k \geq 2$, we have to measure the cost for $T_{1,1}, \ldots, T_{k+1,1}$, i.e. for $D_{n_i}^{t+H,t}(x)$ for $i = 1, \ldots, k+1$.

EXAMPLE 1. *Explicit Euler discretization (code* EULEX*).* The subdivision sequence used in EULEX is the harmonic sequence given by $n_i = i$ for $i = 1, 2, \ldots$. The amount of work sequence $\{A_k\}$ is recursively defined by

$$A_0 := n_1 \quad \text{and} \quad A_i = A_{i-1} + n_i - 1 \quad \text{for } i = 1, 2, \ldots .$$

Since $f(x_0, t_0)$ is computed only once at the beginning of the extrapolation process, we must subtract 1 in the last formula.

EXAMPLE 2. *Linearly implicit Euler discretization (code* EULSIM*).* The subdivision sequence of EULSIM is again the harmonic sequence. For the linearly implicit Euler discretization we have to take into account the cost for an evaluation of the Jacobian and the solution of the arising linear equations. Therefore we have to introduce the following work coefficients:

C_f	cost of an f-evaluation
C_J	cost of an evaluation of the Jacobian $f_x(x_0)$
C_{LU}	cost of a decomposition of $I - hf_x(x_0)$
C_{subst}	cost of the forward/backward substitutions for the decomposed matrix

Using this information, we get the amount of work sequence

$$\begin{aligned} A_0 &= C_J + n_1(C_{\text{subst}} + C_f) \\ A_i &= A_{i-1} + C_{LU} + n_i C_{\text{subst}} + (n_i - 1)C_f \quad \text{for} \quad i = 1, 2, \ldots \end{aligned}$$

In the present implementation of EULSIM the default values are

$$C_f = 1, \quad C_J = nC_f \quad \text{and} \quad C_{LU} = C_{\text{subst}} = 0 \ ,$$

where n is the dimension of the problem.

EXAMPLE 3. *Explicit mid-point rule (code* DIFEX*).* The subdivision sequence used in DIFEX is the double harmonic sequence $n_i = 2i$, $i \geq 1$. The sequence A_k is given by

$$A_0 = n_1 + 1, \quad A_i = A_{i-1} + n_{i+1} \quad \text{for } i \geq 1.$$

Note that due to Gragg's final step one additional f-evaluation is needed for the computation of $T_{i,1}$ for $i \geq 1$.

Concerning the second sequence B_k, the approximation $T_k = T_{k+1,k}$ only contains information from $T_{2,1}, \ldots, T_{k+1,1}$, as easily derived from the extrapolation table. Thus, B_k measures the information necessary for $T_{2,1}, \ldots, T_{k+1,1}$, i.e., for the basic discretizations $D_{n_i}^{t+H,t}(x)$ for $i = 2, \ldots, k+1$. Let $\bar{A}_k$ denote the information needed to compute $T_{2,1}, \ldots, T_{k+1,1}$. Then, in the ODE context, the sequences $\bar{A}_k$ and B_k are related by

$$B_k = \bar{A}_k - \bar{A}_0 + 1 \ .$$

In EULEX and DIFEX the information $\bar{A}_k$ can be measured by the number of necessary f-evaluations. Thus, $\bar{A}_k$ equals A_k in these codes. In EULSIM the information contained in the terms $(I - hf_x(x_0))^{-1} f(x)$ may be counted. Therefore $\{\bar{A}_k\}$ and $\{A_k\}$ differ. Here, $\{\bar{A}_k\}$ is given by

$$\bar{A}_0 = n_1, \quad \bar{A}_i = \bar{A}_{i-1} + n_i - 1 \quad \text{for } i = 1, 2, \ldots .$$

2.3 Additional Order and Stepsize Restrictions

Proceeding, we describe the residual oriented monotonicity test that replaces the stepsize restrictions as described in [HW92] and [Deu89]. The implicit Euler discretization for an ordinary differential equation

$$x' = f(x, t)$$

and the stepsize h is given by the formula

$$x_{k+1} = x_k + hf(x_{k+1}, t_{k+1}) \approx x(t_{k+1}), \quad t_k = t_0 + kh \, .$$

Equivalently, we have to solve in each time step the nonlinear equation

$$F_k(x) := F_k(x, h) := x - x_k - hf(x, t_k + h) = 0 \, . \tag{2.1}$$

The linearly implicit Euler discretization is just the first step of an inexact Newton method for the nonlinear problem $F_k(x) = 0$. Equivalently, we solve the parameter dependent problem $F_k(x, h) = 0$ with h fixed, that is, using "fixed parametrization". The Jacobian

$$F_k'(x) = I - hf_x(x, t_k + h)$$

is replaced by

$$J := I - hA, \quad \text{where} \quad A := f_x(x_0, t_0) \, .$$

For autonomous systems this is in fact the exact Jacobian for the first step. As initial guess for the solution x_{k+1} of $F_k(x) = 0$, we take the last step x_k, finally leading to the linearly implicit Euler formula

$$x_{k+1} = x_k + \Delta_k, \quad \text{where} \quad \Delta_k := -J^{-1}F(x_k) = (I - hA)^{-1} f(x_k, t_k + h) \, .$$

