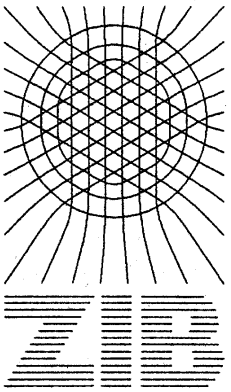
Konrad-Zuse-Zentrum für Informationstechnik Berlin



ANDREAS HOHMANN AND CLAUDIA WULFF

Modular Design of Extrapolation Codes

Technical Report TR-92-5

LSIM, and DIFEX. In addition, we amused ourselves by using the general extrapolation module for the adaptive Romberg quadrature algorithm TRAPEX. Besides the claims for flexibility and modularity, the efficiency of the programs should not suffer. For that purpose we required the number of function evaluations to be comparable to the existing FORTRAN codes. For large problems the organization overhead will be of less importance.

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 $\phi_{t_0,t_1}(y_0)$ into appropriate subproblems $\phi_{t,t+H}(y)$ 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.

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

Substituting the unavailable error by the norm of the error estimator E_k, we require

1 Order and Stepsize Control

In this section we discuss the order and stepsize control for extrapolation method as proposed by Deuflhard in [1]. To avoid the double indices connected with extrapolation tableaus, we use a slightly different notation. In fact, the control mechanism works only on the diagonal and subdiagonal entries of the extrapolation tableau. Therefore, all we need to apply the techniques of [1], is a discretization scheme of variable order (with respect to a stepsize) and a corresponding error estimator.

To start with, consider the standard initial value problem of non-autonomous ODE's

$$\dot{y} = f(y,t), \quad y(t_0) = y_0, \quad y(t_1) = ?$$

If ϕ denotes the corresponding flow, which we assume to exist for all t in question, we have to compute $y_1 = \phi_{t_0,t_1}(y_0)$ up to a prescribed accuracy tol. To solve this problem, we are given a family of discretization schemes $T_k(t,H)$, $k = 1,2,\ldots$, to approximate a subproblem $\phi_{t,t+H}(y)$. Thus, each T_k represents a fixed formula which computes for a given initial value y, time t, and stepsize H an approximation of $\phi_{t,t+H}(y)$.

We suppose that T_k is computed using the former approximations $T_1,\ldots,T_{k-1}$ and that the order of the discretization T_k with respect to the stepsize H is pk for some fixed $p \in \mathbf{N}$, i.e., the approximation error ε_k of T_k satisfies

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

for some proportionality factor $\gamma(t)$ only depending on the subproblem given by f, t and y. In addition, we need a corresponding family of error estimators $E_k(t,H)$ such that

$$\phi_{t,t+H}(y) - T_k(t,H) = E_k(t,H) + o(H^{pk+1}) \,. \tag{1.2}$$

In what follows, we will call k the *order* of the approximation, although this is not true but for $p = 1$.

In the extrapolation context, which we obviously have in mind, the approximations T_k are the subdiagonal entries $T_k = T_{k+1,k}$ of the extrapolation tableau $\{T_{ik}\}$ corresponding to the subproblem $\phi_{t,t+H}(y)$ with the outer stepsize H. 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

$$E_k(t,H) = T_{k+1,k+1} - T_{k+1,k} \,.$$

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 $\phi_{t_0,t_1}(y_0)$ into appropriate subproblems $\phi_{t,t+H}(y)$ 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.

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

Substituting the unavailable error by the norm of the error estimator E_k, we require

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

If this error criterion is fulfilled, we say that the method *converged* for the subproblem $\phi_{t,t+H}(y)$ 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, i.e., satisfy $\varepsilon_k(t,H_k) = \text{tol}$:

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

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

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

as the expected *optimal stepsize* for T_k. Assuming that the proportionality coefficient $\gamma(t)$ 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

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

Deuflhard proposed in [1] 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 $E_k^{(\text{in})}$ is supposed to be proportional to the number B_k of function evaluations needed for T_k,

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

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

$$E_k^{(\text{out})} = -\operatorname{ld} \varepsilon_k(t, H)$$

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

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

for some proportionality factor β, or equivalently

$$-\operatorname{ld} \varepsilon_k(t, H) = cB_k \tag{1.5}$$

for some constant c. Of course, (1.5) won't be true in reality, but if we know a single error ε_q for some q, we may use (1.5) to define estimates $\alpha_k^{(q)} = \alpha_k^{(q)}(\varepsilon_q)$ for all errors ε_k by

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

satisfying $\alpha_k^{(q)} = \varepsilon_q$ and the relationship $-\operatorname{ld} \alpha_k^{(q)} = cB_k$ for some $c > 0$ as derived from the (linear) information theoretic model. These a priori estimates $\alpha_k^{(q)}$ constitute the base of the convergence monitor.

1.3 Convergence Monitor

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

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

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

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

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.,

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

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

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

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

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

In 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

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

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

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

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 1 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, i.e., $\varepsilon_k(t, H_k) = \rho\,\text{tol}$. Therefore we expect

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

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

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

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*

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

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 $\|E_{k_{\text{opt}}}(t, H_{\text{red}})\| = \rho\,\text{tol}$.

1.6 Determination of a feasible maximal order

So far, we have no upper limit for the order k. Based on the information theoretic model, we can determine a maximal order $k_{\max}$ for which we expect computational profit. More precisely, we compute the smallest order k such that using the next order would be more expensive, i.e.

$$W_{k+1} > W_k \quad \text{or equivalently} \quad \frac{A_{k+1}}{A_k} > \frac{H_{k+1}}{H_k}.$$

Obviously, we have no stepsizes H_k at hand, but we can give 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

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

2 Application to Extrapolation Methods

In this section we want to 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 $D(y,t,h,n)$, where y and t are the initial value and time, and h the (inner) stepsize. This notation will become clear by the following examples.

Example 1. *Explicit Euler discretization (code* Eulex*).* The discretization $D(y,t,h,n) =: y_n$ may be recursively defined by

a) $y_0 := y,\ t_0 := t$

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

Example 2. *Semi-implicit Euler discretization with fixed Jacobian (code* Eulsim*).* $D(y,t,h,n) = y_n$ is recursively defined by

a) $y_0 := y,\ t_0 := t$

b) $y_{i+1} := y_i + h(I - hf_y(y,t))^{-1}y_i,\ t_{i+1} := t_i + h$ for $i = 0,\ldots,n-1$.

