Efficient Numerical Solution of a Singular Initial Value Problem in Avalanche Modeling

Othmar Koch
Alexander Paul
Andreas Traxler
Ewa Weinmüller

Technical Report ANUM Preprint No. 11/02

Technische Universität Wien
Vienna University of Technology

Institute for Applied Mathematics and Numerical Analysis

Für den Inhalt verantwortlich:
Dr. Othmar Koch, DI Alexander Paul, DI Andreas Traxler und Dr. Ewa B. Weinmüller, Wien.

Verlag:
Institut für Angewandte und Numerische Mathematik (E115), Technische Universität Wien.

Abstract

A leading-edge model due to McClung, [25] and [26], is used to describe the run-up and run-out of avalanches. The modeling process results in a singular initial value problem which is solved numerically using an appropriately adapted version of the IDeC method. In Section 1 the model and its parameters are specified. In Section 2 existence, uniqueness and smoothness properties of the analytical solution of the problem are discussed. In Sections 3 and 4 we describe the algorithm and technical details of the implementation. The results of some numerical experiments are given in Section 5.

Contents

List of Tables

List of Figures

1 The Leading-Edge Model

A leading-edge model (see Figure 1.2) is used to describe avalanche run-up and run-out. It was proposed by McClung ([25], [26]) to improve the so-called center-of-mass models (see Figure 1.1). These older models (e. g. the lumped-mass model proposed by Voellmy, [30]) which were one-dimensional center-of-mass models, often underestimated the run-out and run-up distances. The original idea was to assume the mass of the avalanche to be concentrated in one point. Consequently, it was not possible to take passive snow pressure forces into consideration. However, these aspects are crucial because the dense core of the avalanche usually pushes the edge of the avalanche which results in longer run-out and run-up. With McClung's approach, a more realistic description of the leading edge of the avalanche can be given. Clearly, a set of physical parameters has to be prescribed for the model equations. The important question of how to estimate some of them, incoming speed, friction coefficients, etc., is not addressed in this paper.

1.1 Parameters

The list of parameters including a short description is given in Table 1.1. Where this is possible, the approximate range of the parameters' values determined from observations is also specified.

$\bar{\rho}$	mean density of material in the core of the avalanche	100 – $300 kg/m^3$
g	gravitation constant	$9.81 m/s^2$
$\bar{h}$	mean flow depth along the run-up	$3m$
x	distance along the run-up slope	m
ψ	slope angle (run-up); negative (up-hill)	0 – $\left(-\frac{\pi}{2}\right)$
h_0	flow depth at $t = 0$	2 – $3m$
v_0	incoming speed	15 – $35 m/s$
ψ_0	slope angle	0 – $\frac{\pi}{2}$
μ	dynamic coefficient of friction (bottom)	0.155 – 0.3
ρ_t	mean density of the snow-dust-air mixture	$10 kg/m^3$
C_D	a drag coefficient for turbulent flow over a rough surface	0.01
$v = v(t)$	avalanche speed	m/s
k_p	$k_p = \frac{\cos\psi_0 + \sqrt{\cos^2\psi_0 - \cos^2\phi}}{\cos\psi_0 - \sqrt{\cos^2\psi_0 - \cos^2\phi}}$ if $\phi \geq \psi_0$ $k_p = 1$ otherwise	≥ 1
ϕ	internal friction angle (angle of material debris)	$\frac{5\pi}{36}$

Table 1.1: Parameters

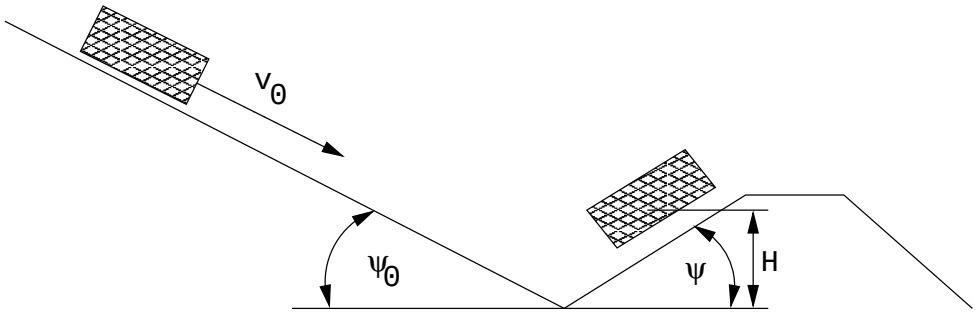

Figure 1.1: Lumped-mass model

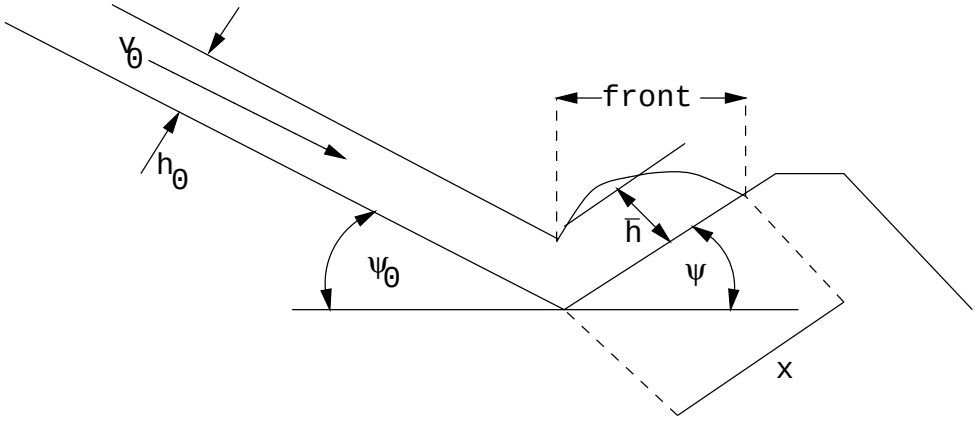

Figure 1.2: Leading-edge model

1.2 The Differential Equation

In [26], five forces are used to describe the avalanche motion. Forces T_1, T_3 and T_4 are resistive terms, while T_2 and T_5 are driving forces. The formulation of the differential equation is based on Newton's second law. The dimension of the differential equation is force per unit width of the flow, i. e. kg/s^2.

Driving force $T_1 = \bar{\rho}\bar{h}xg\sin\psi$:
$\bar{\rho}\bar{h}x$ is the mass of the leading front. T_1 stands for the gravity driving force or resisting force, where ψ is negative if run-up is considered. Note that this force is equal to 0 in the case of flat run-out ($\psi = 0$).

Momentum flux $T_2 = \bar{\rho}h_0v_0^2\cos(\psi_0 - \psi)$:
This is the momentum flux between the main body of the avalanche and the front, calculated as a product of mass and velocity. T_2 can be written as $\underbrace{\bar{\rho}h_0v_0}_{\text{mass flux}} \cdot v_0\cos(\psi_0 - \psi)$. The mass flux is thus calculated as mass × velocity = density × length × velocity. This force vanishes if the incoming slope and the run-up slope are orthogonal.

Dynamic Coulomb resistive force $T_3 = -\mu\bar{\rho}g\bar{h}x\cos\psi$:
This is the frictional resisting force at the base of the leading front. The

parameter μ depends on the roughness of the ground. Naturally, $T_3 = 0$ if the run-up slope is vertical.

Turbulent resistive force $T_4 = -\frac{1}{2}\rho_t C_D v^2 x$:
T_4 is the turbulent resisting drag force (at the top)[1].

Passive snow pressure force $T_5 = \frac{1}{2}\bar{\rho} g h_0^2 k_p \cos\psi_0 \cos(\psi_0 - \psi)$:
T_5 models the fluid thrust between the main body and the front. Not surprisingly, $T_5 = 0$ if the incoming slope is vertical and/or the incoming and outgoing slopes are orthogonal.

Differential Equation Combining the resisting and the driving forces to compute the total force acting on the leading front yields

$$\frac{\text{force}}{\text{unit length}} = \frac{1}{\text{unit length}} \cdot \frac{\partial \text{momentum}}{\partial \text{time}} = \frac{d(\bar{\rho}\bar{h} v x)}{dt} = T_1 + T_2 + T_3 + T_4 + T_5. \tag{1.1}$$

The growth of the front must also satisfy the continuity equation

$$\bar{h} x = h_0 v_0 t, \tag{1.2}$$

which simply states that the total material in the run-up zone at any moment (i.e. $\bar{h}x$) is equal to the influx from the left side of the slope (i.e. $h_0 v_0 t$). Rearranging equation (1.1) and using (1.2) yields

$$v'(t) = -\frac{v(t)}{t} - D_0 v^2(t) + \frac{V}{t} - G_0, \quad t > 0, \tag{1.3a}$$

$$v(0) = v_0, \tag{1.3b}$$

where

$$G_0 = g(\mu \cos\psi - \sin\psi), \tag{1.4}$$

$$V = v_0 \cos(\psi_0 - \psi)\left(1 + \frac{k_p g h_0 \cos\psi_0}{2 v_0^2}\right), \tag{1.5}$$

$$D_0 = \frac{\rho_t C_D}{2\bar{\rho}\bar{h}}. \tag{1.6}$$

Note that only for $v_0 = V$ the problem is well-posed. We will discuss this crucial point in full detail in Section 2. The numerical experiments in Section 5 suggest that the behavior of the *numerical* solution is fairly stable with respect to perturbations in v_0 *even in the case when the analytical problem is not well-posed*. It will become clear in Section 5 in what sense this stability can be

[1] Note that the length x is missing in McClung's paper [26].

observed. For typical input parameters determined from observations (cf. Table 1.1), we have $G_0 \in [1.55, 10]$, $V \in (0, 35.4]$ and $D_0 \in [5.5 \cdot 10^{-5}, 1.6 \cdot 10^{-4}]$.

Our aim is to compute the run-up of the avalanche whose dynamics are governed by the initial value problem (1.3). This means that we have to compute the velocity $v(t)$ from (1.3), determine the (unique) positive root $t^* > 0$, $v(t^*) = 0$, and integrate v up to this point,

$$\int_0^{t^*} v(t)\,dt =: X_R = \text{run-up distance.}$$

The value of D_0 influences the nonlinear part of the differential equation. Due to the fact that for real-life situations $D_0 \ll 1$ holds, this influence is not very strong. For $D_0 = 0$ the unique continuous solution of the resulting linear problem is[2]

$$v(t) = -\frac{1}{2}G_0 t + V, \tag{1.7}$$

which yields

$$t^* = \frac{2V}{G_0}, \tag{1.8}$$

and a run-up distance

$$\int_0^{\frac{2V}{G_0}} \left(-\frac{1}{2}G_0 t + V\right) dt = \frac{V^2}{G_0} = X_R. \tag{1.9}$$

In the study of the convergence behavior of the numerical method, we used different values of D_0 (sometimes unrealistically large), see Section 5 for details.

[2] Note that $v(0) = V$.

2 Analytical Results for the Singular Initial Value Problem

We now investigate the existence, uniqueness and smoothness of the analytical solution of the problem

$$v'(t) = -\frac{v(t)}{t} - D_0 v^2(t) + \frac{V}{t} - G_0, \quad t > 0, \tag{2.1a}$$

$$v(0) = v_0. \tag{2.1b}$$

This analysis heavily relies on techniques developed for the following class of singular initial value problems:

$$z'(t) = \frac{M(t)}{t} z(t) + f(t, z(t)), \quad t \in (0, 1], \tag{2.2a}$$

$$B_0 z(0) = \beta, \tag{2.2b}$$

$$z \in C[0, 1], \tag{2.2c}$$

where $z \in \mathbb{R}^n$, $B_0 \in \mathbb{R}^{r \times n}$, $M \in C^1[0, 1]$ and f is a *smooth* vector-valued function. This problem class was analyzed in [19], [20] and [21]. It turns out that the condition (2.2c) is equivalent to a set of $n - r$ linearly independent linear equations which $z(0)$ has to satisfy in order to be a *continuous* solution of the problem. Here, r is the dimension of the kernel of the matrix $M(0)$. These conditions are augmented by (2.2b) with suitably chosen B_0 to yield a *unique* solution.