The convergence of this Newton method may be checked using the the monotonicity test

$$\mu_k < \mu_{\max} := 1 \;, \tag{2.2}$$

where

$$\mu_k := \frac{\|F(x_{k+1})\|}{\|F(x_k)\|} = \frac{\|\Delta_k - hf(x_{k+1}, t_{k+1})\|}{\|hf(x_k, t_{k+1})\|}$$

is quotient of the residuals (μ for "monotonicity coefficient"). If condition (2.2) is violated, the stepsize should be reduced (cf. [Hoh93]) by

$$h_{\text{new}} = \rho \frac{\mu_{\max}}{\mu_k} \cdot h_{\text{old}} \;,$$

where $\rho < 1$ is some safety factor, say, $\rho = 0.5$. This simple monotonicity test is very cheap since it only requires an additional vector subtraction and the two norms. For non autonomous systems we have to compute an additional right hand side $f(x_{k+1}, t_{k+1})$. As demonstrated in [Hoh93], this stepsize reduction facility increases the robustness of the linearly implicit extrapolation codes drastically.

3 Norms and Scaling

In our description of the extrapolation method we used an abstract norm $\|\cdot\|$ to measure the error estimate E_k. The choice of a suitable norm plays an important role for the performance of the algorithm. First of all, we recommend a smooth norm, since the behaviour of the order and stepsize control depends on the given norm. The most common choice is the Euclidean norm $\|\cdot\|_2$. On the other hand, we require the algorithm to be *scaling invariant*, i.e., independent of the units chosen for the components x_i of the state variable x. To this end, we introduce a *scaled norm*

$$\|x\|_{scal}^2 := \frac{1}{n}\|D^{-1}x\|_2^2 = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i}{s_i}\right)^2 \;,$$

where $D = \operatorname{diag}(s_1, \ldots, s_n)$ is the *current scaling matrix*. Using a scaled norm in the accuracy check (1.3) also allows us to control the relative (local) error of the solution rather than the absolute one. The current scaling D may depend on all solutions computed so far. Thereby, we want the scaling factors s_i to meet the following requirements:

1) The algorithm should be scaling invariant.

2) The accuracy check $\|E_k(H)\| < \text{tol}$ should (in principle) control the relative error.

3) If a component becomes too small (regarding the modulus), the accuracy requirement should be softened to control the absolute error (*absolute lower bound*).

4) The scaling must not change abruptly. In particular, a zero component in a single step should not alter the scaling.

5) As in 3), a component which is relatively small with respect to the maximal value over all time steps computed so far should be controlled using its absolute error (*relative lower bound*).

These claims lead to a standard scaling strategy. We use the following information:

x	current solution vector
x_{last}	last accepted solution (last time step)
$x_{\max}$	maximum over all solutions accepted so far
s_{abs}	absolute lower bound for the scaling factors
s_{rel}	relative lower bound for the scaling factors

The scaling consists of two phases. At the beginning we have to *initialize* the scaling factors taking into account the initial value and the lower scaling bound given by the user. After each integration step we have to *rescale* the norm using the already accepted steps. All vector operations are to be understood componentwise.

- Initialization: $s := x_{\max} := x_{\text{last}} := \max\{|x_{\text{start}}|, s_{\text{abs}}\}$
- Rescaling:

nonstiff case:	$s := \max\{\|x\|, \|x_{\text{last}}\|, s_{\text{rel}}\, x_{\max}, s_{\text{abs}}\}$
stiff case:	$s := \max\{\|x\|, x_{\max}, s_{\text{abs}}\}$
both:	$x_{\max} := \max\{x_{\max}, \|x\|\}$ and $x_{\text{last}} := x$

II. Description of the Class Libraries

4 General Design and Style Conventions

The whole class library was realized in C++ on a SUN SPARC 10 workstation. As compilers we employed the the GNU gcc 2.5.4 compiler and the Sun C++ compiler Sparcworks 3.0 (based on AT&T's cfront 2.1). For a consistent style we use the following conventions.

4.1 Identifier

- Multi-word names capitalize each word but possibly the first one.
- Classes, types, member functions and procedures start with a capital letter.
- Variables start with a small letter.

4.2 Header Files

Each header file includes all headers the compiler needs to understand it. To prevent multiple inclusions, we employ the standard preprocessor strategy:

```
#ifndef _<filename>_h
#define _<filename>_h
...
#endif /* _<filename>_h */
```

To save parsing time and to avoid unnecessary dependencies, header files are only included if really needed. Sometimes, this strategy is accomplished by handles to private data, i.e., pointers to user unknown classes. As an example, the *Timer* class uses a handle *TimerData *data* to avoid the inclusion of *time.h* and *types.h* in the header file *Timer.h*.

4.3 Order of Function Arguments

The arguments of procedures are ordered according to the following rules:

- Input or input/output arguments are posed in front of output arguments.
- More complex arguments are posed in front of less complex ones, i.e., classes before reals before integers.

4.4 Usage of the const Qualifier

We use the *const* qualifier in the original C++ sense. This means that an argument or a member function is declared *const* whenever the corresponding class is not changed in a stronly bitwise sense. More precisely, the class variables have to remain unaltered, whereas 'pointed to objects' may be changed. As an example, the assignment operator (see section 5.6.2) for mathematical container classes such as *RealVec* is a *const* operator, since the structure (size, index range) of the object is not changed, although its elements may contain new values afterwards.