Example 3. *Trapezoidal sum discretization used for quadrature (code* Trapex*).*

$$D(y,t,h,n) := y + h\left(\frac{1}{2}(f(t) + f(t+nh)) + \sum_{j=1}^{n-1} f(t+jh)\right) .$$

Example 4. *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 $D(y,t,h,n) = x_n$.

a) Explicit Euler start step:

$$y_0 := y,\ \ t_0 := t,\ \ y_1 := y_0 + hf(y_0,t_0),\ \ t_1 := t_0 + h,$$

b) Explicit mid-point rule:

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

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

2.2 Order and Stepsize Control in the Extrapolation Context

As already mentioned in section 1 the main application of the rather general order and stepsize control are extrapolation methods. We denote by $\{T_{ik} = T_{ik}(t, H)\}$ the extrapolation table corresponding to the discretization scheme $D(y, t, h, n)$. More precisely, let $n_1 < n_2 < \cdots$ be a subdivision sequence and define $\{T_{ik}\}$ by the well-known formula for polynomial extrapolation with respect to h^p:

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

Since the subdiagonal entries $T_{k+1,k}$ are of order pk, they constitute a family T_k of approximations as defined in section 1. 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 $D(y, t, H/n_i, n_i)$ for $i = 1, \ldots, k+1$.

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 $D(y, t, H/n_i, n_i)$ for $i = 2, \ldots, k+1$.

We continue our four examples by calculating the corresponding sequences $\{A_k\}$ and $\{B_k\}$.

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 $f(y_0, t_0)$ is computed only once at the beginning of the extrapolation process, we must subtract 1 in the last formula.

EXAMPLE 2. *Semi-implicit Euler discretization with fixed Jacobian (code* EULSIM*).* The subdivision sequence of EULSIM is again the harmonic sequence. For the semiimplicit 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 $f_y(y_0)$
C_{LU}	cost of an LU-decomposition of a (n, n)-matrix
C_{subst}	cost of a backward and forward substitution of a linear triangular system

Using this information, we get the amount of work sequence

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

In the present implementation of EULSIM we set

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

EXAMPLE 3. *Trapezoidal sum discretization used for quadrature (code* TRAPEX*).* For the quadrature problem the function f only depends on the time t. Therefore, we need to evaluate f only once for a given time t independent from the current state variable y (this is different from a true initial value problem where we always have to use the latest approximation of $y(t)$ to evaluate $f(y(t), t)$). For the so-called *Bulirsch sequence*

$$n_i = \begin{cases} 2^k & : \ i = 2k, \ k = 1, 2, \ldots \\ 3 \cdot 2^{k-1} & : \ i = 2k+1, \ k = 1, 2, \ldots \\ 1 & : \ i = 1 \end{cases},$$

the sequence A_k is given by the following formulas (the initial value $f(t_0)$ is assumed to be available from the preceding step):

$$A_0 = 1, \quad A_2 = 4, \quad A_i = \begin{cases} A_{i-1} + n_{i+1}/2 & : \quad i \quad \text{odd} \\ A_{i-1} + n_{i+1}/3 & : \quad i > 2 \quad \text{even} \end{cases} .$$

EXAMPLE 4. *Explicit mid-point rule (code* DIFEX*).* The subdivision sequence used in DIFEX is the double harmonic sequence $n_i = 2i$, $i = 1, \ldots$. The sequence A_k is given by

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

Note that due to Gragg's final step one additional f-evaluation is needed for the computation of $T_{i,1}$ for $i = 1, 2, \ldots$.

Computation of the sequences $\{B_k\}$. Let $\bar{A}_k$ denote the information needed to compute $T_{2,1}, \ldots, T_{k+1,1}$. For ODE's the sequences $\bar{A}_k$ and B_k are related by

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

In EULEX, TRAPEX 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 $(I - hf_y(y_0))^{-1} f(y)$ may be counted. Therefore $\{\bar{A}_k\}$ and $\{A_k\}$ differ. $\{\bar{A}_k\}$ 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

Sometimes the discretization scheme provides additional information to restrict the order or stepsize of the surrounding extrapolation method (or more general: the surrounding method of variable order and stepsize). As an example, we will discuss the integration of a stiff ODE using the semiimplicit Euler discretization. Following Deuflhard [3], there are two tools which may be used to control the stepsize: The estimation of the *logarithmic norm* μ of the Jacobian and the *natural monotonicity test* for the first step of the Newton iteration which constitutes the implicit Euler discretization.

If $\mu = \mu(A)$ denotes the logarithmic norm

$$\mu(A) = \sup_{x \neq 0} \frac{\langle x, Ax \rangle}{\langle x, x \rangle}$$

of the Jacobian $A = f_y(y,t)$, an investigation of the convergence of the Newton method associated with the implicit Euler discretization leads to the stepsize bound

$$\mu h < 1 \,. \tag{2.1}$$

Replacing the unavailable logarithmic norm μ by local estimates

$$\mu(x) := \frac{\langle x, Ax\rangle}{\langle x, x\rangle} < \mu$$

and the 1 by some upper bound $c_{\mu h}^{(\max)} < 1$, we can use (2.1) in the algorithm to restrict the stepsize by

$$\mu(x)\, h < c_{\mu h}^{(max)} \,.$$

If this test is violated, we return $c_{\mu h}^{(\mathrm{safe})}/\mu(x)$ as the maximal allowed stepsize, where $c_{\mu h}^{(\mathrm{safe})} < c_{\mu h}^{(\max)}$ is a safe upper bound for the product μh. In the present implementation we have chosen $c_{\mu h}^{(\max)} := 0.9$ and $c_{\mu h}^{(\mathrm{safe})} := 0.5$.