We first derive a representation of the general solution of (2.1a). The variation of constants formula yields

$$\begin{aligned} v(t) &= \frac{c}{t} + \frac{1}{t} \int_1^t \tau \left(-D_0 v^2(\tau) + \frac{V}{\tau} - G_0 \right) d\tau \\ &= \frac{\tilde{c}}{t} + V - \frac{G_0}{2} t - t D_0 \int_0^1 s v^2(st)\, ds, \end{aligned}$$

cf. Theorem 3.15 from [19]. Consequently, a continuous solution satisfies

$$v(t) = V - \frac{G_0}{2} t - t D_0 \int_0^1 s v^2(st)\, ds =: (\mathcal{K}\mathcal{F} v)(t) + \varphi(t). \tag{2.3}$$

This means that every continuous solution of (2.1) satisfies $v(0) = V$ which is the only admissible initial condition of the form (2.1b).

Let us consider a domain $G := \{x \in \mathbb{R} : |x| \leq R\}$, where $R > V$. The function $x \mapsto x^2$ satisfies a Lipschitz condition on G and thus the operator $\mathcal{K}\mathcal{F}$ is a contraction on the space $C[0, \delta]$ provided that $\delta > 0$ is sufficiently small[3]. Consequently, it follows from the Banach Fixed Point Theorem (see for example [36, Theorem 2.2.4]) that (2.3) and equivalently (2.1) has a unique continuous solution on $[0, \delta]$. Since for $t > \delta$ the problem is regular, classical theory yields

[3] For a more general result of this type, see [20, Theorem 6] or [19, Lemma 3.2].

the existence of a solution for all $t > 0$, see [36, Theorem 2.3.4]. By substituting (2.3) into (2.1a) we conclude that this solution satisfies $v \in C^\infty[0, \infty)$.

We now show that the solution $v(t)$ is bounded from above[4]. Suppose that $\exists t > 0 : v(t) \geq V$ and $v'(t) \geq 0$. Then

$$\begin{aligned} 0 &\leq -\frac{v(t)}{t} - D_0 v^2(t) + \frac{V}{t} - G_0 \\ &\leq -\frac{V}{t} - D_0 V^2 + \frac{V}{t} - G_0 \\ &= -(D_0 V^2 + G_0) \\ &< 0. \end{aligned}$$

This is a contradiction, and therefore

$$v(t) \leq V, \quad t > 0$$

holds. Finally, we show that the function $v(t)$ is monotonously decreasing and a unique positive root t^* of $v(t)$ exists. We also provide an upper bound for this root. Consider $v(t)$ to be nonnegative. Then,

$$v'(t) \leq -\frac{v(t)}{t} + \frac{V}{t} - G_0$$

and

$$v'(t) = -\frac{v(t)}{t} + \frac{V}{t} - G_0$$

is the linear version of (2.1a) solved by $v(t) = -\frac{1}{2}G_0 t + V$, see (1.7). This means that the solution of (2.1) is bounded from above by this linear polynomial and

$$v'(t) \leq -\frac{G_0}{2}, \quad t \geq 0.$$

Moreover, we have the following bounds:

$$\begin{aligned} t^* &\leq \frac{2V}{G_0}, \\ \int_0^{t^*} v(t)\, dt &\leq \frac{V^2}{G_0}. \end{aligned}$$

Finally, we can conclude that t^* is the only positive root of $v(t)$.

We recapitulate the main result of the above discussion: For $v_0 = V$ the problem (2.1) has a unique, strictly monotonously decreasing smooth solution for $t \in [0, t^*]$, where t^* is a positive root of $v(t)$. If $v_0 \neq V$, (2.1) is not well-posed.

[4]Since we are only interested in the solution up to its positive root t^*, this means that the solution is bounded on the relevant interval.

3 The Numerical Algorithm

3.1 The IDeC Method

For the numerical solution of the initial value problem (2.1) we use the *Iterated Defect Correction (IDeC) method* based on the implicit Euler method. This acceleration technique was analyzed by Frank, see for example [4], and is based on an idea due to Zadunaisky. It was originally proposed for the estimation of the global error of Runge-Kutta methods applied to solve initial value problems of ODEs, see [37]. A comprehensive analysis of various aspects of IDeC for regular problems, asymptotic behavior with respect to the discretization parameter h, convergence towards a fixed point, local and global connection strategies, can be found in [3], [4] –[12], [27] or [28]. For a class of singular initial value problems, see (2.2), the method came into focus in [20] and was investigated experimentally in [1]. The proof of the convergence properties of IDeC in the context of singular problems was given in [24]. The model investigations in [23] and [2] strongly suggest that in case of the singularity, the implicit Euler method is the best candidate among low order methods to serve as a basis for the IDeC iteration.

We now discuss the most important features of the Iterated Defect Correction method. We consider two possible variants of the procedure, the so-called *local* and the *global connection strategy*. While for regular problems these two alternative realizations of IDeC perform similarly this is not the case for the singular problem (2.1). Here, one can observe an order reduction for the local connection strategy, cf. Section 5. A possible reason for such an order reduction may be the following. In the setting of the local connection strategy, IDeC may be viewed as a one-step method proceeding from interval J_i to J_{i+1} (see below for notation). It is well known that for singular problems high-order one-step methods may show order reductions, see for example [17]. A strikingly similar order reduction was also encountered for a certain variant of multiple shooting, cf. [1, pp. 179 sqq.].

We consider initial value problems of the form[5]

$$z'(t) = F(t, z(t)), \quad t \in [a, b], \tag{3.1a}$$

$$z(a) = \beta. \tag{3.1b}$$

We assume that we know the approximate solution, $z_h^{[0]} := z_h = (z_0, \ldots, z_N)$, obtained by the implicit Euler rule on an equidistant grid $\Delta_h := (t_0 = a, t_1, \ldots, t_N = b)$, $t_i = a + i(b-a)/N = a + ih$, $i = 0, \ldots, N$, and denote by $p^{[0]}(t)$ the polynomial of degree N interpolating the values of $z_h^{[0]}$,

$$p^{[0]}(t_i) = z_i^{[0]}, \quad i = 0, \ldots, N.$$

Using this polynomial we construct a neighboring problem associated with (3.1)

[5] For problem (2.1), we may think of $F(t, z(t))$ as being the right-hand side of (2.1a).

and solved by $p^{[0]}(t)$,

$$z'(t) = F(t, z(t)) + d^{[0]}(t), \quad t \in [a, b], \tag{3.2a}$$
$$z(a) = p^{[0]}(a) = \beta, \tag{3.2b}$$

where

$$d^{[0]}(t) := \left(p^{[0]}\right)'(t) - F(t, p^{[0]}(t)).$$

We now solve (3.2) by the same numerical method (implicit Euler rule) and obtain an approximate solution $p_h^{[0]}$ for $p^{[0]}(t)$. This means that for the solution of the neighboring problem (3.2) we know the global error which we can use to estimate the unknown error of the original problem (3.1),

$$\varepsilon_h = R_h(z) - z_h \approx \delta_h^{[0]} := R_h\big(p^{[0]}\big) - p_h^{[0]} = z_h^{[0]} - p_h^{[0]}. \tag{3.3}$$

Here and in the following, $R_h(z) := (z(t_0), \ldots, z(t_N))$ is a projection from the space of continuous functions on $[a, b]$ onto the space of grid vectors.

Zadunaisky provided the following heuristic argument for his method to work: If the values z_h are good approximations for the values of the solution $R_h(z)$ at the grid points, then the polynomial $p^{[0]}(t)$ is a good approximation for the solution $z(t)$ itself. Consequently, the defect $d^{[0]}(t)$ is small and hence the neighboring problem (3.2) and the original problem (3.1) are closely related. This implies that the global error of the solution of (3.2) is closely related to the global error of the solution of (3.1), and therefore the estimate (3.3) shall provide some dependable information about its size.

Having the estimate for the global error of the solution $z_h^{[0]}$ we are able to improve this solution by setting

$$z_h^{[1]} := z_h^{[0]} + \delta_h^{[0]} = z_h^{[0]} + \left(R_h\big(p^{[0]}\big) - p_h^{[0]}\right).$$

We use these values to define a new interpolating polynomial $p^{[1]}(t)$ by requiring $p^{[1]}(t_i) = z_i^{[1]}$, $i = 0, \ldots, N$, and the associated defect reads:

$$d^{[1]}(t) := \left(p^{[1]}\right)'(t) - F(t, p^{[1]}(t)).$$

Clearly, the next neighboring problem is

$$z'(t) = F(t, z(t)) + d^{[1]}(t), \quad t \in [a, b], \tag{3.4a}$$
$$z(a) = \beta, \tag{3.4b}$$

and we solve it by the Euler method to obtain the approximation $p_h^{[1]}$ which is used to correct the basic solution again,

$$z_h^{[2]} := z_h^{[0]} + \delta_h^{[1]} = z_h^{[0]} + \left(R_h\big(p^{[1]}\big) - p_h^{[1]}\right).$$

The procedure can be continued in the above manner.

For obvious reasons one does not use one interpolating polynomial for the whole interval in practice. Instead, a continuous piecewise polynomial function, composed of polynomials of (moderate) degree m defined on subintervals $J_i := [\tau_i, \tau_{i+1}],\ i = 0, \ldots, N_1 - 1,\ \tau_i = t_{i \cdot m} \in \Delta_h$, of the integration interval $[a, b]$ is used to specify the neighboring problem. Here, $N = N_1 m$.

For the *global connection strategy*, the algorithm is realized in the way described above where only one polynomial was used: The problem is solved on $[a, b]$ using the implicit Euler method, and the piecewise polynomial function is used to interpolate the solution values[6]. The iteration is then performed simultaneously on $[a, b]$. For problems with sufficiently smooth data this procedure successively improves the order of convergence of the iterates until a theoretical maximum defined by the degree m of the polynomials used for the interpolation is reached. This result, formulated in the following theorem, holds for regular problems (see [7]) as well as for the class of singular problems discussed in [24]:

Theorem: *Consider the IDeC method based on the implicit Euler rule and on the piecewise interpolation with polynomials of degree m for the numerical solution of problem* (2.2)*. For the approximations obtained in the course of the iteration,*

$$\|z_h^{[j]} - R_h(z)\|_h := \max_{0 \le l \le N} |z_l^{[j]} - z(t_l)| = O(h^{j+1}) \tag{3.5}$$

holds for $j = 0, \ldots, m-1$, provided that f and M are sufficiently smooth. Further iteration does not increase the asymptotic order of the approximation in general.

Note that this result also means that the correction used in the last step of the iteration, $\delta_h^{[m-2]}$, is an asymptotically correct estimate for the global error of the one-but-last iterate $z_h^{[m-2]}$. In practice, however, the more accurate solution $z_h^{[m-1]}$ is considered to be the final result and $\delta_h^{[m-2]}$ is accepted as a reasonable, but pessimistic estimate for the error of this solution.

The *local connection strategy* takes advantage of the fact that for an initial value problem, the final IDeC solution can be computed successively on each subinterval J_i. We start by solving (3.1) on the first interval $J_0 = [a, \tau_1]$. Then we use a polynomial of degree m for the definition of the neighboring problem and carry out all iteration steps of the IDeC procedure on this interval. After we have obtained a solution with the desired level of accuracy, we use its value at the point τ_1 as initial value to start the same procedure on the next interval $J_1 = [\tau_1, \tau_2]$. Continuing this process, we eventually obtain a solution on the whole interval $[a, b]$.