4.5 Type Definitions

The following standard types and constants are used:

```
typedef double Real;
typedef int Int;
typedef int Bool;
const Bool false = 0, true = 1;
extern Real epsMach, sqrtEpsMach, pi;
```

Here, *epsMach* is the relative machine precision for *Real* and *sqrtEpsMach* its square root. These constants are set automatically.

4.6 Message and Error Handling

We distinguish three types of messages:

- *Messages:* providing information about the *regular* behaviour of the program
- *Warnings:* providing information about some *irregular* behaviour of the program (recoverable errors)

- *Errors:* not recoverable errors leading to the termination of the program

A typical not recoverable error is a bound error in a matrix or vector class or a similar bad argument in a procedure. Since the use of such a bad data would lead to a logic error, the program is aborted. Corresponding to the three types of messages, we provide the functions *Message*, *Warning* and *Error*.

```
extern void Message(char *format ...);
extern void Warning(char *format ...);
extern void Error(char *format ...);
```

They are called with a *printf*-like format string and variable argument list. These functions print the type of the message followed by the message itself on the current error stream *cerr*.

5 The Matrix Vector Library

The last few years gave birth to an almost exponentially increasing number of C++ class libraries for the basic mathematical objects of vectors and matrices. We would like to mention Rogue Wave's rather complete and efficient LINPACK.H++ library and the new C++ front end LAPACK++ of the LAPACK numerical linear algebra library. Regarding these libraries it seems to be completely superfluous to develope one's own. Nonetheless, we use our own classes. The main reason for that is that we are still looking for a C++ class library that meets all our demands. It should

- have an easy and consistent interface
- allow a flexible access to substructures avoiding copying
- be highly efficient concerning storage and speed
- provide arbitrary lower index bounds
- be public domain
- offer support

For the first three points, LINPACK.H++ is definitely the standard which seems to be also met by LAPACK++ (being still under development). Both packages are based on the BLAS libraries and thus may exploit the architecture of the actual platforms. The Lapack++ library has the advantage that it is public domain and thus satisfies the most restrictive demand. The fourth requirement is more of aestetical nature. Using arbitrary index ranges (e.g., from -5 to 5) allows to implement a mathematical algorithm in its most natural notation. Since this feature is easily accomplished in C, we would give up this point only reluctantly.

In this situation our strategy was to define a C++ interface for vectors and matrices that may serve as a placeholder for any other library. Hence, the package presented here is definitely not meant as a substitute for the general purpose libraries mentioned above. Its main objective is to simplify and unify the usage of vectors and matrices in the context of the ODE package. Accordingly, we tried to restrict the functionality to the major functions provided by almost all linear algebra class libraries such as element access, arithmetic operators, assignment and solver classes. Fortunately, the design of the libraries seems to converge in the sense that most libraries use similar semantics and even the same storage schemes (see the data-view concept below).

Naturally, our decision for using our own classes was influenced by the fact that we only had to implement the C++ interface, since we already had the modular C library at hand which the extrapolation codes in C (cf. [HW92]) are based on.

5.1 The Base Class MyObject

Table I sketches the class hierarchy of the libraries. At the top of the class hierarchy we have defined a class *MyObject* which provides

- Standardized input and output via the virtual members *ReadFrom* and *PrintOn*. Overloading these functions automatically defines the stream operators >> and << in a suitable way. Thus, we avoid the corresponding operator friends in the derived classes.

- The message and error handling for classes. Calling the functions *Message*, *Warning*, or *Error* inside a member function, these functions

Range MathData					
MyObject	Command				
	MathView	VecView	IntVec RealVec ComplexVec		
		MatView	IntMat RealMat ComplexMat		
	MatrixSolver	LRSolver QRSolver			
	ListNodeBase ListIterBase ListBase Shell ExtObject	ListNode ListIter List MiniTcl			
		Ode	BrussOde ChemOszOde VarOde		
		Integrator	HpIntegrator	ExMethod	Eulsim Eulex
	IntegrationProtocol HpStep	HpProtocol EulsimStep			

TABLE I. Class hierarchy of the libraries

print the type of the message followed by the class name defined by the virtual function *Name* and the message itself.

- A control facility counting the instances of *MyObjects*. The static member function *MyObject::Count* should return zero at the end of a program (if there are no *MyObjects* left).

As an example regard the following piece of code:

```
class A : public MyObject {
public:
    char *Name() const { return "A"; }
    void F(Real x) {
        if (x<0) Warning("F: x=%f less than zero", x);
    }
}

main() {
    A a;
    a.F(-1);
}
```

Calling *main* will print the warning:

```
warning: A::F: x=-1 less than zero
```

5.2 The List Template

The class *List* is the only template class in the whole package. It supports the typical functions of lists like insertion, removal, and access, and it is implemented as a doubly linked list of pointers to objects of the given type (cf. Stroustrup [Str91]). Loops through a list are implemented via the usual iterator class. The following example contains some of the features.

```
List<RealVec> list;
RealVec x, y, *p;

list.AppendHead(&x);
list.AppendTail(&y);
list.InsertBefore(&x, &y);
```

```
list.InsertAfter(&x, &y);

ListIter<RealVec> iter(list);
for (iter.First(); p=iter(); iter++) cout << *p << ' ';
for (iter.First(); iter; iter++) cout << *iter() << ' ';

cout << "list: " << list << '\n';
cout << "remove " << *list.Remove(&a) << endl;
```

Here, we consider a list of vectors of real numbers. The first list commands show how to put new objects at the head or tail of the list and before or after a given object. The next lines demonstrate two different ways to use the iterator class. Finally, we show the output procedure (based on the *PrintOn* function) and the removal of a single element from the list. The function *Remove* returns the pointer to the removed object so that it can be deleted by the user afterwards. To remove all elements, call the method *EmptyList*. The function *DeleteAll* not only removes all elements from the list but also deletes the pointed to objects.