Concerning the natural monotonicity test, we regard the nonlinear equation connected with the implicit Euler discretization:

$$F(y) := y - y_0 - hf(y, t_0) = 0$$

The semiimplicit Euler discretization may be viewed as the first step of the simplified Newton method

$$y^0 := y_0, \quad y^{k+1} := y^k + \Delta y^k, \quad \Delta y^k := -F'(y_0)F(y^k)$$

for this nonlinear equation. The Jacobian of F is $F'(y_0) = I - hA$, where $A := f_y(y_0, t_0)$ is the derivative of f at the initial value and the first Newton correction

$$\Delta y^0 = h(I - hA)^{-1} f(y_0)$$

is just the correction $\Delta y_0 = y_1 - y_0$ of the semiimplicit Euler discretization. The convergence of the simplified Newton method may be checked using the *natural monotonicity test*

$$\theta := \frac{\|\bar{\Delta} y^1\|}{\|\Delta y^0\|} \;<\; \bar{\theta} := \frac{1}{4} \,,$$

where $\bar{\Delta} y^1 := -F'(y_0)F(y^1)$ is the first *simplified Newton correction*. Since we employ only the first step of the Newton method, we require the *monotonicity coefficient* θ to be well below the maximal coefficient $\bar{\theta}$ and check

$$\theta \le \theta_{\max} := \frac{1}{8} \,. \tag{2.2}$$

This monotonicity test needs, besides the ordinary Newton correction $\Delta y^0 = \Delta y_0$, being already computed for the semiimplicit Euler discretization, the simplified Newton correction $\bar{\Delta} y^1$. In our context, we get

$$\bar{\Delta} y^1 = (I - hA)^{-1}(y_1 - y_0 - hf(y_1, t_1)) = (I - hA)^{-1}\Delta y_0 - \Delta y_1 \ ,$$

using the correction $\Delta y_1 = (I - hA)^{-1} f(y_1, t_1)$ of the next discretization step. In addition, we have to solve another linear system to calculate $(I - hA)^{-1}\Delta y_0$. If (2.2) is violated, the stepsize should be reduced by an ad hoc factor, say, $\lambda_{\text{red}} = 0.7$.

2.4 Norms and Standard 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 y_i of the state variable y. To this end, we introduce a *scaled norm*

$$\|y\|_{scal} := \frac{1}{\sqrt{n}}\|D^{-1}y\|_2 = \left(\frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i}{s_i}\right)^2\right)^{1/2}$$

where $D = \text{diag}(s_1, \ldots, s_n)$ is the *current scaling matrix*. Using a scaled norm in the accuracy check (1.3) also allows 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 $\|E_k(t, H)\| < \text{tol}$ 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 have to use the following information:

y	current solution vector
y^{last}	last accepted solution (last time step)
y^{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
$y^{\text{userScale}}$	componentwise absolute lower bounds for the scaling factors.

Moreover, we have to distinguish three phases of scaling. 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. During a single integration step we have to compute an appropriate scaling for each order k, but without updating y^{last} and y^{max} (*intermediate scaling*). The latter has to be done when we achieved prescribed accuracy and thus accepted the approximation (*rescaling*).

- Initialization: for $i = 1, \ldots, n$

$$s_i := y_i^{\text{max}} := \max\{|y_i^{\text{start}}|, y_i^{\text{userScale}}, s_{\text{abs}}\}$$

- Intermediate scaling: for $i = 1, \ldots, n$

nonstiff case: $s_i := \max\{|y_i|, |y_i^{\text{last}}|, y_i^{\text{max}} \cdot s_{\text{rel}}, s_{\text{abs}}, y_i^{\text{userScale}}\}$

stiff case: $s_i := \max\{|y_i|, y_i^{\text{max}}, s_{\text{abs}}\}$

- Rescaling: as intermediate scaling, in addition

$$y_i^{\text{max}} := \max\{y_i^{\text{max}}, |y_i|\} \quad \text{for } i = 1, \ldots, n \text{ and } y^{\text{last}} := y.$$

2.5 Scaling for Variational Equations

The standard scaling described in the last section is only feasible for a general initial value problem without any knowledge about its inner structure. As an example, we will study the variational equation

$$\dot{W} = f_y(y,t)\,W, \quad W(t_0) = I, \quad W(t_1) = ?$$

for the Wronskian $W(t) = \partial y(t)/\partial y(0) \in \mathrm{Mat}_n(\mathbf{R})$. Together with the original problem $\dot{y} = f(y,t)$, $y(t_0) = y_0$, we get an initial value problem of dimension $n + n^2$.

$$\begin{pmatrix} \dot{y} \\ \dot{W} \end{pmatrix} = f_{\mathrm{var}}(y, W, t) := \begin{pmatrix} f(y,t) \\ f_y(y,t)\, W \end{pmatrix}, \quad \begin{pmatrix} y(t_0) \\ W(t_0) \end{pmatrix} = \begin{pmatrix} y_0 \\ I \end{pmatrix}. \tag{2.3}$$

The Wronskian $W(t) : \mathbf{R}^n \to \mathbf{R}^n$ maps a perturbation $\partial y(t_0)$ of the initial value on the resulting perturbation $\partial y(t)$ of the solution $y(t)$,

$$\partial y(t_0) \longmapsto \partial y(t) = W(t)\partial y(t_0)\,,$$

Regarding the mapping of the scaled variables

$$D(t_0)^{-1}\partial y(t_0) \longmapsto D(t)^{-1}\partial y(t) = \left(D(t)^{-1}W(t)D(t_0)\right)\left(D(t_0)^{-1}\partial y(t_0)\right)\,,$$

we see that we have to scale the columns of $W(t)$ using the scaling $D(t_0)$ at the initial value $y(t_0)$, while the rows are to be scaled by the inverse of the current scaling $D(t)$ at $y(t)$. The column and row scalings correspond to a scaling of the domain and image of $W(t)$, respectively. Thus, the scaling of the Wronskian part of the IVP (2.3) depends on the scaling of the first n components belonging to the original IVP. An optimal choice for a norm of the Wronskian part would be a scaled spectral norm $\|D(t)^{-1}AD(t_0)\|_2$ for a Matrix $A \in \mathrm{Mat}_n(\mathbf{R})$. Since this is too expensive, we use the scaled Frobenius norm instead:

$$\|A\|_{\mathrm{scal}} := \frac{1}{n}\|D(t)^{-1}AD(t_0)\|_F = \frac{1}{n}\left(\sum_{i,j=1}^{n}\left(s_i(t)^{-1}a_{ij}s_j(t_0)\right)^2\right)^{\frac{1}{2}}$$