For reasons made clear in the next sections, we implemented both variants of the IDeC iteration, to use them where appropriate. In both cases we chose $m = 4$.

Remark: The proof of the properties of the numerical solution of (2.1) is very similar to the analysis of the above algorithm when it is used to solve (2.2). From the representation of the solution obtained by the implicit Euler method

[6] Note that the number N_1 of "Zadunaisky-intervals" J_i depends on the stepsize h.

we have[7],

$$\begin{aligned} |v_i| &= \left| \prod_{l=1}^{i} \left(1+\frac{h}{t_l}\right)^{-1} v_0 + \sum_{l=1}^{i} \prod_{k=l}^{i} \left(1+\frac{h}{t_k}\right)^{-1} h t_l^{-1} \left(V - t_l D_0 v_l^2 - t_l G_0\right) \right| \\ &\leq \left| \prod_{l=1}^{i} \left(1+\frac{h}{t_l}\right)^{-1} v_0 + \sum_{l=1}^{i} \prod_{k=l}^{i} \left(1+\frac{h}{t_k}\right)^{-1} h t_l^{-1} \left(V - t_l G_0\right) \right| \\ &\leq \text{const}(V + \delta G_0), \quad i = 1, \ldots, N, \end{aligned}$$

which implies that the numerical solution v_h is bounded independently of h. Now the theory from [18] and [20] applies, since the existence of a bounded discrete solution and the existence and smoothness of the solution of the analytical problem are the only crucial assumptions for the analysis to hold. Note that in particular, no additional modifications are necessary due to the unsmooth term $\frac{V}{t}$. The same holds for the arguments given in [24] for the IDeC with global connection strategy.

The local connection strategy for (2.2) was not discussed in [24] and [1]. Its use in the context of the numerical treatment of (2.1) was mainly motivated by the search for the root of $v(t)$, described in the next section. Since the numerical results seem unambiguous, we do not attempt theoretical investigations here either.

3.2 Computation of the Root of $v(t)$

Due to the fact that the interval of integration for the IVP (2.1), more precisely its end-point, is initially not known, the IDeC-Algorithm is applied successively on "Zadunaisky intervals" of the length $mh = 4h$ using the local connection strategy until the root of $v(t)$ is found. For each of these subintervals the basic solution is computed first, using the implicit Euler method (to provide solution values at 5 equidistantly distributed grid points), and the IDeC method with $m = 4$ is used afterwards to improve the solution. This procedure is continued until a negative solution value is encountered in the most accurate approximation (i. e. $v_i^{[3]} \cdot v_{i+1}^{[3]} < 0$). Then the algorithm terminates. To locate the root t^* more precisely, we use a linear interpolant for the points $(t_i, v_i^{[3]})$, $(t_{i+1}, v_{i+1}^{[3]})$ and choose its zero as an approximation for t^*,

$$t_h^* := t_i - \frac{t_{i+1} - t_i}{v_{i+1}^{[3]} - v_i^{[3]}} v_i^{[3]}. \tag{3.6}$$

A reasonable estimate for the error of t_h^* can be determined by using the estimate of the error of the IDeC routine,

$$\varepsilon_1 := \|\delta_h^{[2]}\|_h, \tag{3.7}$$

[7] For details see [18] and [20].

and the first derivative of the function $v(t)$ which can be calculated from the right-hand side of the differential equation (2.1a) on page 7,

$$\Delta t^* := \frac{\varepsilon_1}{|v'(t_h^*)|}. \tag{3.8}$$

3.3 Integration

In order to approximate $\int_0^{t^*} v(t)\,dt$, we apply a numerical quadrature rule to the grid vector $v_h^{[3]}$. Here, two Newton-Cotes formulas with three and five evaluation points within each subinterval J_i are used for integration and error estimation. These formulas are executed on an equidistant grid and read:

$$Q_3 = \frac{h}{3}(f_0 + 4f_1 + f_2), \tag{3.9}$$

$$Q_5 = \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4), \tag{3.10}$$

cf. page 7 for the definition of the grid and [29][8]. The convergence orders of Q_3 and Q_5 are $O(h^4)$ and $O(h^6)$, respectively, provided that f is appropriately smooth. The difference of the approximations calculated with these two methods is used to estimate the error of the integration (see (3.11)). The choice of the above integration formulas was motivated by the fact that each interval J_i contains five points used during the IDeC iteration. This means that the integration formulas Q_5 and Q_3 are carried out once respectively twice per interval. Due to the unknown position of the root of $v(t)$ we do not perform the integration on the last interval that includes the root in the first place. Instead, we restart the initial value problem solver at the endpoint t_j of the last "complete" Zadunaisky interval (the starting value is equal to the numerical approximation obtained previously) and compute the numerical solution on $[t_j, t_h^*]$ using the adjusted step-size $\tilde{h} := \frac{t_h^* - t_j}{m}$. The resulting approximation values are used for the quadrature in a manner described before.

The error estimate for the integral is defined as

$$\varepsilon_0 := |Q_5 - Q_3|. \tag{3.11}$$

3.4 Error Estimation

The estimate of the total error of the algorithm is calculated from the previous estimates (3.7), (3.8) and (3.11),

$$\varepsilon = |t_h^* + \Delta t^*| \cdot \varepsilon_1 + \varepsilon_0. \tag{3.12}$$

We are aware that this estimate is rather conservative.

[8] f_j denotes the evaluation of the integrand in the j-th abscissa of the integration interval.

4 Program Description

The algorithms were implemented in FORTRAN 90. The programs were tested on different computers, an *IBM RS/6000-390* machine and an *SGI Power Challenge XL R10000*, but should be portable to any platform supporting FORTRAN 90. The code does not use any machine specific features, but it has a high demand on the size of memory. The run-time may vary widely with the machine used.

4.1 File Organisation

The entire code is split into five parts. The first one is the main program. The file is called `avalanche.f`. The second file is called `parameters.f` and contains the model parameters. The third part is an interface between the main part and the subroutines performing the different parts of the calculation. This file is called `work.f` and is a module rather than an independent program. Therefore this file cannot be compiled and executed independently. The fourth file is called `ieulidec.f90` and includes the IDeC routines. This module requires the additional files `types.f90`, `dynvect.f90` and `linalg.f90`.

Now we give a detailed description of the files and subroutines. The source of each file is listed in the appendix.

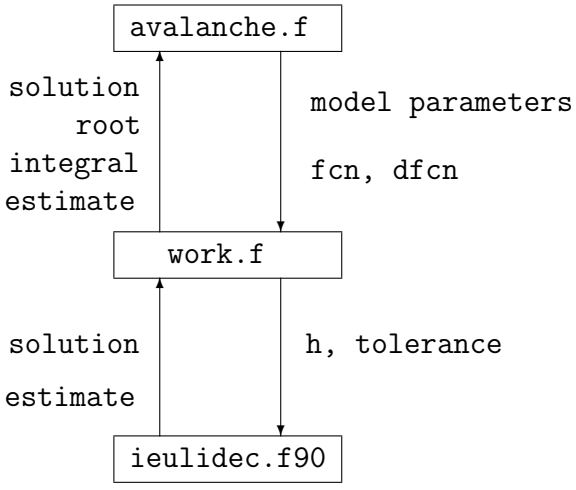

Figure 4.1: Flowchart

Avalanche.f

This file contains the main program. At the beginning it calls the routine that initializes the model parameters. Then the local connection strategy is used to obtain a first approximation of the root (this routine is found in `work.f`). Finally, the global connection strategy is run to compute more accurate results

which are saved to an output file (this refinement can optionally be suppressed to reduce the run-time).

Parameters.f

Here the model parameters described in Section 1.1 are initialized. Because some of them need further calculations we also need an initializing subroutine:

Init The parameters k_p, G_0, V and D_0 are calculated depending on the other parameters. For some tests D_0 was set to a value different from that given in (1.6).

The step-size is also set in this file. The variable `accurate` is used to force the program to refine the results. Valid values are `.false.`, which indicates that no second run using the information about the position of t_h^* from the first run is executed. `.true.` on the other hand forces a second run. This takes as much time as the first run. The difference between the two methods is the following. In the first run, the local connection strategy is used since it is not initially clear where the integration stops. Test results given in Section 5 indicate that this leads to a reduction of the convergence order. This implies that the error estimate for the numerical solution $v_h^{[3]}$ may become unreliable. However, since the position of t_h^* is now known, we can use the global connection strategy for the second run and get a more accurate solution and asymptotically correct error estimate. We recommend that the second alternative be used whenever run-time is not excessively critical.
The last variable is called `filename`. This variable contains a filename, which is used to save the whole solution vector. The output format consists of two columns. The first one is the t value and the second one is the corresponding value of $v_h^{[3]}$. The columns are separated by blanks. This format is used for `gnuplot` for example.

Work.f

In this file the IVP solver is called for the integration on the successive Zadunaisky intervals as described in Section 3.2.

module work This module is an interface between the main program and the working routine of the IDeC solver. It uses the modules `types`, `linalg`, `ieulidec` and `parameters`. The module consists of the subroutine

calc As mentioned before, this routine calls the solver for singular initial value problems for each Zadunaisky interval. After the sign change between two consecutive solution values is detected, the integration using two different *Newton-Cotes formulas* is carried out. Finally, the vector that contains the complete solution of the differential equation is created. The routine returns the error estimate for the solution of the initial value problem, and the error of the integration procedure.

fcn Here we define the right-hand side of the differential equation (see Section 1).

dfcn We also need to specify the Jacobian matrix of f which in our case is a scalar function, since the problem is one-dimensional (the module `ieulidec` is designed to solve problems of arbitrary dimension).

sint This routine performs the integration on the last interval by restarting the IDeC routine with an adapted step size, see §3. The solution values are integrated using Q_3 and Q_5, and the difference between these approximations for the integral value is used as an error estimate for Q_5 and returned to the calling procedure.

Ieulidec.f90

This file contains the solver for IVPs and some auxiliary subroutines. For a detailed description and test results see [1]. Note that the routine we use here is an updated version with slightly adjusted I/O parameters as compared to the version from [1].

Main The file contains one module consisting of only one subroutine:

ivp_ieuleridec This routine calls the routines `fcn` and `dfcn`. A description of `fcn` and `dfcn` can be found in the previous paragraph. The routine first calculates a basic approximation using the implicit Euler method. Then the IDeC iteration (see Section 3.1) is performed. Information on the solution process can be obtained if the logical variable `debug_output` is set to `.true.`.

Linalg.f90

This module contains standard algorithms of linear algebra.

Types.f90

In this module, the type of floating point (and integer) variables is defined. Here, double precision was used. Moreover, the logical parameter `debug_output` determines whether information on the solution process of the initial value problem is displayed on the screen.

Dynvect.f90

This modules defines a new data type used in `ieulidec.f90`.

4.2 Compilation

Since the source code is split into several files the compilation depends on the compiler used. One example for compiling is the command `f90 types.f90 dynvect.f90 linalg.f90 ieulidec.f90 parameters.f work.f avalanche.f`. Note that some compilers require the option `-freeform` in order to interpret the code correctly.

4.3 How to Run the Program

Usually, the compiler sends the output to the file `a.out` (on UNIX systems), but it can be redirected to any other file with the option `-o filename`. This file is executable and can be started by the command `a.out` if the permissions are set correctly.

4.4 Parameters

The parameters of the model as well as the step-size are set in the `parameters` module.

4.5 How to Obtain the Source Code

The source code of the program described above is freely available from either

`http://fsmat.at/~othmar/software.html`

or

`http://math.tuwien.ac.at/~ewa`

This documentation as well as the extensive description of `ieulidec` given in [1] are available from the same homepages

`http://fsmat.at/~othmar/research.html`

`http://math.tuwien.ac.at/~ewa`

together with some of the technical reports and papers we referred to in this report.