5.3 The Minimal Command Language

The package contains a minimal object oriented interface to a command language tool such as the well-known *tcl* (tool command language) by Ousterhout. We only make use of user-defined commands and variables of integer, double or string type. This minimal subset of functions of a command language is encapsuled in the abstract class *Shell*.

```
typedef int (*CommandProc)(void *data, int argc, char **argv);

class Shell : public MyObject {
public:
    virtual void MainLoop() = 0;
    virtual void Prompt(char*) {}
    virtual void CreateCommand(
        char* name,             // name of the command
        void* clientData,       // pointer to user data
        CommandProc proc        // pointer to user procedure
    ) = 0;
    virtual Real GetReal(char *name, Real defaultValue) = 0;
```

```
    virtual Int GetInt(char *name, Int defaultValue) = 0;
    virtual char *GetString(char *name, char *defaultValue) = 0;
    virtual void Result(char *) {}

    char *Name() const { return "Shell"; }
};
```

A user defined command procedure has to be of type *CommandProc*. A new command for a shell is defined using the method *CreateCommand*. The shell should at least know the command *exit* to leave the command shell and the commands *set* and *unset* to define and undefine variables. The abstract virtual methods *Get<type>* should give the value of the corresponding variable, if it is set and of the corresponding type. Otherwise, the default value should be returned. The function *MainLoop* starts the command shell and thus gives the control to the command language.

An implementation of this minimal command language is given by the class *MiniTcl*. Due to the intensive use of the list structure, it comprises only a few pages of source code. The following example demonstrates most of the features.

```
static int CmdPrint(void *data, int argc, char **argv) {
    Shell *shell = (Shell *) data;
    Bool done;

    if (done = argc==2) {
     if (strcmp(argv[1], "tol") == 0) {
    cout << "tol = " << shell->GetReal("tol", 1e-4);
     }
     else {
         done = false;
         shell->Result("error: unknown parameter");
     }
    }
    else shell->Result("usage: print <parameter>");
    return done;
}

main() {
    Shell *shell = new MiniTcl();
```

```
    shell->CreateCommand("print", shell, CmdPrint);
    shell->Prompt("main>");
    shell->MainLoop();
    delete shell;
}
```

We define a command procedure *print* that may be used to print all the variables which correspond to parameters of a program. Here, we only consider the single real accuracy parameter *tol*. The command procedures are called in the standard C format with the number of arguments followed by the array of argument strings. If the argument of *print* is different from *tol*, we print an error message. Otherwise, we print the value of the variable *tol*. Since we used the pointer to the shell as client data, we know the calling shell and are able to look for the variable *tol*. If it is not defined or if it is no real number, the default value 10^{-4} is printed.

Observe that there may be arbitrary many command shells, each with its own list of commands and variables. This may be used to implement several command levels, where a main shell gives control to a subshell, etc.

5.4 Matrix and Vector Classes

So far, the package only contains classes for full matrices and vectors of *Real*s and integers as well as classes for LR and QR decomposition.

REMARK 1. The class library will be extended by complex matrices and vectors (which already exist but are not yet contained in the public library), symmetric and antisymmetric matrices and the computation of eigenvalues.

The matrix and vector classes are supplied with the standard arithmetic operators (=, +=, -=, *=, +, -, *). Moreover, we defined for vectors and matrices of *Real* the (scaled) Euclidian product and norm, and the (componentwise) functions *abs, min* and *max*.

For the algebraic solvers, we employ the standard interfaces as used by most object oriented linear algebra packages. For each solver, such as LR and QR decomposition, there is an associated class derived from a base class *Solver*. A preparatory method called *Decompose* should contain the decomposition step. If this step has been successful, calling

```
Solve(const RealVec& b, const RealVec& x)
```

computes the solution of the linear system $Ax = b$. Here, x has to be well sized. The following example shows a typical usage of the LR solver class.

```
Int n=3;
RealMat A(n);
RealVec b(n), x(n);
LRSolver solver(A);
```

Set A and b.

```
if (!solver.Decompose()) Warning("LR decomposition failed");
else {
    solver.Solve(b, x);
    cout << "solution x = " << x << endl;
}
```

The solution may be equivalently obtained from *Solution(const RealVec& b)* as is the following example.

```
Int n=3;
RealMat A(n);
RealVec b(n);
LRSolver solver(A);
```

Set A and b.

```
if (!solver.Decompose()) Warning("LR decomposition failed");
else {
    cout << "solution x = " << solver.Solution() << endl;
}
```

5.5 Index Ranges

While many packages use a fixed lower index (in most cases 0 as in C), we prefer arbitrary index ranges of contiguous integers corresponding to the mathematical formulation. As an example,

```
RealVec x(-5, 5);
```

defines a vector of Reals having 11 elements indexed from -5 to 5. Similarly,

```
RealMat A(0, m, 1, n);
```

defines a matrix of Reals with $m + 1$ rows (form 0 to m) and n columns, where m and n are some positive integers. The default lower index is always 1 (as in the mathematical literature) so that

```
IntVec v(10);
```

is synonymous to `IntVec v(1, 10)` and similarly

```
IntMat A(m, n), B(n);
```

means `IntMat A(1, m, 1, n), B(1, n, 1, n)`.