For a complete vector $(x, A) \in \mathbf{R}^{n+n^2}$ of the variational IVP (2.3) we end up with the scaled norm

$$\begin{aligned} \|(x,A)\|_{\mathrm{scal}} &= \frac{1}{\sqrt{n^2+n}}\left(\|D(t)^{-1}x\|_2^2 + \|D(t)^{-1}AD(t_0)\|_F^2\right)^{\frac{1}{2}} \\ &= \frac{1}{\sqrt{n^2+n}}\left(\sum_{i=1}^{n}\frac{x_i^2}{s_i(t)^2} + \sum_{i,j=1}^{n}\left(s_i(t)^{-1}a_{ij}s_j(t_0)\right)^2\right)^{\frac{1}{2}}. \end{aligned}$$

Almost the same considerations hold in case of parameter dependent ODE's $\dot{y} = f(y,t,\lambda)$, $\lambda \in \mathbf{R}^q$, if we want to know the derivative

$$P(t) = \partial y(t)/\partial\lambda \in \mathrm{Mat}_{n,q}(\mathbf{R})$$

with respect to the parameter λ. $P(t)$ is the solution of the linear inhomogeneous initial value problem

$$\dot{P} = f_\lambda(y,t,\lambda) + f_y(y,t,\lambda)P, \quad P(t_0) = 0\,.$$

The parameter derivative $P : \mathbf{R}^q \to \mathbf{R}^n$ maps a perturbation $\partial\lambda$ of the parameter on the corresponding perturbation $\partial y(t)$ of the solution. Hence, the rows (image) are to be scaled by the current scaling $D(t)$ of $y(t)$, while we use for the columns (domain) an appropriate scaling $D_\lambda = \mathrm{diag}(s_1,\ldots,s_q)$ of the parameter. Thus, a suitable norm for the parameter part is

$$\|B\|_{\mathrm{scal}} := \frac{1}{\sqrt{nq}}\|D(t)^{-1}BD_\lambda\|_F = \left(\frac{1}{nq}\sum_{i=1}^{n}\sum_{j=1}^{q}\left(s_i(t)^{-1}b_{ij}s_j\right)^2\right)^{\frac{1}{2}}$$

for a matrix $B \in \mathrm{Mat}_{n,q}(\mathbf{R})$.

2.6 Advanced Scaling

Using the scaling strategy described in section 2.4, we control the *local* relative discretization error, although we would definitely prefer to control the *global* relative error. The following scaling strategy which was proposed by Deuflhard [2] in fact allows us to achieve

$$\|y_m - \phi_{t_0,t_m}(y_0)\|_{\mathrm{scal}} \;\le\; m\ \mathrm{tol}\,,$$

where tol is the prescribed accuracy and y_m the approximation of $\phi_{t_0,t_m}(y_0)$ obtained after m discretization steps. To fix notation, let

$$t_{\mathrm{start}} = t_0 < t_1 < \cdots < t_m = t_{\mathrm{end}}$$

be the time steps, automatically chosen by the integration method with (outer) stepsizes $h_j := t_{j+1} - t_j$, and let $y_i \approx \phi_{t_{i-1},t_i}(y_{i-1})$ be the local approximations. The algorithm always controls the local discretization error $\delta y_i := y_i - \phi_{t_{i-1},t_i}(y_{i-1})$ by

$$\|\delta y_i\|_{\mathrm{scal}} \;\le\; \mathrm{tol}$$

in some scaled norm $\|\cdot\|_{\mathrm{scal}}$. In addition, we denote the global discretization error by $\delta y(t_i) := y_i - \phi_{t_0,t_i}(y_0)$. In linearized theory the errors are propagated by the Wronskian $W(s,t) = \partial y(s)/\partial y(t)$ along the solution, leading to the recurrence formula

$$\delta y(t_{j+1}) \;\doteq\; \delta y_{j+1} + W(t_{j+1},t_j)\,\delta y(t_j) \tag{2.4}$$

for $j = 0, \ldots, m-1$. Hence, if we want to give an upper bound for the global discretization error $\delta y(t_{j+1})$, we have to estimate the norm $\|W(t_{j+1}, t_j)\|$ of the Wronskian. If the right hand side f of the ODE satisfies the one-sided Lipschitz condition

$$\langle f(u) - f(v), u - v \rangle \leq \mu \langle u - v, u - v \rangle \quad \text{for all } u, v, t \tag{2.5}$$

the Wronskian is bounded by

$$\|W(s,t)\| \;\leq\; e^{\mu(s-t)} \,,$$

where $\|\cdot\|$ is the spectral norm corresponding to the scalar product in (2.5).

Next we turn to the scaled situation. Again, $D_j = \mathrm{diag}(s_1(t_j), \ldots, s_n(t_j))$ denotes the scaling matrix for time t_j. The resulting scaled variables are marked by a bar, e.g. $\bar{y}_j = D_j^{-1} y_j$ is the scaled approximation, $\delta\bar{y}_j = D_j^{-1}\delta y_j$ the scaled local error, and so on. The recurrence formula (2.4) translates to

$$\delta\bar{y}(t_{j+1}) \;\doteq\; \delta\bar{y}_{j+1} + \bar{W}(t_{j+1}, t_j)\, \delta\bar{y}(t_j) \,,$$

where $\bar{W}(t_{j+1}, t_j) := D_{j+1}^{-1} W(t_{j+1}, t_j) D_j$ is the scaled Wronskian. If the local error is controlled by $\|\delta\bar{y}(t_j)\| \leq \mathrm{tol}$, the global error is bounded by

$$\|\delta\bar{y}(t_{j+1})\| \;\leq\; \mathrm{tol} + \|\delta\,\bar{y}(t_j)\|\, \|\bar{W}(t_{j+1}, t_j)\| \,. \tag{2.6}$$

Hence, to meet the global error bound $\|\delta\bar{y}(t_m)\| \leq m\,\mathrm{tol}$, the local errors should not be amplified in each step (2.6), i.e.

$$\|\bar{W}(t_{j+1}, t_j)\| \;\leq\; 1 \,.$$

To estimate this scaled norm of the Wronskian, we assume that we have at least an estimate of the one-sided Lipschitz constant μ_j of f with respect to the scaled product $\langle D_j^{-1}\cdot, D_j^{-1}\cdot\rangle$ at hand. We obtain

$$\begin{aligned} \|\bar{W}(t_{j+1}, t_j)\| \;&=\; \|D_{j+1}^{-1} W(t_{j+1}, t_j) D_j\| \;\leq\; \|D_{j+1}^{-1} D_j\|\, \|D_j^{-1} W(t_{j+1}, t_j) D_j\| \\ &\leq\; \max_{i=1,\ldots,n} \frac{s_i(t_j)}{s_i(t_{j+1})} e^{\mu_j h_j} \,. \end{aligned}$$