5 Test Problems

For the test runs we chose the parameters

$\bar{\rho}$	$200kg/m^3$
g	$9.81m/s^2$
h	$3m$
ψ	$-\pi/6$
h_0	$2m$
v_0	V
ψ_0	$\pi/20$
μ	0.155
ρ_t	$10kg/m^3$
C_D	0.01
k_p	2.31946578325653
ϕ	0.43633231299847

Table 5.1: Physical parameters for the tests

This results in

G_0	6.22183492772341
V	16.41619116478564
D_0	0.00008333333333

Table 5.2: Parameters in the differential equation

5.1 Correct Initial Value

For the first series of tests we set $v_0 = V$, since only for this initial value the problem is well-posed. Perturbation of the initial value will be examined in Section 5.2. As we are interested in the asymptotic qualities of our numerical method, we examine the solution on the interval $[0, 6]$ and omit the question of finding the root t^* of $v(t)$.

All tests in this section were performed on a Silicon Graphics Power Challenge XL R10000 with operating system IRIX V.4. The FORTRAN 90 programs were compiled with the MIPSPRO FORTRAN 90 COMPILER 7.2.1.1M. The tests were run in IEEE double precision with EPS $\approx 1.11 \cdot 10^{-16}$.

For $D_0 \neq 0$ the exact solution of the IVP cannot be calculated explicitly. Therefore, a solution calculated with a very small step-size (in the subsequent tests we used $h = 2^{-14} \approx 6.10 \cdot 10^{-5}$) is used as a reference for the computation of the absolute error of a numerical solution (on a coarser grid). As mentioned in Section 1, D_0 is varied in the tests. Three different values for D_0 are examined:

- $D_0 = 0$: In this case the exact solution (see equation (1.7) on page 6) of the initial value problem can be calculated explicitly. The numerical results are given in Tables 5.7 and 5.8. For details concerning the global and local connection strategy, refer to the next paragraph.
- $D_0 = \frac{\rho_t C_D}{2\bar{\rho}h}$: This is the realistic value for D_0 given by the leading-edge model. For the test parameters (see Table 5.1) this results in $D_0 = 0.00008333333333$. The typical range for D_0 is $[5.5 \cdot 10^{-5}, 1.6 \cdot 10^{-4}]$. For the convergence results, see Tables 5.5 and 5.6.
- $D_0 = 0.065$: This untypically large value is used in order to observe the empirical convergence orders.

We investigate two different strategies to obtain the numerical solution. The first of these is based on the global connection strategy. This means that the IDeC algorithm described in Section 3.1 is applied on the whole interval. The convergence behavior of the IDeC algorithm (see Table 5.3) is of special interest. The initial step-size for the test problems is 0.5 and is subsequently halved in every step. The errors of the numerical solutions are calculated as the maximum of the difference between the approximate solution and the reference solution over all grid-points (of the coarser grid). The improvement of the solution is compared with the solution associated with the previous step-size. To determine the empirical rate of convergence we relate the results associated with two different step-sizes $h_1,\ h_2$. We assume

$$\varepsilon_1 \approx c h_1^p, \tag{5.1}$$

$$\varepsilon_2 \approx c h_2^p, \tag{5.2}$$

where ε_i is the error of the solution calculated with step-size h_i and c a constant independent of h_i. For decreasing step-sizes we expect p to converge to a nonnegative integer (the empirical convergence order) and c to stabilize at a moderate value. From the above relations we can calculate p and c:

$$p \approx \frac{\ln\left(\frac{\varepsilon_1}{\varepsilon_2}\right)}{\left|\ln\left(\frac{h_1}{h_2}\right)\right|}, \tag{5.3}$$

$$c \approx \frac{\varepsilon_1}{h_1^p}. \tag{5.4}$$

It turns out that for the global connection strategy the convergence order of the approximations for (2.1) is equal to the classical order 4, see Table 5.3. A similar result was also observed in [1] for the class of singular initial value problems discussed there. These experiments suggest that the additional unsmooth term $\frac{V}{t}$ in the right-hand side of the differential equation does not influence the convergence rate. Both quantities p and c stabilize up to the point where the global error has the order of magnitude of the round-off error, whence its behavior becomes unsystematic. This also means that the error estimate is asymptotically correct for the one but last improved solution $v_h^{[2]}$ and can be

used to estimate the error of the final solution. Note that it slightly overestimates this error, but in a quite systematic way. In Table 5.5 no systematic behavior of the error can be observed except for the coarsest step-sizes. The reason is obvious: the error (and its estimate) are of the order of magnitude of the round-off error. This interpretation is also supported by the fact that the error increases slightly for smaller h. Nonetheless, the solution's accuracy is already very satisfactory. The same holds for the results displayed in Table 5.7. Since in this case the exact solution is a linear polynomial (see (1.7)), the initial value problem is solved exactly by the implicit Euler method. Consequently, the deviation from the exact solution which is shown in Table 5.7 is entirely caused by round-off errors. Again, the solution is satisfactory albeit its behavior appears unsystematic.

However, for the local connection strategy an order reduction down to $O(h^2)$ occurs. Thus, the method is not as efficient in this setting and we may expect problems with the error estimate. The order reduction can be observed in both Tables 5.4 and 5.6. Due to the reduced accuracy we observe a systematic behavior of the error, cf. Table 5.6, for smaller step-sizes than this was the case earlier, see Table 5.5. The error estimate procedure always slightly over-estimates the true error. Consequently, we may put some trust in this information even in the case where the order reduction occurs. In Table 5.8, again, only round-off error effects are observed.

5.2 Perturbation of the Initial Value

As discussed in Section 2 the IVP is only well-posed if $v_0 = V$. If the problem is posed with an initial value that is perturbed from that value, however, experiments demonstrate that the numerical solution still shows a very stable behavior. The perturbation of the solution (compared to the reference solution of the problem with $v_0 = V$) is damped towards the end of the interval and the maximal error is always assumed at the first grid point $t_0 = 0$, which means that the error of the solution is never larger than the perturbation. The errors of the solution are displayed in Figure 5.1, where the vertical axis has a logarithmic scaling (base 10). The different curves show the error along the integration interval for the respective perturbations in the initial value. From the rapid damping of the errors it is obvious that the contribution to the integral in whose value we are finally interested is minimal. It seems that we can obtain a solution carrying enough meaningful information even by solving an ill-posed problem, provided that the perturbation is appropriately small.

5.3 Test Runs

We now present the results of program runs for three different typical situations which we discussed in previous sections. The step-size used throughout was $h = 2^{-7}$. All other parameters except for D_0 were set as in Table 5.1. For the first test run, D_0 was manually changed to $D_0 = 0$. In this case we are able to compare the numerical results with the exact values of the solution specified

h	p	c	error	estimate
0.50000E+00	0.45617E+01	0.21030E-01	0.89047E-03	0.46120E-02
0.25000E+00	0.42970E+01	0.17504E-01	0.45300E-04	0.44726E-03
0.12500E+00	0.41497E+01	0.14272E-01	0.25521E-05	0.49353E-04
0.62500E-01	0.40743E+01	0.12200E-01	0.15150E-06	0.58005E-05
0.31250E-01	0.40368E+01	0.10995E-01	0.92304E-08	0.70321E-06
0.15625E-01	0.40176E+01	0.10288E-01	0.56989E-09	0.86570E-07
0.78125E-02	0.39970E+01	0.94430E-02	0.35692E-10	0.10739E-07
0.39062E-02	0.27356E+01	0.20755E-04	0.53588E-11	0.13373E-08
0.19531E-02	0.15506E-01	0.58400E-11	0.53015E-11	0.16679E-09
0.97656E-03	0.38455E-01	0.67388E-11	0.51621E-11	0.20735E-10

Table 5.3: Convergence of the global connection strategy ($D_0 = 0.065$)

h	p	c	error	estimate
0.50000E+00	0.33247E+01	0.64513E-02	0.64389E-03	0.39904E-02
0.25000E+00	0.18209E+01	0.22748E-02	0.18225E-03	0.14202E-02
0.12500E+00	0.18957E+01	0.25235E-02	0.48978E-04	0.37032E-03
0.62500E-01	0.19564E+01	0.28627E-02	0.12621E-04	0.86696E-04
0.31250E-01	0.19615E+01	0.29036E-02	0.32405E-05	0.20653E-04
0.15625E-01	0.19766E+01	0.30602E-02	0.82336E-06	0.50424E-05
0.78125E-02	0.19848E+01	0.31657E-02	0.20802E-06	0.12464E-05
0.39062E-02	0.19904E+01	0.32536E-02	0.52351E-07	0.31007E-06
0.19531E-02	0.19952E+01	0.33411E-02	0.13131E-07	0.77348E-07
0.97656E-03	0.19983E+01	0.34056E-02	0.32867E-08	0.19317E-07
0.48828E-03	0.20043E+01	0.35515E-02	0.81922E-09	0.48271E-08
0.24414E-03	0.20392E+01	0.46323E-02	0.19932E-09	0.12065E-08

Table 5.4: Convergence of the local connection strategy ($D_0 = 0.065$)

in Section 1. The only error source while integrating a linear polynomial, see Table 5.9, are round-off errors. The results shown in Tables 5.10 and 5.11 correspond to a very small D_0 typical in applications. For the exact starting value $v_0 = V$, cf. Table 5.10, the behavior of the solution is almost linear, since the influence of D_0 is negligible. The program provides very accurate results for both the global and the local connection strategy[9]. The global strategy seems to be preferable, however. In the case where v_0 is set to $v_0 = V + 1$, the error increases significantly, but the global connection strategy still seems to yield reasonable results in spite of the fact that the analytical problem is ill-posed. It may be of interest to compare the results in Tables 5.10 and 5.11 with the theoretical bounds from Section 2.

[9]This is probably also the reason why the error of the quadrature procedure is estimated to be equal to 0.

h	p	c	error	estimate
0.50000E+00	0.38603E+01	0.16567E-08	0.11407E-09	0.84012E-09
0.25000E+00	0.23015E+01	0.56231E-09	0.23139E-10	0.10409E-09
0.12500E+00	0.29712E+00	0.34933E-10	0.18832E-10	0.12928E-10
0.62500E-01	0.17180E-01	0.19517E-10	0.18610E-10	0.16280E-11
0.31250E-01	-0.10325E-02	0.18556E-10	0.18623E-10	0.19718E-12
0.15625E-01	0.38583E-02	0.18874E-10	0.18573E-10	0.52847E-13
0.78125E-02	0.72623E-02	0.19143E-10	0.18480E-10	0.18785E-12
0.39062E-02	0.11172E-01	0.19509E-10	0.18337E-10	0.35572E-12
0.19531E-02	0.20479E-01	0.20543E-10	0.18079E-10	0.75895E-12
0.97656E-03	0.40976E-01	0.23345E-10	0.17573E-10	0.13789E-11
0.48828E-03	0.11383E+00	0.38680E-10	0.16239E-10	0.31939E-11
0.24414E-03	0.28519E+00	0.14286E-09	0.13327E-10	0.62768E-11
0.12207E-03	0.48529E+00	0.75470E-09	0.95199E-11	0.11405E-10

Table 5.5: Convergence of the global connection strategy with realistic D_0

h	p	c	error	estimate
0.50000E+00	0.18915E+01	0.10286E-05	0.27722E-06	0.21274E-05
0.25000E+00	0.19482E+01	0.10698E-05	0.71842E-07	0.53173E-06
0.12500E+00	0.19706E+01	0.11035E-05	0.18330E-07	0.13294E-06
0.62500E-01	0.19849E+01	0.11369E-05	0.46308E-08	0.33237E-07
0.31250E-01	0.19921E+01	0.11599E-05	0.11640E-08	0.83285E-08
0.15625E-01	0.19936E+01	0.11660E-05	0.29229E-09	0.20866E-08
0.78125E-02	0.19997E+01	0.11959E-05	0.73086E-10	0.52213E-09
0.39062E-02	0.16008E+01	0.17260E-06	0.24096E-10	0.13069E-09
0.19531E-02	0.20620E-01	0.27015E-10	0.23754E-10	0.32683E-10
0.97656E-03	0.41701E-01	0.30812E-10	0.23078E-10	0.81761E-11
0.48828E-03	0.36949E-01	0.29814E-10	0.22494E-10	0.20681E-11
0.24414E-03	0.27401E+00	0.18173E-09	0.18603E-10	0.65577E-12