5.6 The Data-View Concept

A widely accepted concept for the design of linear algebra classes is the distinction between the *data* and the different ways to *access* these data in terms of mathematical objects such as vectors and matrices. Following Rogue Wave's LINPACK.H++, we call this principle the *data-view concept* (which coincides with the concept of 'virtual vectors and matrices' as developed by M. Wulkow and the author). The *data* are thought of as contiguous block of memory, a *view* as a pointer to some block of data and a description of the view's structure (size, index ranges, etc.). Since the same block of data may be accessed by different views, copying is reduced to a minimum. Moreover, the allocation and deallocation of memory, which produces at least 50% of all C errors, is confined to the single class defining the block of data. The memory is freed properly by simply counting the views on the data (via constructor/destructor) and deallocating the block if no view is left.

5.6.1 Realization

The implementation of the data-view concept is based on two classes, *MathData* and *MathView*. The former organizes the allocation and deallocation of contiguous blocks of data, whereas the latter is the base class for all kinds of views on these data. Although the class *MathData* is hidden to the user, it seems to be useful to regard its very simple definition.

```
class MathData {
    friend class MathView;
private:
    MathData(Int n, size_t size);
    ~MathData();

    char *s;
    Int refs;
};

MathData::MathData(Int n, size_t size) : refs(1) {
    s = new char [n*size];
}

MathData::~MathData() {
    delete s;
}
```

By definition, *MathData* may be used by its friend *MathView* only.

```
class MathView : public MyObject {
public:
    MathView();
    MathView(Int n, size_t size);
    MathView(const MathView& x);
    virtual ~MathView();
protected:
    void Reference(const MathView&);
    void *Data();

    MathData *data;
};
```

```
MathView::MathView() : data(0) {}

MathView::MathView(Int n, size_t size) {
    if (n<1) Error("negative length %d", n);
    data = new MathData(n, size);
}

MathView::MathView(const MathView& x) {
    data = x.data;
    if (data) data->refs++;
}

MathView::~MathView() {
    if (data) if (--data->refs == 0) delete data;
}
```

Its main constructor *MathView(Int, size_t)* initializes the *MathData* pointer *data* with an appropriate block of data, thereby setting its references counter to 1. The copy constructor *MathView(const MathView&)* constructs a new view to the data of the given *MathView* argument. Accordingly, only the *MathData* pointer is copied and its reference counter incremented by 1. To free the data block properly, the *MathView* destructor decrements the reference counter of its data and deletes the data only if no view to these data is left.

5.6.2 Assignment and Copy Constructor

A crucial point for a consistent and efficient structure of the mathematical matrix-vector classes are the semantics of the copy constructor and the assignment operator. For an easy migration to the (commercial) LINPACK.H++ package, we follow the semantics of this matrix-vector library.

- *Assignment:* The assignment operator of a mathematical container class (derived from *MathView*) is realized as elementwise copy, i.e., the values of the elements of the right hand side are copied to the elements of the left hand side. Both objects have to be of the same structure (size, index range). Otherwise, an error occurs.

- *Copy constructor:* A copy constructor creates a new view on the data of the given *MathView* argument. Thus, the new object refers to the same block of data.

Since the assignment operator does not change the structure of the left hand side, the view remains unaltered in the bitwise sense. Hence, assignment is a *const* operator.

5.6.3 Access to Substructures

Let us consider the following example for the copy constructor. The lines

```
RealVec x(5);
RealVec y = x;
```

first construct a vector x of five real components and then let y be a vector pointing to the same data. At first sight it appears rather dangerous to permit two objects to point to the same data. And, obviously, we would hardly use two different names for an identical view on the same data as in the example above. The definition of the copy constructor allows us, however, a very flexible access to substructures of mathematical objects without copying. As a more useful example, regard the following sequence:

```
RealVec x(5);
RealVec y(x, Range(1, 3)), z(x, Range(4, 5));
```

We create three different views on the same data. The helper class *Range* describes a range of integers, here from 1 to 3 and from 4 to 5, respectively. The constructor

```
RealVec::RealVec(const RealVec& x, const Range&, Int l=1);
```

creates a view on the the given range of the data of the argument vector x with l as the new lower index bound (default 1). Consequently, y is a vector pointing to the first three elements of x and z to the last two components of x. Mathematically speaking, the data are viewed either as the single vector $x \in \mathbb{R}^5$ or the pair $(y, z) \in \mathbb{R}^3 \times \mathbb{R}^2$. The assignment

```
z(1) = 5;
```

for example, sets the fourth component of x to 5.

Obviously, this is only the simplest example of a substructure. The package allows vector type substructures of arbitrary *contiguous* blocks of data. Since we store matrices columnwise, we may for instance view a column of a matrix as a vector:

```
RealMat A(4, 5);
RealVec x = A.Column(2);
```

Here, x is a view on the second column of the matrix A.