Thus, the requirement $\|\bar{W}(t_{j+1}, t_j)\| \leq 1$ is fulfilled, if

$$s_i(t_{j+1}) \;\geq\; s_i(t_j) e^{\mu_j h_j} \quad \text{for } i = 1, \ldots, n \,.$$

If we require in addition that the scaling should not be more restrictive than the scaling obtained by the standard relative concept, we end up with the following scaling strategy:

$$s_i(t_{j+1}) = \max\{s_i(t_j) e^{\mu_j h_j},\, |y_i(t_{j+1})|,\, s_{\mathrm{abs}}\} \,.$$

Of course, the unavailable one-sided Lipschitz constant μ has to be replaced by some local estimate

$$\mu_j(x) = \frac{\langle D_j^{-1} A x, D_j^{-1} x \rangle}{\langle D_j^{-1} x, D_j^{-1} x \rangle} \leq \mu_j \ ,$$

where $A = f_y(y_j, t_j)$ is the Jacobian of f. For stiff problems, we already have this estimate at hand. That is why we implemented this scaling strategy only for stiff ODE's.

3 IMPLEMENTATION

The extrapolation method was realized in ANSI C using the GNU gcc compiler and debugger on a SUN SPARC station. So far, the package consists of the following modules:

mystd	standard types and procedures, machine dependent constants
message	simple message facility
MatVec	matrix and vector types and procedures, allocation and deallocation, utilities such as matrix vector multiplication, etc.
LRMat	types and procedures for the LR decomposition of a square matrix and the solution of a linear system.
intex	abstract extrapolation method with order and stepsize control
ivpScale	structure and procedures for the standard scaling in the IVP context
ivp	structure describing an initial value problem and the associated procedures (allocation, deallocation, etc.)
eulsim	extrapolation integrator based on the semiimplicit Euler discretization with fixed Jacobian
trapex	extrapolation method for quadrature based on the trapezoidal rule (Romberg quadrature)
difex	extrapolation integrator based on the explicit midpoint rule
variation	construction of the variational IVP corresponding to a given IVP

3.1 STYLE CONVENTIONS

For identifiers and the organization of the modules we use the following style conventions.

- Multi-word names capitalize each word but possibly the first one.
- Types and procedures start with a capital letter.

- Variables start with a small letter.
- Constants of enumeration type start with an 'e', with the exception of 'true' and 'false'.
- Global (extern) variables start with a 'g'.
- The following standard types are used:

  ```
  typedef double Real;
  typedef int Int;
  typedef enum {false, true} Bool;
  ```

- Each new structured type ⟨type⟩ is associated with allocation and deallocation procedures

  ```
  ⟨type⟩ *New⟨type⟩();
  void Free⟨type⟩(⟨type⟩ *);
  ```

  New⟨type⟩() may also have additional arguments (e.g. dimensions) needed for the allocation.
- The arguments of procedures are ordered according to:
  - 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., structures before reals before integers.

Sometimes compatibility with standard C headers and functions necessitates deviations from these rules.

3.2 MODULARIZATION

The modularization concept we employed for the extrapolation package was inspired by the object oriented approach, i.e., data strongly connected with the procedures acting on them. Unfortunately (and unlike C++), C does not provide such nice facilities as classes with its associated member functions which know the class data automatically. A C-procedure only knows its arguments and global (extern or module) variables. Moreover, there is no equivalent to PASCAL's WITH statement, which would present an alternative

(structure pointer as argument whose components are accessed using the WITH statement) avoiding the lengthy direct access to structure components.

To circumvent these difficulties we use the following strategy. A module contains data (in most cases pointers) as module variables which all procedures act on. If a procedure is called, these variables have to be set to the components of the structure the procedures should work on. Therefore, we call a globalization procedure at the beginning of every procedure exported by the module. If the procedure finished its job, we have to write the non-pointer variables back into the given structure.

In what follows we briefly describe each module, concentrating on the exported structures and procedures.

3.3 MODULE MYSTD

In addition the the definition of the standard types *Int, Real* and *Bool*, this module contains machine dependent constants such as the relative machine precision *epsMach*, which are set calling the initialization procedures *InitMystd*.

```
extern Real epsMach, sqrtEpsMach;

extern void InitMystd();
```

Moreover, the type *ScaleMode* is defined herein.

```
typedef enum SCALEMODE {
    eNoScale, eStandardScale, eUserScale
} ScaleMode;
```

3.4 MODULE MESSAGE

This module provides a simple message facility used all over the package. There are four procedures

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

using variable argument lists in the same way as the standard printf functions do. These four procedures are based on a single output function accessed by the global function pointer

```
extern void (*gPrint)(char *s);
```

which is by default set to the printf procedure on the calling shell.

3.5 Module MatVec

To implement vectors, matrices and tensors of type ⟨type⟩ with arbitrary index range, we use pointers to ⟨type⟩, pointers to pointers to ⟨type⟩, and so one. The module defines the corresponding types

```
typedef Real* RealVec;
typedef RealVec* RealMat;
typedef RealVec** RealTen;
typedef Int* IntVec;
typedef IntVec* IntMat;
```

and provides dynamic (de-)allocation procedures for various kinds of vectors, matrices and tensor. For example, *NewRealPtrVec(l, h)* gives a vector to pointers of Reals with the index range $[l \ldots h]$ and *NewRealUpMat(rl, rh, cl, ch)* an upper diagonal matrix with the accessible entries $(i, j) \in [rl \ldots rh] \times [cl \ldots ch]$ satisfying $i - rl \leq j - cl$. The memory for the entries of matrices and tensors are allocated in one go, so that they may be used as a vector. In addition, all entries are automatically set to zero in the allocation procedures. The rest of the module defines a bulk of useful procedures acting on vectors and matrices, such as addition of vectors, matrix vector multiplication, norms and scalar products, copy procedures, solution of triangular systems, etc..