Table 5.6: Convergence of the local connection strategy with realistic D_0

h	error	estimate
0.50000E+00	0.15987E-13	0.88818E-14
0.25000E+00	0.14211E-13	0.10658E-13
0.12500E+00	0.30198E-13	0.62172E-14
0.62500E-01	0.40856E-13	0.33307E-13
0.31250E-01	0.83489E-13	0.75495E-14
0.15625E-01	0.23448E-12	0.13545E-12
0.78125E-02	0.24425E-12	0.19318E-12
0.39062E-02	0.29399E-12	0.38503E-12
0.19531E-02	0.79581E-12	0.82645E-12
0.97656E-03	0.10330E-11	0.11662E-11
0.48828E-03	0.25757E-11	0.27800E-11
0.24414E-03	0.49267E-11	0.57616E-11
0.12207E-03	0.89102E-11	0.10177E-10

Table 5.7: Error of the global connection strategy ($D_0 = 0$)

h	error	estimate
0.50000E+00	0.15099E-13	0.82489E-13
0.25000E+00	0.17764E-13	0.56243E-13
0.12500E+00	0.20428E-13	0.93738E-13
0.62500E-01	0.31974E-13	0.14061E-12
0.31250E-01	0.46185E-13	0.17810E-12
0.15625E-01	0.76383E-13	0.16873E-12
0.78125E-02	0.22560E-12	0.18748E-12
0.39062E-02	0.31086E-12	0.22497E-12
0.19531E-02	0.63327E-12	0.24372E-12
0.97656E-03	0.13065E-11	0.29996E-12
0.48828E-03	0.26210E-11	0.31871E-12
0.24414E-03	0.52029E-11	0.26247E-12

Table 5.8: Error of the local connection strategy ($D_0 = 0$)

result	program results		theoretical
	global connection strategy	local connection strategy	
root	5.27696133230186	5.27696133230185	5.27696133230342
error of root	$7.537422920E-14$	$5.119047926E-14$	
integral	43.31380300011892	43.31380300011876	43.31380300013745
error of integral	$3.388131789E-21$	0	
global error	$1.236510911E-12$	$1.592476212E-13$	

Table 5.9: Test run with $D_0 = 0,\ v_0 = V$.

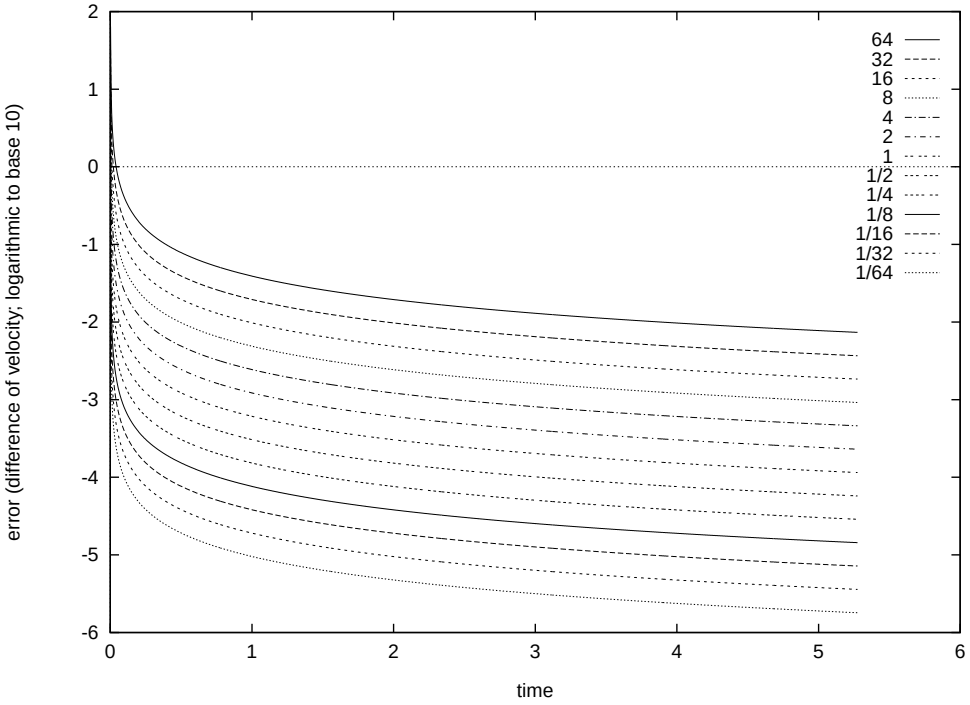

Figure 5.1: Errors for the perturbed IVPs

result	program results	
	global connection strategy	local connection strategy
root	5.27379405258795	5.27379405258823
error of root	$6.684849889E-14$	$2.163802616E-10$
integral	43.25747367208126	43.25747367210393
error of integral	$2.646977960E-22$	0
global error	$1.095998293E-12$	$6.727368662E-10$

Table 5.10: Test run with $D_0 = 0.0000833333333,\ v_0 = V$.

result	program results	
	global connection strategy	local connection strategy
root	5.27382796252824	5.27383104607456
error of root	$8.040849091E-3$	$4.240569340E-2$
integral	43.26327625545632	43.26355028391055
error of integral	$8.716368895E-6$	$8.715021948E-6$
global error	$1.318423119E-1$	$1.318423105E-1$

Table 5.11: Test run with $D_0 = 0.00008333333333,\ v_0 = V + 1$.

References

[1] W. Auzinger, O. Koch, P. Kofler, and E. Weinmüller, *The application of shooting to singular boundary value problems*, Techn. Rep. Nr. 126/99, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technology, Austria, 1999. Available at `http://fsmat.at/~othmar/research.html`.

[2] ——, *Acceleration techniques for singular initial value problems*, Techn. Rep. Nr. 129/00, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technology, Austria, 2000. Will be available soon at `http://fsmat.at/~othmar/research.html`.

[3] W. Auzinger and J. Monnet, *IDeC — Convergence independent of error asymptotics*, BIT, 27 (1987), pp. 350–367.

[4] R. Frank, *The method of Iterated Defect Correction and its application to two-point boundary value problems, Part I*, Numer. Math., 25 (1976), pp. 409–419.

[5] ——, *The method of Iterated Defect Correction and its application to two-point boundary value problems, Part II*, Numer. Math., 27 (1977), pp. 407–420.

[6] R. Frank, F. Macsek, and C. Überhuber, *Iterated Defect Correction for differential equations, Part II: Numerical experiments*, Computing, 33 (1984), pp. 107–129.

[7] R. Frank and C. Überhuber, *Iterated Defect Correction for Runge-Kutta methods*, Techn. Rep. Nr. 14/75, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technolgy, Austria, 1975.

[8] ——, *Collocation and Iterated Defect Correction*, Techn. Rep. Nr. 21/76, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technology, Austria, 1976.

[9] ——, *An extension of the applicability of Iterated Deferred Corrections*, Techn. Rep. Nr. 23/76, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technology, Austria, 1976.

[10] ——, *Iterated Defect Correction for the efficient solution of ordinary differential equations*, Techn. Rep. Nr. 17/76, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technology, Austria, 1976.

[11] ——, *Iterated Defect Correction for the efficient solution of stiff systems of ordinary differential equations*, BIT, 17 (1977), pp. 146–159.

[12] ——, *Iterated Defect Correction for differential equations, Part I: Theoretical results*, Computing, 20 (1978), pp. 207–228.

[13] F. Frommlet and E. Weinmüller, *Asymptotic error expansions for singular boundary value problems*, Math. Models Methods Appl. Sci., 11 (2001), pp. 71–85.

[14] F. d. Hoog and R. Weiss, *Difference methods for boundary value problems with a singularity of the first kind*, SIAM J. Numer. Anal., 13 (1976), pp. 775–813.

[15] ——, *The application of linear multistep methods to singular initial value problems*, Math. Comp., 32 (1977), pp. 676–690.

[16] ——, *Collocation methods for singular boundary value problems*, SIAM J. Numer. Anal., 15 (1978), pp. 198–217.

[17] ——, *The application of Runge-Kutta schemes to singular initial value problems*, Math. Comp., 44 (1985), pp. 93–103.

[18] O. Koch, *Numerische Lösung singulärer Anfangswertprobleme zweiter Ordnung*, Ph.D. Thesis, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technology, Austria, 1998. Available at `http://fsmat.at/~othmar/research.html`.

[19] O. Koch, P. Kofler, and E. Weinmüller, *Analysis of singular initial and terminal value problems*, Techn. Rep. Nr. 125/99, Inst. for Appl. Math. and Numer. Anal., Vienna Univ. of Technology, Austria, 1999. Available at `http://fsmat.at/~othmar/research.html`.

[20] ——, *The implicit Euler method for the numerical solution of singular initial value problems*, Appl. Num. Math., 34 (2000), pp. 231–252.

[21] ——, *Initial value problems for systems of ordinary first and second order differential equations with a singularity of the first kind*, Analysis, 21 (2001), pp. 373–389.

[22] O. Koch and E. Weinmüller, *The convergence of shooting methods for singular boundary value problems.* To appear in Math. Comp. Also available as ANUM Preprint Nr. 9/01 at `http://www.math.tuwien.ac.at/~inst115/preprints.htm`.

[23] ——, *Acceleration techniques for singular initial value problems*, in Problems in Modern Applied Mathematics, N. Mastorakis, ed., WSES Press, 2000, pp. 6–11.

[24] ——, *Iterated Defect Correction for the solution of singular initial value problems*, SIAM J. Numer. Anal., 38 (2001), pp. 1784–1799.

[25] D. M. McClung and O. Hungr, *An equation for calculating snow avalanche run-up against barriers*, in Avalanche Formation, Movement and Effects, IAHS Publ. no. 162, 1987, pp. 605–612.

[26] D. M. McClung and A. I. Mears, *Dry-flowing avalanche run-up and run-out*, J. Glaciol., 41 (1995), pp. 359–369.

[27] K. Schild, *Gaussian collocation via defect correction*, Numer. Math., 58 (1990), pp. 369–386.

[28] H. J. Stetter, *The defect correction principle and discretization methods*, Numer. Math., 29 (1978), pp. 425–443.

[29] J. Stoer, *Numerische Mathematik 1*, Springer-Verlag, Berlin-Heidelberg-New York, 8th ed., 1999.

[30] A. Voellmy, *Über die Zerstörungskraft von Lawinen*, Schweiz. Bauztg., 73 (1955), pp. 159–165, 212–217, 246–249, 280–283.

[31] E. Weinmüller, *A difference method for a singular boundary value problem of second order*, Math. Comp., 42 (1984), pp. 441–464.

[32] ——, *On the boundary value problems for systems of ordinary second order differential equations with a singularity of the first kind*, SIAM J. Math. Anal., 15 (1984), pp. 287–307.

[33] ——, *Collocation for singular boundary value problems of second order*, SIAM J. Numer. Anal., 23 (1986), pp. 1062–1095.

[34] ——, *On the numerical solution of singular boundary value problems of second order by a difference method*, Math. Comp., 46 (1986), pp. 93–117.

[35] ——, *Stability of singular boundary value problems and their discretization by finite differences*, SIAM J. Numer. Anal., 26 (1989), pp. 180–213.

[36] H. Werner and H. Arndt, *Gewöhnliche Differentialgleichungen: eine Einführung in Theorie und Praxis*, Springer-Verlag, Berlin-Heidelberg, 1986.