REMARK 2. In contrast to the LINPACK.H++ library, vectors have to point to *contiguous* blocks of data. Otherwise, we would have to use variable increments to access vector components. For small to medium sized vectors this is a major drawback, since the direct access *v[i]* for a pointer *Real *v* is 2-7 times faster than the access *v[i*step]* with a variable integer increment *step* (measured on a SPARC workstation and a 486 PC, the factor depending on the compiler and the optimization). As a consequence, we can only view either rows or columns of matrices. Since we prefer the latter, we adopted the FORTRAN-like columnwise storage scheme.

In addition, it is possible to create views on submatrices of a given matrix. In the following example, B is a view on the submatrix consisting of the first two rows of the last two columns of A.

```
RealMat A(4, 5);
RealMat B(A, Range(1, 2), Range(4, 5));
```

We also implemented the *operator()* for *Range* arguments, giving the corresponding substructure as result. Thus, the subvector structure $x = (y, z)$ may be almost equivalently realized as

```
RealVec x(5);
RealVec y = x(Range(1, 3)), z = x(Range(4, 5));
```

The only difference is that an additional temporary view is created as the argument of the copy constructor. A more helpful example is the assignment to a submatrix:

```
RealMat A(10), B(5);
A(Range(1, 5), Range(1, 5)) = B;
```

Here, the 5×5 matrix B is copied (elementwise) to the upper left submatrix of A.

It is also possible to interpret a matrix as a vector and vice versa. As an example, we view the six elements of v once as a $(2,3)$-matrix A and once as a $(3,2)$-matrix B.

```
RealVec v(6);
RealMat A(v, 2, 3), B(v, 3, 2);
```

The storage scheme looks like follows:

$$v = \boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\boxed{6} \quad \longrightarrow \quad A = \begin{array}{|c|c|c|}\hline 1 & 3 & 5 \\ \hline 2 & 4 & 6 \\ \hline \end{array} \qquad B = \begin{array}{|c|c|}\hline 1 & 4 \\ \hline 2 & 5 \\ \hline 3 & 6 \\ \hline \end{array}$$

Hence, the matrix elements $A(2,2)$ and $B(1,2)$ both point to the fourth component $v(4)$ of v.

5.6.4 Copy Constructor and Temporary Objects

Besides the access to substructures, the most important application of the copy constructor are temporary objects. Suppose you have defined a function

```
RealVec Result() {
    RealVec result(5);
    ...
    return result;
}
```

returning a temporary *RealVec* object. What happens if we use this function in a copy constructor?

```
RealVec x = Result();
```

In the procedure *Result*, the compiler creates a new vector with a new block of data. This temporary object is then used as argument for the copy constructor of x. Hence, x becomes a new view on the data of the temporary variable. These data now have two references. The scope of the temporary variable ends behind the copy constructor so that its destructor is called afterwards. The temporary object is killed, but its data object is not deleted since there is still a reference left. As a consequence, x is a view on the result of the procedures. Note that only the pointer to the data is copied rather than the data themselves.

This usage of the copy constructor and temporary objects is very helpful for an efficient implementation of binary operators. As an example, the expression

```
RealVec y(5), z(5);
RealVec x = y + z;
```

creates a new vector x that contains the sum of y and z without copying. Thus, we achieve almost C efficiency. Care must be taken with expressions such as

```
RealVec x(5), y(5), z(5);
x = y + z;
```

Here, the copying is inavoidable. Furthermore, in contrast to a C implementation of the same expression, a temporary object is first created and then destroyed including its data. Thus, it is often more efficient to introduce a new variable (see also section 5.6.5). Moreover, the use of the unary operators +=, −=, etc., is always very efficient.

5.6.5 Copy and Reference

Suppose you want to get a physical copy of a vector *RealVec* x. You can not use a copy constructor

```
RealVec y = x;
```

to this end, since it only makes a copy of the *view* but not of the *data*. Therefore, we defined for each *MathView* class a member function *Copy()* that returns a complete physical copy of the given object. Hence, the line

```
RealVec y = x.Copy();
```

will do the job. Now suppose that you want to change an already created vector *RealVec* x. In this case, the assignment operator is of no use, since it does not change any view. Hence, we introduce a member function *Reference(MathView&)* that changes the view to the given argument. Thereby, the old view is destroyed. Thus, the function *Reference* may be interpreted as an assignment for the views in contrast to the elementwise assignment operator. Consequently, the expression

```
x.Reference(y);
```

changes the vector x to the given vector y. Afterwards, x and y point to the same data. Moreover, the line

```
x.Reference(y.Copy());
```

changes x to a physical copy of y. As this situation occurs quite often, we have defined a corresponding function *Copy(MathView&)* so that

```
x.Copy(y);
```

does the same.

6 The ODE Library

The basic idea behind the classes was to provide as much flexibility as possible while keeping the interfaces simple for standard applications. Thus, the user should be able to define his own solvers, scaling procedures or trajectories, but he/she does not need to do so in order to solve a standard ODE. In C++ this means that we had to make extensive use of virtual functions which are almost always predefined but could be overloaded by the user. This concept supersedes the function pointers used in the C implementation. As an example, the user may explicitly define the derivative of the ODE's right hand side, or he/she may trust the default derivative by numerical differentiation.