3.6 Module LRMat

The LR decomposition of a square matrix is implemented using a structure *LRMat* which contains the required data.

```
typedef struct {
    RealMat AUser;
    RealMat A;
    IntVec pivot;
    Int n, signum;
    Bool rankDefect;
} LRMat;
```

AUser is a pointer to the matrix which should be decomposed, while A is the matrix used for the decomposition itself. *pivot* contains the pivoting permutation vector, n the dimension of the matrix and *signum* the sign of the permutation. For each matrix $A \in \mathrm{Mat}_n(\mathbf{R})$ to be decomposed, the user has to define an *LRMat*, e.g. by *ALR = NewLRMat(A, n)*, thereby allocating the appropriate vectors and matrices in the LRMat structure ALR and setting AUser to the given matrix A. If the procedures *LRDecompose(ALR)* is called, the contents of AUser is copied to A and the LR decomposition with partial pivoting is run on the matrix A (destroying the copy A but keeping AUser untouched). If the algorithm recognizes a rank defect, it returns false, otherwise true. Once decomposed, the matrix may be used to solve a linear system $Ax = b$ using the procedure *LRSolve(b, x)* which embodies the solution of the arising triangular systems.

3.7 MODULE INTEX

This module realizes the abstract extrapolation method with order and stepsize control for an arbitrary discretization scheme
$D(y_{\mathrm{start}}, t, h, k)$. The structure *ExtrapolationProblem* includes all information needed for the extrapolation process.

```
typedef struct {
    Bool (*BasicIntegrator)(RealVec yStart, Real t, Real h, Int k,
        Bool newStep, RealVec y, Real *reductionFactor);
    void (*InitSequence)(Int kMax, IntVec nSub);
    void (*InitAmountOfWork)(IntVec nSub, Int kMax,
        RealVec A, RealVec B);

    void (*IntermediateScale)(RealVec y);
    void (*Rescale)(RealVec y);
    Real (*ScaledDistance)(RealVec y, RealVec z);

    Real (*MaximalStepsize)(Real HConv);
    Int (*MaximalOrder)(Int kConv);

    RealVec yStart, y;
    Real tStart, tEnd, hStart, tol;
    Int n, p, kStart;
    Bool saveSteps;
    Int nStep, nStepMax;
```

```
    RealVec tSave;
    RealMat ySave;
} ExtrapolationProblem;
```

Obviously, n is the dimension of the problem, p the order of the basic discretization, *tStart* and *tEnd* the start and stop time, respectively. The user has to give the basic discretization scheme $D(y_{\text{start}}, t, h, k)$ by defining the *BasicIntegrator*. Note that the extrapolation method does not know the function itself but only the discretization scheme.

```
Bool (*BasicIntegrator)(RealVec yStart, Real t, Real h, Int k,
        Bool newStep, RealVec y, Real *reductionFactor);
```

In addition to the arguments y_{start}, t, h, and k, there is the boolean variable *newStep* indicating that the BasicIntegrator is called for the first time for the particular initial value (y_{start}, t). This information may be used to evaluate the function $f(y_{\text{start}}, t)$ (and its derivative, if used) only once. The BasicIntegrator may fail by returning false and providing a stepsize reduction factor.

The extrapolation method uses the user-defined procedures *InitSequence* and *InitAmountOfWork* to initialize the subdivision sequence $\{n_j\}$ (in the code called *nSub*) and the sequences $\{A_k\}$ and $\{B_k\}$, respectively.

Additional order and stepsize restrictions as described in section 2.3 may be added defining the procedures

```
Int (*MaximalOrder)(Int kConv);

Real (*MaximalStepsize)(Real HConv);
```

where k_{conv} and H_{conv} are the order and stepsize of the last accepted step.

The (scaled) norm used in the extrapolation algorithm to compute the error $\|E_k\| = \text{dist}(T_{k+1,k+1}, T_{k+1,k})$ is specified by the function

```
Real (*ScaledDistance)(RealVec y, RealVec z);
```

The procedures *IntermediateScale* and *Rescale* are called to give the user the opportunity to update his scaling for each new order k and after an accepted step, respectively.

The module also allows to save the time steps of the extrapolation process together with the corresponding approximations (e.g. to be used for graphical output). To this end, the boolean component *saveSteps* has to be set, causing

the time steps and approximations to be saved in the vector $tSave[0 \ldots nStep]$ and the matrix $ySave[0 \ldots nStep][1 \ldots n]$.

The extrapolation method is applied on a given extrapolation problem by calling the main procedure

```
extern Bool Intex(ExtrapolationProblem *);
```

3.8 MODULE IVP

To describe an initial value problem we employ the following structure:

```
typedef void (*Func)(RealVec y, Real t, RealVec fy);
typedef void (*Deriv)(RealVec y, Real t, RealMat dfy);

typedef enum  eStiff, eNonStiff, eQuadrature  IvpType;

typedef struct INITIALVALUEPROBLEM {
    Func f;
    RealVec yStart, yEnd, yScale;
    Real tStart, tEnd;
    Real hStart;
    Real tol;
    Int n;
    IvpType type;
    Int nStep, nStepMax, kStart;

    ScaleMode scaleMode;
    void (*InitScale)(struct INITIALVALUEPROBLEM *ivp);
    void (*IntermediateScale)(RealVec y);
    void (*Rescale)(RealVec y);
    Real (*ScaledDistance)(RealVec y, RealVec z);
    void *scale;

    Bool saveSteps;
    Int nSave;
    IvpSaveData *saveData;

    Deriv df;
    void (*Jacobian)(RealVec x, Real t);
    Bool (*NewtonJac)(Real h);
    Bool (*Solve)(RealVec b, RealVec x);
```

```
    Real (*LogarithmicNorm)(RealVec x);
    Real (*MonotonicityCoefficient)(RealVec deltaFirst, RealVec delta);
    Real (*LastScaledProduct)(RealVec y, RealVec z);
} InitialValueProblem;
```

At first sight, this structure seems to be rather complicated, but fortunately most components are predefined. In the simplest case of a small non-stiff ODE, it is sufficient to define the dimension n of the problem, the right hand side f, the initial values y_{start} and t_{start}, the stop time t_{end} and the required accuracy tol. The rest is chosen automatically. For small stiff ODE's, the semiimplicit method needs in addition the derivative *df*, and the type has to be set to *eStiff*, since it is by default set to eNonStiff.

It may also be useful to set the initial (outer) stepsize h_{start}, although the integration method behave rather robust with regard to the initial stepsize.

The scaling procedures are same as in the ExtrapolationProblem, except for the scale mode *scaleMode* and the void pointer *scale*. The first is used to choose whether a predefined scaling (standard or no scaling) should be employed or the user provides his own scaling procedures. The latter may contain data of the scaling process.