[37] P. Zadunaisky, *On the estimation of errors propagated in the numerical integration of ODEs*, Numer. Math., 27 (1976), pp. 21–39.

A Program Code

The following subsections contain the program code, split into various files for better readability. For further details about the functionality of each file or subroutine see Section 4.

A.1 Files

A.1.1 Main Program

```
1  PROGRAM avalanche
2
3  !****************
4  !* USEd Modules *
5  !****************
6  USE types
7  USE linalg
8  USE ieulidec
9  USE parameters
10 USE work
11
12 !*******************
13 !* Local variables *
14 !*******************
15 IMPLICIT NONE
16
17 REAL(kind = dp)          :: y(1)                    !initial value
18 REAL(KIND = dp)          :: integral1,integral2     !integral with two different
19                                                     !methods
20 REAL(KIND = dp)          :: slope(1),integral       !auxiliary variables
21 REAL(KIND = dp), POINTER :: solution(:,:),sol(:,:)  !the solution vector
22 REAL(KIND = dp)          :: nstx=0                  !the root of v(t)
23 REAL(KIND = dp)          :: est=0,estimator=HUGE(1.0_dp)       !error estimates
24 REAL(KIND = dp)          :: global_rel=1.0_dp,tol=HUGE(1.0_dp) !tolerances
25 REAL(KIND = dp)          :: estint, estidec                    !auxiliary variables
26 INTEGER(KIND = i4b)      :: ifail,j                 !auxiliary variables
27
28 !*********************************
29 !* INTERFACE for USEd subroutines *
30 !*********************************
31 INTERFACE
32
33 FUNCTION fcn(t, y)
34  USE types, ONLY:dp
35  IMPLICIT NONE
36  REAL (KIND=dp), INTENT(IN) :: t, y(:)
37  REAL (KIND=dp) :: fcn(size(y))
38 END FUNCTION fcn
39
40 FUNCTION dfcn(t, y)
41  USE types, ONLY :dp
42  IMPLICIT NONE
43  REAL (KIND=dp), INTENT(IN):: t, y(:)
44  REAL (KIND=dp) :: dfcn(size(y), size(y))
45 END FUNCTION dfcn
46
47 FUNCTION sint(a,b,y,estint,estidec)
48  USE TYPES
49  USE IEULIDEC
50  IMPLICIT NONE
51  REAL(KIND = dp), INTENT(IN) :: a, b
52  REAL(KIND = dp), INTENT(INOUT):: y(:)
```

```
 REAL(KIND = dp), INTENT(OUT):: estint, estidec
 REAL(KIND = dp) :: sint
END FUNCTION sint
END INTERFACE

!*****************
!* program code  *
!*****************

!Initializations
!---------------
  CALL init
  y(1)=ys

!Display information about model parameters
!------------------------------
  WRITE (*,*) "model parameters:"
  WRITE (*,*) "-----------------"
  WRITE (*,FMT = '(A,f20.14)') "rho_bar                              =",rho_bar
  WRITE (*,FMT = '(A,f20.14)') "g                                    =",g
  WRITE (*,FMT = '(A,f20.14)') "h_bar                                =",h_bar
  WRITE (*,FMT = '(A,f20.14)') "psi                                  =",psi
  WRITE (*,FMT = '(A,f20.14)') "h_null                               =",h_null
  WRITE (*,FMT = '(A,f20.14)') "v_null                               =",v_null
  WRITE (*,FMT = '(A,f20.14)') "psi_null                             =",psi_null
  WRITE (*,FMT = '(A,f20.14)') "mu                                   =",mu
  WRITE (*,FMT = '(A,f20.14)') "rho_t                                =",rho_t
  WRITE (*,FMT = '(A,f20.14)') "C_D                                  =",C_D
  WRITE (*,FMT = '(A,f20.14)') "k_p                                  =",k_p
  WRITE (*,FMT = '(A,f20.14)') "phi                                  =",phi
  WRITE (*,*)
  WRITE (*,FMT = '(A,f20.14)') "G_null                               =",G_null
  WRITE (*,FMT = '(A,f20.14)') "V                                    =",V
  WRITE (*,FMT = '(A,f20.14)') "D_null                               =",D_null

  WRITE (*,*)
  WRITE (*,*) "General settings:"
  WRITE (*,*) "-----------------"

  WRITE (*,FMT='(A,G20.10E2)')  "EPS (maschine accuracy)            =",epsilon(h)
  WRITE (*,FMT = '(A,f20.14)')  "initial step size                 h =",h
  WRITE (*,*)

!program start
!-------------

  integral=0
  WRITE (*,*) "Calculating, please wait ..."
  CALL calc(fcn, y, h, dfcn, sol, nstx, integral, est)  ! calls local connection strategy
  WRITE (*,*) "... finished"                            ! from module work
  WRITE (*,*)
  WRITE (*,*) "The solution is written to file ", filename
  OPEN(unit=1,FILE=filename,ACCESS='sequential')
  DO j = 0, SIZE(sol)-1                                 ! positive values of v(t) are
    WRITE (1,*) j*h/4,"   ",sol(j+1,1)                  ! written to file
    IF (sol(j+2,1)<0) EXIT                              ! note that the last (negative)
  END DO                                                ! value is not included in sol
  CLOSE(1)

!print the results
!-----------------
  WRITE (*,*)
  WRITE (*,*) "The results of the local connection strategy are"
  WRITE (*,*) "------------------------------------------------"
  WRITE (*,FMT = '(A,f20.14)') "The root is at                       :",nstx
  WRITE (*,FMT = '(A,f20.14)') "The integral is                      :",integral
```

```
119   WRITE (*,FMT='(A,G20.10E2)') "The global error estimate is        :",est
120
121 !check for second run with global connection strategy
122 !---------------------------------------------------
123   IF (.not. accurate) stop
124   y(1)=ys
125   WRITE (*,*)
126   WRITE (*,*) "You have chosen to rerun the calculation for higher accuracy."
127   WRITE (*,*) "Calculating, please wait ..."
128   nstx=h*CEILING(nstx/h)
129
130 estimator=tol           ! resetting INOUT parameters for ivp_ieuleridec
131 global_rel=1.0_dp
132 h0=h/4.0_dp             ! length of one Euler step
133
134 ! call of IDeC routine from module ieulidec
135
136   call ivp_ieuleridec(fcn, dfcn, 0._dp, y, nstx, .false., h0, estimator, &
137            global_rel, 10._dp**(-12), 4, ifail, solution=solution, &
138            order_poly=4, use_damping=.true., use_stepcontrol=.false.)
139
140   WRITE (*,*) "  ... finished"
141
142 !Calculate root and integral for improved solution
143 !-------------------------------------------------
144   j=0
145   nstx=0
146   integral1=0
147   integral2=0
148   integral=0
149   DO WHILE ((j<size(solution)) .AND. (nstx .EQ. 0))
150     IF (solution(j,1)*solution(j+1,1) .le. 0) THEN !has the sign changed?
151       nstx = (j*h/4)-solution(j,1)*((h/4)/(solution(j+1,1)-solution(j,1)))
152     END IF
153     IF ((mod(j,4).eq.0) .AND. (j>0)) THEN
154       integral1=integral1+2*h/4*(7*solution(j-4,1)+32*solution(j-3,1)+&
155                 12*solution(j-2,1)+32*solution(j-1,1)+7*solution(j,1))/45
156       integral2=integral2+h/4*(solution(j-4,1)+4*solution(j-3,1)+&
157                 2*solution(j-2,1)+4*solution(j-1,1)+solution(j,1))/3
158     END IF
159     j=j+1
160   END DO
161
162 ! auxiliary integration of the values from the beginning
163 ! of the last interval to the root of v(t)
164
165   integral=sint(h*FLOOR(1._dp*(j-1)/4),nstx,solution(FLOOR(1._dp*j/4)*4,:),&
166                                                          estint,estidec)
167   IF (integral>0._dp) THEN
168     integral=integral+integral1
169   ELSE
170     integral=integral1
171   END IF
172   slope=abs(fcn(nstx,(/0._dp/)))      ! used for error estimation of root
173   est=abs(integral1-integral2)+estint+estimator*h*(j-mod(j,4))/4+&       ! error of X_R
174       estidec*(nstx+abs((estimator+estidec)/slope(1))-h*(j-mod(j,4))/4)
175   WRITE (*,FMT = '(A)') "/-----------------------------------------------------------\"
176   WRITE (*,FMT = '(A)') "|   The final results of the global connection strategy   |"
177   WRITE (*,FMT = '(A)') "|-----------------------------------------------------------|"
178   WRITE (*,FMT = '(A,f20.14,A)')  "| The root is at                    :", nstx, " |"
179   WRITE (*,FMT = '(A,G20.10E2,A)') "| Error estimation for the root     :",&
180                                             abs((estimator+estidec)/slope(1)), " |"
181   WRITE (*,FMT = '(A,f20.14,A)') "| The integral is                   :", integral,&
182                                                                    " |"
183   WRITE (*,FMT = '(A,G20.10E2,A)') "| Error estimate for the integral :",&
184                                               abs(integral1-integral2)+estint, " |"
```

```
185     WRITE (*,FMT = '(A,G20.10E2,A)') "| The global error estimate is      :", est, " |"
186     WRITE (*,FMT = '(A)') "\------------------------------------------------------------/"
187     WRITE (*,*)
188     WRITE (*,*) "The solution is written to file ", filename
189     OPEN(unit=1,FILE=filename,ACCESS='sequential')
190     DO j = 0, SIZE(solution)-1                ! positive values of v(t) are written
191       WRITE (1,*) j*h/4,"   ",solution(j,1)    ! to file
192       IF (solution(j+1,1)<0) EXIT             ! solution contains all points of the
193     END DO                                    ! solution
194     CLOSE(1)
195
196
197   !clear memory
198   !-------------------
199     DEALLOCATE(solution)
200   END PROGRAM AVALANCHE
```