6.1 The Class Ode

The class *Ode* represents the base class for first order ordinary differential equations

$$x' = f(x, t)$$

each ODE problem has to be derived from. It contains a series of virtual member functions describing

- the right hand side f and (optionally) its derivative
- the handling of implicit discretizations (linear solver etc.)
- a user defined scaling
- a parameter dependence

In the simplest case, the derived user defined ODE only has to define the right hand side f. The other members are all predefined. Thus, the trivial scalar ODE $x' = x$ is defined as follows:

```
class TrivialOde : public Ode {
public:
    TrivialOde() : Ode(1) { autonomous = true; }

    virtual Bool f(const RealVec& x, Real t, const RealVec& fx);
    virtual Bool Df(const RealVec& x, Real t, const RealMat& A);

    char* Name() const { return "TrivialOde"; }
};

Bool TrivialOde::f(const RealVec& x, Real t, const RealVec& fx) {
    fx(1) = x(1);
    fEval++;
    return true;
}

Bool TrivialOde::Df(const RealVec& x, Real t, const RealMat& A) {
    A(1, 1) = 1;
    dfEval++;
    return true;
}
```

6.2 The Ode Solver Classes

The classes for the ODE solvers were designed along the lines of the abstract order and stepsize control for evolutional h-p methods and their application to extrapolation codes as described in the first part. The base class *Integrator* only provides some formal functionality such as setting the initial time t_0, the final time t_1, and the accuracy tol. The problem to be solved is set by the abstract virtual method *SetOde(Ode&)*. Moreover, it defines the usage of trajectories (see Section 6.3) and protocols (see Section 6.4). A typical command sequence for a standard ODE looks as follows (*Eulex* is an integrator derived from the base class *Integrator*, see Section 6.2.3).

```
main() {
    TrivialOde ode;
    Integrator *integrator = new Eulex;

    integrator->SetOde(ode);
    integrator->TStart(0);
    integrator->TEnd(1);
    integrator->Accuracy(1e-5);

    RealVec x0(1), x(1);
    x0 = 1;
    if (integrator->Solve(x0, x)) {
        cout << "solution x = " << x << endl;
    }
    else cout << "no solution found" << endl;

    delete integrator;
}
```

6.2.1 The Class HpIntegrator

The class *HpIntegrator* realizes the abstract evolutional h-p method as described in Section 1. It is based on the abstract virutal method *DiscretizationScheme* which is nothing but the discretization scheme (T_k, E_k) introduced in Section 1. The abstract virtual methods *Work(Int k)* and *Info(Int k)* represent the amount of work and information sequences A_k and B_k, respectively. The basic order p is set in the constructor. Due to the protocol

facilities (see Section 6.4) we distinguish two modes of operation: The *adaptive mode*, where the orders and stepsizes are chosen adaptively as outlined in Section 1, and the *slave mode*, where the orders and stepsizes are taken from a given protocol. Accordingly, there are two assiciated integration procedures, *AdaptiveSolve* and *SlaveSolve*. A single adaptive h-p step is performed by the function *AdaptiveStep*, whereas *SlaveStep(const HpStep&)* realizes a single step of the given protocol.

At the beginning of each step the function *PrepareStep* (adaptive mode) or *PrepareSlaveStep* (slave mode) is called. Here, the derived integrator may compute all the information needed at the initial point $x(t)$ of the integration step. In an explicit method this might be the function value at (x, t). In a linearly implicit method we may also need the evaluation of the Jacobian $f_x(x, t)$ at (x, t). If the solution is written to a trajectory, the method *SavePoint* is called after each accepted integration step.

For a flexible scaling strategy, the algorithm uses the virtual methods *Norm* and *Rescale*. The former defines the current (scaled) norm, whereas *Rescale* is called after each accepted integration step to update the scaling.

6.2.2 The Class ExMethod

The class *ExMethod* implements an abstract extrapolation method as a special case of an evolutional h-p method as described in Section 2. Hence, the discretization scheme for the h-p method (method *DiscretizationScheme* in its base class *HpIntegrator*) is defined by extrapolating a basic discretization scheme over the inner stepsizes given by a subdivision sequence. The basic integration formula $D_n^{t+H,t}(x)$ introduced in Section 2 has to be given as the abstract virtual method *BasicIntegrator*. The subdivision sequence n_k is represented by the virtual method *Sequence(Int k)*.

In addition to the functions needed for the h-p method, the class *ExMethod* contains the major parts of the algorithm for continuous output according to Hairer and Ostermann [HO90]. If the derived extrapolation method (e.g. *Eulex* or *Eulsim*) provides the table f of function values at the inner grid points, the method *ForwardDerivatives* may be used to compute an extrapolated approximation of the derivatives at the beginning of the integration step. These approximation are then used in *HermiteTrajectory* (see Section 6.3) for the Hermite interpolant defining the continuous approximation of

the trajectory.

6.2.3 The Class Eulex

The simplest example of an extrapolation code is based on the explicit Euler discretization. The corresponding class *Eulex* defines the *BasicIntegrator* of its base class *ExMethod* as the given number of explicit Euler steps. As preparation of a single h-p step, we only have to evaluate the ODE's right hand side f at the beginning (x, t) of the integration interval. This value is then saved for the forthcoming basic integration steps.

6.2.4 The Class Eulsim

The extrapolation code *Eulsim* based on the linearly implicit Euler formula for stiff ODEs is only sligthly more complicated. In the preparation procedure, we not only have to evaluate the right hand side f, but in addition the Jacobian $f_x(x, t)$. Note that to this end, the virtual method *Jacobian* of the given *Ode* is called that may be redefined by the user. Furthermore, we have incorporated the residual oriented monotonicity test as sketched in Section 2.3. It is realized in the method *MonotonicityTest*.