For stiff IVP's, the (implicit or semiimplicit) methods need the Jacobian and associated functions. Due to the flexibility claim, the user may either provide the derivative *df* of the ODE's left hand side as described above, or specify his own procedures to handle the Jacobian.

For that purpose, there are there are the three function pointers *Jacobian*, *NewtonJac* and *Solve*. The first one forms the pure evaluation of the Jacobian $A = f_y(y,t)$ to be stored somewhere for further computations. The procedure *NewtonJac(h)* has to compute the Jacobian $J = I - hA$ (without destroying the contents of A). Often, this procedure will also hold the decomposition of the J. Finally, *Solve(b,x)* has to solve the linear system $Jx = b$. If the user gives the derivative *df*, these procedures are predefined using a full matrix to save the Jacobian A and LR decomposition to solve the linear system $Jx = b$.

For the additional stepsize restrictions (see section 2.3), we need in addition the function *LogarithmicNorm(x)* calculation the estimate $\mu(x) = \langle x, Ax\rangle / \langle x, x\rangle$ of the logarithmic norm of the Jacobian $A = f_y(y,t)$ with respect to some (scaled) product $\langle \cdot, \cdot \rangle$. Furthermore, the function *MonotonicityCoefficient* has to compute the monotonicity coefficient

$$\theta(\delta_{\text{first}}, \delta) := \|\delta - J^{-1}\delta_{\text{first}}\| / \|\delta_{\text{first}}\| \ ,$$

where $J = I - hA$ is again the Jacobian of the implicit discretization. Obviously, the procedures for stiff problems depend somehow on the particular discretization scheme. In the present form they are suitable for the semi-implicit Euler discretization (EULSIM) or the semiimplicit mid-point rule (METAN, not yet included in the package).

3.9 MODULE IVPSCALE

Since we want to use the standard scaling strategy in different contexts without copying parts of code, we introduced a structure *IvpScale*.

```
typedef struct {
    RealVec yScale, yMax, yScaleLast, yScaleMin;
    Real scalMinRel, scalMinAbs;
    IvpType ivpType;
    Int nMin, nMax;
} IvpScale;
```

It contains all data to scale the components $[nMin \ldots nMax]$ of a solution vector according to the standard scaling mechanism as described in section 2.4. To employ this scaling facility, the user has to generate an appropriate IvpScale structure using the allocation procedure

```
extern IvpScale *NewIvpScale(Int nMin, Int nMax, IvpType ivpType);
```

Once generated, this structure has to be given to the scaling procedures

```
extern void IvpInitScale(IvpScale *, RealVec yStart, RealVec yUser-
Scale, Real tol);

extern void IvpIntermediateScale(IvpScale *ivpScale, RealVec y);

extern void IvpRescale(IvpScale *ivpScale, RealVec y);
```

3.10 MODULE VARIATION

As already mentioned, we are also interested in parameter dependent ODE's. Therefore, we combined the construction of the variational equation of a given ODE with the parameter derivative (see section 2.5). Since we wanted to access the partial derivatives $\partial y(t)/\partial \lambda_j$ as vectors, we do not compute the matrix $P(t) = \partial y(t)/\partial \lambda \in \mathrm{Mat}_{q,n}(\mathbf{R})$ but its transpose which is nonetheless denoted by P in the code. (We could also say that we store the matrix as a

vector of pointers to the columns instead to the rows.) A variational problem, including the parameter derivative is necessary, is described by the following structure:

```
typedef struct VARIATIONPROBLEM {
    InitialValueProblem *ivp, *ivpVar;
    Int q;
    Deriv dfPar;
    RealVec lambdaScale;
    void *hiddenData;
} VariationProblem;
```

In this structure we used a void pointer *hiddenData* to hide the data which are only used inside the module. For the structure used to store these data is not known but inside the module, the components of the hiddenData structure are not accessible by the user.

The variational problem for a given initial value problem InitialValueProblem *ivp is created by simply calling *NewVariationProblem*.

```
VariationProblem *varProblem = NewVariationProblem(ivp, q);
```

where q is the number of parameters, i.e., $q = 0$ for the true variation problem. Once the variational problem has been created, an arbitrary integrator (eulex, eulsim, difex) may be used to solve the corresponding initial value problem (which is, of course, of dimension $n + n^2 + qn$) by calling the procedure

```
extern Bool SolveVariationProblem(VariationProblem *, IvpSolver,
    RealMat W, RealMat P);
```

where $W \in \mathrm{Mat}_n(\mathbf{R})$ and $P \in \mathrm{Mat}_{q,n}$ will contain the Wronskian and the parameter derivative (if $q > 0$), respectively.

3.11 Modules eulex, eulsim, difex and trapex

These module presents perhaps the simplest part of the whole package. In fact, they mainly define the function BasicIntegrator which realizes k steps of the particular discretization schemes. The exported procedures

```
extern Bool Eulex(InitialValueProblem *ivp, RealVec y);
```

```
extern Bool Eulsim(InitialValueProblem *ivp, RealVec y);

extern Bool Difex(InitialValueProblem *ivp, RealVec y);

extern Bool Trapex(InitialValueProblem *ivp, RealVec y);
```

create an extrapolation problem and call the extrapolation integrator intex. For the semiimplicit Euler discretization (module eulsim) we have in addtition to define the additional stepsize restriction as described in section 2.3. Until now, we also set the default Jacobian and solver in this module (since it is the only one for stiff ODE's).

4 EXAMPLES

In this section we would like to give two examples, a non-stiff and a stiff initial value problem. They are not intended to show the numerical performance of the extrapolation codes, which is already demonstrated in the literature, but to illustrate how to use the programs. Moreover, we only give examples for the canonical usage of the package, without user defined solvers or scaling. The interested reader may take the module variation as an example which uses a lot of these features.

4.1 A NON-STIFF EXAMPLE

As a non-stiff example we take the two dimensional non-autonomous equation $\dot{y} = f(y,t)$, where the right hand side f is defined as

$$f(y,t) := \begin{pmatrix} y_2 \\ \sqrt{1 + y_2^2/(25-t)} \end{pmatrix}.$$

Realized in C, we get the following procedure.

```
static void f(RealVec y, Real t, RealVec fy) {
    fy[1] = y[2];
    fy[2] = sqrt(1+y[2]*y[2]) / (25-t);
}
```

To use EULEX, the following header files have to be included.