A.1.2 Interface Routine

```
1           MODULE work
2
3           !calculates the solution using the local connection strategy
4           !-----------------------------------------------------------
5
6           !***************
7           !* USEd Modules *
8           !***************
9           USE types
10          USE linalg
11          USE ieulidec
12          USE parameters
13
14          !*************
15          !* Variables *
16          !*************
17          IMPLICIT NONE
18
19          CONTAINS
20
21          SUBROUTINE calc(fcn, y, h, dfcn, solution, nstx, integral, est)
22           USE types
23           IMPLICIT NONE
24             INTERFACE
25                FUNCTION fcn(t, y)     ! right-hand side of the IVP, y'(t) = fcn(t, y)
26                  USE TYPES, ONLY : dp
27                  IMPLICIT NONE
28                  REAL(KIND = dp), INTENT(IN) :: t, y(:)
29                  REAL(KIND = dp) :: fcn(SIZE(y))
30                END FUNCTION fcn
31                FUNCTION dfcn(t, y)    ! Jacobian of fcn(t, y)
32                  USE TYPES, ONLY : dp
33                  IMPLICIT NONE
34                  REAL(KIND = dp), INTENT(IN) :: t, y(:)
35                  REAL(KIND = dp) :: dfcn(SIZE(y), SIZE(y))
36                END FUNCTION dfcn
37                FUNCTION sint(a, b, y, estint, estidec)
38                  USE types
39                  USE ieulidec
40                  IMPLICIT NONE
41                  REAL(KIND = dp), INTENT(IN) :: a, b
42                  REAL(KIND = dp), INTENT(INOUT):: y(:)
43                  REAL(KIND = dp), INTENT(OUT):: estint, estidec
44                  REAL(KIND = dp) :: sint
45                END FUNCTION sint
46             END INTERFACE
```

```
47
48          ! Local variables
49          !-----------------
50             REAL(KIND = dp), INTENT(INOUT) :: y(:)   !used for inital values
51             REAL(KIND = dp) :: h,h0                  !step size
52             REAL(KIND = dp), POINTER :: solution(:,:)!solution vector
53             REAL(KIND = dp), INTENT(OUT) :: nstx     !the root of v(t)
54             REAL(KIND = dp), INTENT(OUT) :: integral !Integral on [0,nstx]
55             REAL(KIND = dp), INTENT(OUT) :: est      !error estimate
56             LOGICAL :: linear                        !flag for ieulidec
57             INTEGER(KIND = i4b) :: method, pord      !see ieulidec.f90
58             INTEGER(KIND = i4b) :: ifail             !error flag for ieulidec
59             REAL(KIND = dp) :: a, b, Newton_tol      !auxiliary variables
60             REAL(KIND = dp) :: slope(1:1), h_min     !auxiliary variables
61             REAL(KIND = dp) :: integral1, integral2  !integral auxiliary variables
62             REAL(KIND = dp) :: global_rel=1.0_dp,tol=HUGE(1.0_dp) !tolerances, see ieulidec.f90
63             REAL(KIND = dp) :: estimator=HUGE(1.0_dp),estint      !error estimates
64             REAL(KIND = dp) :: estidec               !estimate auxiliary variable
65             INTEGER(KIND = i4b) :: i,w=0             !auxiliary variables
66
67             TYPE TGlobalsol                          ! auxiliary structure
68               REAL(KIND = dp), pointer :: daten(:,:) ! to build solution vector
69               TYPE(TGlobalsol), POINTER :: next
70             END TYPe TGlobalsol
71             TYPE (TGlobalsol), pointer :: globalsol
72             TYPE (TGlobalsol), pointer :: hp
73
74          !Initialization
75          !--------------
76
77            pord = 4
78            method = 3
79            a = 0._dp
80            b = h
81            nstx = 0
82            ifail = 0
83            est = 0
84            estimator = 0
85            integral = 0
86            integral1 = 0
87            integral2 = 0
88            w = 0
89            NULLIFY(globalsol)
90            NULLIFY(hp)
91            linear = .false.
92
93          !Main loop for local connection strategy
94          !---------------------------------------
95            DO WHILE ((nstx==0) .AND. (ifail==0))
96
97          estimator=tol        ! reset INOUT parameters for ivp_ieuleridec
98          global_rel=1.0_dp
99          h0=h                 ! stepsize is reduced to h/4 in the subroutine
100
101         ! IVP integration on one Zadunaisky interval
102
103           call ivp_ieuleridec(fcn, dfcn, a, y, b, .false., h0, estimator, &
104                    global_rel, 10._dp**(-12), 4, ifail, solution=solution, &
105                    order_poly=4, use_damping=.true., use_stepcontrol=.false.)
106
107             IF (estimator>est) est=estimator
108
109             IF (ifail .ne. 0) write (*,*) "ifail: ", ifail, "delta: ", h-b+a
110
111   !concatenate solution vector
112   !---------------------------
```

```
113       ALLOCATE(hp)
114       ALLOCATE(hp%daten(size(solution,1),size(solution,2)))
115       hp%daten=solution
116       hp%next=>globalsol
117       globalsol=>hp
118       w=w+1
119
120     !check for sign change
121     !---------------------
122       DO i=0, SIZE(solution(:,1))-2
123         IF (solution(i,1)*solution(i+1,1) .lt. 0) THEN !has the sign changed?
124           nstx = (a+i*h/pord)-solution(i,1)*(h/pord/(solution(i+1,1)-&
125                                              solution(i,1))) !interpolate nstx
126         END IF
127       END DO
128
129     !perform integration
130     !----------------------
131       IF (nstx .EQ. 0) THEN
132          integral1=integral1+2*h/pord*(7*solution(0,1)+32*solution(1,1)+&
133                   12*solution(2,1)+32*solution(3,1)+7*solution(4,1))/45
134          integral2=integral2+h/pord*(solution(0,1)+4*solution(1,1)+&
135                    2*solution(2,1)+4*solution(3,1)+solution(4,1))/3
136       END IF
137
138     !jump to the next interval
139     !-------------------------
140       a=a+h
141       b=a+h
142       y=solution(pord,1)
143       estimator=0
144     END DO
145
146     !integration of the last interval
147     !-------------------------------
148     integral=sint(a-h, nstx, solution(0,:), estint, estidec)
149     DEALLOCATE(solution)
150     IF (ASSOCIATED(globalsol)) THEN
151       ALLOCATE(solution(w*(SIZE(globalsol%daten,1)-1),SIZE(globalsol%daten,2)))
152     ELSE
153       WRITE (*,*) "Fatal error!"
154     END IF
155
156     !generate solution vector
157     !------------------------
158     w=0
159     hp=>globalsol
160     DO WHILE (ASSOCIATED(hp))
161       w=w+1
162       solution(SIZE(solution)-w*(SIZE(hp%daten,1)-1)+1:SIZE(solution)-&
163            (w-1)*(SIZE(hp%daten,1)-1),:)=hp%daten(1:SIZE(hp%daten)-1,:)
164       hp=>hp%next       ! note that solution does not contain the endpoint
165     END DO              ! because v(t) is negative at this point anyway
166
167     !global error for the method
168     !---------------------------
169     slope=ABS(fcn(nstx,(/0._dp/)))  ! for error estimate of the root of v(t)
170     estimator=ABS(integral1-integral2)+estint+est*ABS(a-h)+&  ! total error in X_R
171               estidec*(nstx+ABS((est+estidec)/slope(1))-a+h)
172     est=estimator
173
174     IF (integral>0._dp) THEN
175       integral=integral+integral1
176     ELSE
177       integral=integral1
178     END IF
```

```
179
180  END SUBROUTINE calc
181
182  END MODULE work
183
184  FUNCTION sint(a,b,y,estint,estidec)
185
186  !integration on the last interval
187  !--------------------------------
188
189    USE types
190    USE ieulidec
191    IMPLICIT NONE
192    REAL(KIND=dp), INTENT(IN) :: a, b
193    REAL(KIND=dp), INTENT(INOUT)::y(:)
194    REAL(KIND=dp), INTENT(OUT) ::estint, estidec
195    REAL(KIND=dp) :: global_rel=1.0_dp,h,h0,tol=HUGE(1.0_dp)
196    INTEGER(KIND=i4b) ::ifail
197    REAL(KIND=dp) :: sint, help, estimator, help2
198    REAL(KIND=dp), POINTER :: solution(:,:)
199
200    INTERFACE
201
202    FUNCTION fcn(t, y)        !right-hand side of the differential equation
203      USE types, ONLY:dp
204      IMPLICIT NONE
205      REAL (KIND=dp), INTENT(IN) :: t, y(:)
206      REAL (KIND=dp) :: fcn(size(y))
207    END FUNCTION fcn
208
209    FUNCTION dfcn(t, y)       !Jacobian of fcn
210      USE types, ONLY :dp
211      IMPLICIT NONE
212      REAL (KIND=dp), INTENT(IN):: t, y(:)
213      REAL (KIND=dp) :: dfcn(size(y), size(y))
214    END FUNCTION dfcn
215    END INTERFACE
216
217    h=b-a
218
219
220  estimator=tol        ! reset INOUT parameters for ivp_ieuleridec
221  global_rel=1.0_dp
222  h0=h                 ! step size is adjusted correctly in the subroutine
223
224  ! solution from the start of the last interval to the root of v(t)
225
226    call ivp_ieuleridec(fcn, dfcn, a, y, b, .false., h0, estimator, &
227             global_rel, 10._dp**(-12), 4, ifail, solution=solution, &
228             order_poly=4, use_damping=.true., use_stepcontrol=.false.)
229
230    help=2._dp*h/4*(7*solution(0,1)+32*solution(1,1)+&
231                    12*solution(2,1)+32*solution(3,1)+7*solution(4,1))/45
232    help2=h/4*(solution(0,1)+4*solution(1,1)+&
233                     2*solution(2,1)+4*solution(3,1)+solution(4,1))/3
234    estint=abs(help-help2)
235    estidec=estimator
236    sint=help
237  END FUNCTION sint
238
239  FUNCTION fcn(t, y)  !right-hand side of the differential equation
240   USE types, ONLY:dp
241   USE parameters
242   IMPLICIT NONE
243   REAL (KIND=dp), INTENT(IN) :: t, y(:)
244   REAL (KIND=dp) :: fcn(size(y))
```

```
245
246     fcn=(/ -y(1)/t-D_null*y(1)*y(1)+V/t-G_null/)
247   END FUNCTION fcn
248
249   FUNCTION dfcn(t, y)  !Jacobian of the rigth-hand side
250    USE types, ONLY :dp
251    USE parameters
252    IMPLICIT NONE
253    REAL (KIND=dp), INTENT(IN):: t,y(:)
254    REAL (KIND=dp) :: dfcn(size(y),size(y))
255
256    dfcn(1,:)=(/ -1/t-2.*D_null*y(1) /)
257   END FUNCTION dfcn
```

A.1.3 Parameters Routine

```
1   MODULE parameters
2
3   !the model parameters and step-size are set
4   !-----------------------------------------
5
6   USE types
7
8   REAL (KIND = dp)     :: pi       =   3.1415926535897932385_dp
9   REAL (KIND = dp)     :: g        =   9.81_dp
10  !model parameters
11  !----------------
12  REAL (KIND = dp)     :: rho_bar  = 200._dp
13  REAL (KIND = dp)     :: h_bar    =   3_dp
14  REAL (KIND = dp)     :: psi
15  REAL (KIND = dp)     :: h_null   =   2._dp
16  REAL (KIND = dp)     :: v_null   =  35._dp
17  REAL (KIND = dp)     :: psi_null
18  REAL (KIND = dp)     :: mu       =   0.155_dp
19  REAL (KIND = dp)     :: rho_t    =  10._dp
20  REAL (KIND = dp)     :: C_D      =   0.01_dp
21  REAL (KIND = dp)     :: phi
22  REAL (KIND = dp)     :: k_p      =   1._dp
23  REAL (KIND = dp)     :: G_null
24  REAL (KIND = dp)     :: V
25  REAL (KIND = dp)     :: D_null
26
27  !initial step-size
28  !-----------------
29  REAL (KIND = dp)     :: h0,h        = 2._dp**(-7)
30
31  !initial value - will be set automatically
32  !-----------------------------------------
33  REAL (KIND = dp)     :: ys
34
35  !-----------------------------------------------------
36  !run calculations with global connection strategy
37  !.true. means yes. This takes as much time as it took
38  !       for the local connection strategy
39  !.false. means no, this halves the runtime
40  !-----------------------------------------------------
41  LOGICAL              :: accurate = .true.
42
43  !The filename for solution output
44  !--------------------------------
45  CHARACTER(LEN=15)    :: filename = 'avalanche.dat'
46
47  CONTAINS
48
49  SUBROUTINE init
```

```
50
51  !calculates the parameters in the differential equation
52
53  psi      = -pi/6._dp
54  psi_null = pi/20._dp
55  phi      = 5*pi/36._dp
56
57  IF (phi .GE. psi_null) THEN
58   k_p=(cos(psi_null)+sqrt(cos(psi_null)**2_dp-cos(phi)**2_dp))/&
59       (cos(psi_null)-sqrt(cos(psi_null)**2_dp-cos(phi)**2_dp))
60  ELSE
61   k_p=1.0_dp
62  END IF
63
64  G_null = g*(mu*COS(psi)-SIN(psi))
65  V      = v_null*COS(psi_null-psi)*(1+k_p*g*h_null*COS(psi_null)/(2*v_null**2))
66  D_null = rho_t*.5_dp*C_D/(rho_bar*h_bar)
67  ys     = V
68  END SUBROUTINE init
69
70  END MODULE parameters
```