6.3 The Trajectory Classes

If not only the solution at the final time is of interest, we have to save some information in the course of the integration process. To this end, we introduced some classes describing trajectories.

The abstract class *Trajectory* may be used to store information about the trajectory approximated by the integration process. Once a *Trajectory* is given to an *Integrator* by *Integrator::OpenTrajectory*, the integrator calls the virtual function

```
Trajectory::SavePoint(const RealMat& x, Real t, Int p)
```

after each integration step. The matrix x contains the solution $x(t)$ at the current time t and the derivatives $x'(t), \ldots, x^{(p)}(t)$ as column vectors.

As an example, the class *HermiteTrajectory* provides continuous output by Hermite interpolation of the pointwise solution obtained in the integration process. Therefore, the solution and all available derivatives are stored

at each integration point (class *IntegrationPoint*). In a second step, this information is transferred to the Hermite coefficients of the interpolating polynomials (class *IntegrationStep*). Thus, we obtain a piecewise polynomial approximation of the solution. To get the solution at an arbitrary time t, we look for the enclosing integration interval in the list of integration steps and evaluate the corresponding Hermite polynomial.

In the following sample program we employ the Hermite interpolation classes to print the solution for $t = 0, 0.1, 0.2, \ldots, 1$.

```
TrivialOde ode;
Eulex eulex(ode);
HermiteTrajectory phi;

eulex.TStart(0);
eulex.TEnd(1);
eulex.Accuracy(1e-5);
eulex.OpenTrajectory(phi);

RealVec x(1);
x(1) = 1;

if (eulex.Solve(x, x)) {
    cout << "solution x(1) = " << x << endl;
    for (Real t=0; t<=1; t+=0.1) {
        cout << "solution x(" << t << ") = " << phi(t) << endl;
    }
}
```

6.4 The Protocol Classes

For some applications it is useful to save information about the integration process such as stepsizes and orders. As an example, we may wish to 'freeze' the order and stepsize sequence if two or more similar initial value problems are to be solved, thus avoiding the adaptivity overhead. A typical application is the integration of a variational equation along a given trajectory. In this context it is often feasible to switch off the adaptive stepsize control and to employ the order and stepsize sequence of the integration process for the trajectory, i.e., the original ODE.

To realize this kind of stepsize freezing, we defined the class *IntegrationProtocol* and the associated member functions of the *Integrator* class. These functions only provide a formal framework for the use of protocols. The derived integrator classes determine which kind of data is actually needed to protocol the integration process. The h-p integrator, for example, uses the class *HpProtocol* which mainly consists of the list of order-stepsize pairs. An *Integrator* should be able to construct a new protocol by the virtual function *Integrator::NewProtocol*. The usage of protocols is most easily described by the following example.

```
TrivialOde ode;
Integrator *integrator = new Eulsim(ode);
IntegrationProtocol *protocol = integrator->NewProtocol();

RealVec x0(1), x(1);

integrator->Accuracy(1e-5);
integrator->TStart(0);
integrator->TEnd(1);

x0 = 1;
integrator->OpenForWriting(*protocol);
if (integrator->Solve(x0, x)) cout << "x = " << x << endl;
integrator->CloseProtocol();

x0 = 1.1;
integrator->OpenForReading(*protocol);
if (integrator->Solve(x0, x)) cout << "x = " << x << endl;

delete protocol;
delete integrator;
```

We first get a new protocol from the current integrator and set the standard arguments (initial and final time, accuracy) of the integrator. Then, we solve the (trivial) ODE for the initial value $x_0 = 1$ writing the information about the integration process to the protocol. In a second go, we use this protocol to solve the same ODE for the initial value $x_0 = 1.1$. Here, the integrator employs the order and stepsize sequence from the first integration. In other words, the two solutions are computed by the same formula. Obviously,

this saves the adaptivity overhead but not necessarily guarantees a correct solution, even for small perturbations of the initial value. On the other hand, the thus obtained approximation depends smoothly on the initial value, since it corresponds to a fixed integration formula.

References

[Deu83] P. Deuflhard. Order and stepsize control in extrapolation methods. *Numer. Math.*, 41:399–422, 1983.

[Deu89] P. Deuflhard. Uniqueness Theorems for Stiff ODE initial Value Problems. In *Proceedings 13th Biennial Conference on Numerical Analysis*, pages 74–88. University of Dundee, 1989.

[GB86] W. Gui and I. Babuška. The h, p and h-p versions of the finite element method in one dimension, Part 1 to 3. *Numer. Math.*, 49:577–683, 1986.

[HNW87] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I, Nonstiff Problems*. Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1987.

[HO90] E. Hairer and A. Ostermann. Dense ouput for extrapolation methods. *Numer. Math*, 58:419–439, 1990.

[Hoh93] A. Hohmann. *Inexact Gauss Newton Methods for Parameter Dependent Nonlinear Problems*. PhD thesis, Freie Universität Berlin, 1993.

[HW92] A. Hohmann and C. Wulff. Modular design of extrapolation codes. Technical Report TR 92-5, Konrad-Zuse-Zentrum, Berlin, 1992.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.