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>

#include "mystd/mystd.h"
#include "mystd/message.h"
#include "MatVec/MatVec.h"
#include "ivpSolve/ivp.h"
#include "ivpSolve/eulex/eulex.h"
```

To integrate the non-stiff ODE from $t_{\text{start}} = 0$ to $t_{\text{end}} = 20$ with the initial value $y_{\text{start}} = (0,0)$, we simply have to create the corresponding initial value problem and start the non-stiff integrator, here EULEX. In the following program we use for the initial value and the solution the same vector y.

```
void main() {
    Int n = 2;
    RealVec y = NewRealVec(1, n);
    InitialValueProblem *ivp = NewInitialValueProblem();

    y[1] = y[2] = 0;

    ivp->f  = f;
    ivp->yStart = y;
    ivp->tStart = 0;
    ivp->tEnd  = 20;
```

```
    ivp->tol  = 1e-5;
    ivp->hStart = ivp->tol;
    ivp->n  = n;

    if (Eulex(ivp, y)) {
        Message("solution y = %s", PrintRealVec("%lg", y, 1, n));
    }
    else {
        Warning("Eulex failed");
    }
    FreeRealVec(y, 1, n);
    FreeInitialValueProblem(ivp);
}
```

4.2 A Stiff Example

As an example for a stiff initial value problem, we take an autonomous chemical oscillator of dimension $n = 5$. The right hand is given by

$$f(y) = \begin{pmatrix} 100 - y_1 - 2000\,y_1 y_4 + 100(1 - y_4 - y_5) \\ y_1 - y_2 \\ y_2 - y_3 - 100\,y_3(1 - y_4 - y_5) + 2600\,y_5 \\ -2000\,y_1 y_4 + 100(1 - y_4 - y_5) + 600\,y_5 \\ 100\,y_3(1 - y_4 - y_5) - 2600\,y_5 \end{pmatrix}$$

The function f is realized as a C procedure as in the non-stiff case. Similarly, its Jacobian df corresponds to the following C procedure.

```
static void df(RealVec y, Real t, RealMat dfy) {
    dfy[1][1] = -1.0 -2000.0*y[4];
    dfy[1][2] = 0.0;
    dfy[1][3] = 0.0;
    dfy[1][4] = - 2000.0*y[1] -100.0;
    dfy[1][5] = -100.0;

    dfy[2][1] = 1.0;

                    ⋮       ⋮
```

```
    dfy[5][5] =-2600.0 -100.0*y[3];
}
```

To use the stiff integrator EULSIM, we don't need to include eulex.h but eulsim.h instead.

```
#include "ivpSolve/eulsim/eulsim.h"
```

As for the non-stiff problem, we have to set the components of an initial value problem and call the stiff integrator.

```
void main() {
    Int n = 5;
    InitialValueProblem *ivp = NewInitialValueProblem();
    RealVec y = NewRealVec(1, n);

    y[1] = 8.99293;
    y[2] = 7.1579;
    y[3] = 5.184;
    y[4] = 0.0100777;
    y[5] = 0.164548;

    ivp->n  = n;
    ivp->f  = f;
    ivp->df  = df;
    ivp->yStart = y;
    ivp->tStart = 0;
    ivp->tEnd = 3.02335;
    ivp->tol  = 1e-8;
    ivp->hStart = 0.001;

    if (Eulsim(ivp, y)) {
        Message("solution = %s", PrintRealVec("%f", y, 1, n));
    }
    else {
        Warning("Eulsim failed");
    }
    FreeRealVec(y, 1, n);
    FreeInitialValueProblem(ivp);
}
```

ACKNOWLEDGEMENT

We are pleased to thank U. Nowak for his support. He put us on our way by extracting the details of the algorithms from the FORTRAN codes.

References

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

[2] P. Deuflhard. Numerik von Anfangswertmethoden für gewöhnliche Differentialgleichungen. Technical Report TR 89–2, Konrad-Zuse-Zentrum, Berlin, 1989.

[3] 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.

Veröffentlichungen des Konrad-Zuse-Zentrum für Informationstechnik Berlin
Technical Reports **September 1992**

TR 91- 1. F. Bornemann; B. Erdmann; R. Roitzsch. *KASKADE - Numerical Experiments.*

TR 91- 2. J. Lügger; W. Dalitz. *Verteilung mathematischer Software mittels elektronischer Netze: Die elektronische Softwarebibliothek eLib.*

TR 91- 3. S. W. C. Noelle. *On the Limits of Operator Splitting: Numerical Experiments for the Complex Burgers Equation.*

TR 91- 4. J. Lang. *An Adaptive Finite Element Method for Convection-Diffusion Problems by Interpolation Techniques.*

TR 91- 5. J. Gottschewski. *Supercomputing During the German Reunification.*

TR 91- 6. K. Schöffel.*Computational Chemistry Software for CRAY X-MP/24 at Konrad-Zuse-Zentrum für Informationtstechnik Berlin.*

TR 91- 7. F. A. Bornemann.*An Adaptive Multilevel Approach to Parabolic Equations in Two Space Dimensions.*

TR 91- 8. H. Gajewski; P. Deuflhard; P. A. Markowich (eds.). *Tagung NUMSIM '91 _5.-8. Mai 1991_ Collected Abstracts and Papers.*

TR 91- 9. P. Deuflhard; U. Nowak; U. Pöhle; B. Ch. Schmidt; J. Weyer. *Die Ausbreitung von HIV/AIDS in Ballungsgebieten.*

TR 91-10. U. Nowak; L. Weimann. *A Family of Newton Codes for Systems of Highly Nonlinear Equations.*

TR 92- 1. K. Schöffel. *Ab initio Quantum Chemical Calculations with GAMESS-UK and GAUSSIAN90 Program Packeges - A Comparison -*

TR 92- 2. K. Schöffel. *Computational Chemistry Software at ZIB.*

TR 92- 3. R. Weismantel. *Plazieren von Zellen: Theorie und Lösung eines quadratischen 0/1-Optimierungsproblems.*

TR 92- 4. A. Martin. *Packen von Steinerbäumen: Polyedrische Studien und Anwendung.*

TR 92- 5. A. Hohmann; C. Wulff. *Modular Design of Extrapolation Codes.*