A.1.4 IDeC Routine

```
1   ! ---------------------------------------------------------------------
2   ! This is part of the 'Solve Singular BVP' (SOLVSING) package developed
3   ! by W.Auzinger, O.Koch, P.Kofler, E.Weinmueller,
4   ! Department of Applied Mathematics and Numerical Analysis,
5   ! Vienna University of Technology, Vienna, Austria.
6   ! ---------------------------------------------------------------------
7   MODULE ieulidec
8   ! 'Implicit EULer IDEC method for singular ivps' (IEULIDEC)
9   ! Version 1.09, last modified 05-05-2001.
10  ! --- Function and Algorithm ---
11  ! Integrate the IVP of ODE y'(x) = f(x,y) with constant step size using
12  ! implicit Euler's method with global error estimation by Zadunaisky's
13  ! technique and Iterated Defect Correction (IDeC). Check the global
14  ! error and readjust the step size to satisfy a given global tolerance.
15  ! Repeat the integration if is is necessary. (This tolerance checking
16  ! may be skipped to examine asymtotic behaviour.)
17  ! This is a general purpose module for integrating singular IVPs and
18  ! avoiding order reduction. But away from the singularity, the used
19  ! method is quite inefficient.
20  ! --- Compiling ---
21  ! This module has to be compiled third, following the linalg module and
22  ! it is not dependant on function data. The usage of ivp_ieuleridec can
23  ! be seen in t_idec_a.f90 for asymtotic usage (with tolerance checking
24  ! deactivated) and in t_idec_t.f90 for global tolerance usage.
25  ! --- Bibliography ---
26  ! W.Auzinger,O.Koch,P.Kofler,E.Weinmueller,
27  ! 'The Application of Shooting to Singular Boundary Value Problems',
28  ! Technical Report No. 126, Austria 1999.
29  ! M.Metcalf, J.Reid,
30  ! 'Fortran 90 Explained', Oxford University Press (1990),
31  ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
32  ! 'Numerical Recipes in Fortran 90', Cambridge Univ. Press (1996).
33  ! P.E.Zadunaisky,
34  ! 'On the Estimation of Errors Propagated in the Numerical Integration
35  ! of Ordinary Differential Equations', Numer.Math. 27(1976), pp. 21--39.
36  ! ---------------------------------------------------------------------
37  ! Constants
38  ! ---------------------------------------------------------------------
39     USE TYPES                 ! predefined data-types
40     IMPLICIT NONE
41     REAL(KIND = dp), PARAMETER :: safety_factor_stepsize = 0.9_dp, &
```

```
                    min_update_stepsize = 2.0_dp, &
                    max_update_stepsize = 8.0_dp, &
                    dumping_factor_newton = 1.0_dp, &
                    default_min_stepsize = 10.0_dp*EPSILON(1.0_dp), &
                    relative_error_factor = 1.0_dp
!       safety_factor_stepsize ... safety factor for new step sizes.
!          If automatic step size control is enabled (stepcontrol=true)
!          the new step-size is multiplied by this factor to be on the
!          save side, as the new step-size estimation is not accurate.
!          But smaller factors make the integration inefficient. If the
!          step sizes tend to be too large for special problems, which
!          can be seen by more than two iterations over the given
!          intervall, this factor should be decreased. <= 1.0_dp
!       min_update_stepsize ... min update factor for new step sizes.
!          If automatic step size control is enabled (stepcontrol=true)
!          and the step size has to be reduced it is at least divided
!          by this factor. To deactivate this, use 0.0_dp. >= 0.0_dp
!       max_update_stepsize ... max update factor for new step sizes.
!          If automatic step size control is enabled (stepcontrol=true)
!          and the step size has to be reduced it is at most divided
!          by this factor to avoid too fast reduction. To deactivate
!          this, max_update_stepsize = HUGE(1.0_dp). > 0.0_dp
!       dumping_factor_newton ... factor first dumped Newton iteration
!          If damping for Newton iteration is enabled (damping=true),
!          the first damped step is the full step multiplied by this
!          factor. Then the factor is taken half in each further damped
!          step. <= 1.0_dp
!       default_min_stepsize ... default minimal stepsize
!          If the minimum step-size parameter is skipped, this is used.
!       relative_error_factor ... the factor the global error estimate
!          has to be less than the solution itself to indicate a valid
!          error estimation, or otherwise cause error 5. <= 1.0_dp
   INTEGER(KIND = i4b), PARAMETER :: max_newton_steps = 25, &
                                     max_lambda_steps = 10
!       max_newton_steps ... maximum of Newton iterations for Euler
!          If the Newton interation for nonlinear problems in the
!          Implicit-Euler method needs more than this number of steps,
!          most likely there is no convergence, thus the method stops,
!          reporting an error. For special problems with very slow
!          convergence, this number may be increased.
!       max_lambda_steps ... maximum of damping steps in Newton method
!          Newton convergence is improved by damping (damping=true),
!          using decreasing steps in the direction given by the
!          Newton increment. This is the number of damped steps to
!          try. If no satisfying solution is found, most likely there
!          is no convergence, thus the method stops reporting an error.
   LOGICAL, PARAMETER :: default_damping = .true., &
                         default_stepcontrol = .true.
!       default_damping ... default use damping for Newton iteration
!       default_stepcontrol ... default use of step-control

   PRIVATE
   PUBLIC :: ivp_ieuleridec

CONTAINS
! ----------------------------------------------------------------------
SUBROUTINE ivp_ieuleridec(fcn, dfcn, a, y, b, linear, h, global_tol, &
           global_rel, newton_tol, order_method, ifail, solution, &
           number_iter, min_stepsize, order_poly, use_damping, &
           use_stepcontrol)
! ----------------------------------------------------------------------
! Parameters: fcn(in), dfcn(in), a(in), y(in/out), b(in), linear(in),
! h(in/out), global_tol(in/out), global_rel(in/out), newton_tol(in),
! order_method(in), ifail(out), solution(opt out), number_iter(opt out),
! min_stepsize(opt in), order_poly(opt in), use_damping(opt in),
! use_stepcontrol(opt in)
```

```
108  ! ------------------------------------------------------------------------
109  ! required Arguments
110  ! ------------------------------------------------------------------------
111     USE LINALG, ONLY : lubksb, ludcmp ! we need the LU decomposition
112     IMPLICIT NONE
113     INTERFACE
114        FUNCTION fcn(t, y)     ! right side of the IVP, y'(t) = fcn(t, y)
115           USE TYPES, ONLY : dp
116           IMPLICIT NONE
117           REAL(KIND = dp), INTENT(in) :: t, y(:)
118           REAL(KIND = dp) :: fcn(SIZE(y))
119        END FUNCTION fcn
120        FUNCTION dfcn(t, y)    ! Jacobian of fcn(t, y)
121           USE TYPES, ONLY : dp
122           IMPLICIT NONE
123           REAL(KIND = dp), INTENT(in) :: t, y(:)
124           REAL(KIND = dp) :: dfcn(SIZE(y), SIZE(y))
125        END FUNCTION dfcn
126     END INTERFACE
127     REAL(KIND = dp), INTENT(in) :: a, b, newton_tol
128  !        a, b ... interval to perform the integration
129  !        newton_tol ... tolerance of Newton-iteration in implicit Euler
130  !           method. Both Newton-residual and increment are checked.
131     REAL(KIND = dp), INTENT(in out) :: y(:), h, global_tol, global_rel
132  !        y ... (in) initial value y(a), (out) approximation for y(b)
133  !        h ... (in) starting step, (out) last used step, or suggestion
134  !           for better step if step control is disabled, use h<0 for tvp
135  !        global_tol ... (in) global tolerance to fulfill,
136  !           (out) estimation of the maximum of the global error on a..b
137  !        global_rel ... (in) global relative tolerance to fulfill,
138  !           (out) estimation of the relative global error on a..b
139     LOGICAL, INTENT(in) :: linear
140  !        linear ... true if the initial value problem is linear
141     INTEGER(KIND = i4b), INTENT(in) :: order_method
142  !        order_method ... order of the method = iterations of IDeC + 1
143     INTEGER(KIND = i4b), INTENT(out) :: ifail
144  !        ifail ... error variable, set due to several errors
145  !            0 = everything is ok, integration was successfull
146  !           Negative numbers indicate wrong or unproper input values:
147  !           -1 = illegal interval range, error in parameters a,b
148  !           -2 = illegal smallest step size, error in min_stepsize
149  !           -3 = illegal starting step size, error in parameter h
150  !           -4 = illegal global tolerance, error in global_tol/rel
151  !           -5 = illegal Newton tolerance, error in newton_tol
152  !           -6 = illegal method, error in order_method, order_polynomial
153  !           Positive numbers indicate problems during the integration.
154  !           In these cases a solution *may* be available, but does not
155  !           fulfill given tolerance requirements.
156  !            1 = could not complete integration due to h < h_min
157  !            2 = Newton-iteration needed too many steps, indicating bad
158  !                or no convergence of Newton's method
159  !            3 = global error did not decrease during IDeC
160  !            4 = Newton residual did not decrease during the iteration,
161  !                indicating bad or no convergence of Newton's method
162  !            5 = the global error (estimate) was larger than 100 percent
163  !                indicating a step size which is much too large.
164  ! ------------------------------------------------------------------------
165  ! optional Arguments
166  ! ------------------------------------------------------------------------
167     REAL(KIND = dp), OPTIONAL, POINTER :: solution(:,:)
168  !        solution ... the estimation of the complete solution at the
169  !           used grid (y_0,y_1,...y_N), array indices starting at 0.
170     INTEGER(KIND = i4b), OPTIONAL, INTENT(out) :: number_iter
171  !        number_iter ... number of iterations over stepsize, 1-4. If
172  !           this returns a value >> 2 on output, the safety factor may
173  !           be too large for the given problem.
```

```
174      REAL(KIND = dp), INTENT(in), OPTIONAL :: min_stepsize
175   !        min_stepsize ... smallest absolute allowed step size, >= 0.0
176      INTEGER(KIND = i4b), INTENT(in), OPTIONAL :: order_poly
177   !        order_poly ... order Zadunaisky-polynomials (>=idec_iter+1)
178      LOGICAL, INTENT(in), OPTIONAL :: use_damping, use_stepcontrol
179   !        use_damping ... use damping for Newton iteration
180   !        use_stepcontrol ... use step-control to fullfill tolerance. If
181   !           not, this subroutine returns after the first integration.
182   ! ------------------------------------------------------------------------
183   ! Local variables for optional arguments or their defaults
184   ! ------------------------------------------------------------------------
185      REAL(KIND = dp) :: h_min
186   !        h_min ... used smallest step size from min_stepsize or default.
187      INTEGER(KIND = i4b) :: pord, idec_iter
188   !        pord ... used order polynomials (>=idec_iter+1)
189   !        idec_iter ... number of iterations of IDeC
190      LOGICAL :: damping, stepcontrol
191   !        damping ... use damping for Newton iteration
192   !        stepcontrol ... use step-control to fullfill tolerance
193   ! ------------------------------------------------------------------------
194   ! Local variables
195   ! ------------------------------------------------------------------------
196      INTEGER(KIND = i4b) :: j, iter, step, pi, feldi, idec, maxsteps
197   !        j ... loop counter
198   !        iter ... iteration counter for step size/tolerance iteration
199   !        step ... step counter in integration from a to b
200   !        pi ... step counter for little steps for Zadunaisky-polynomials
201   !        feldi ... reference value for interpolation in yfield
202   !        idec ... counter IDeC iteration
203   !        maxsteps ... total number of steps in the integration
204      REAL(KIND = dp) :: ya(SIZE(y)), x, est(SIZE(y)), global_est, ratio, &
205                         old_h, old_est, global_y, old_y, oop
206   !        ya ... store initial value for start of several integrations
207   !        x ... current position during the integration in [a,b]
208   !        est ... estimation of a single global error by Zadunaisky
209   !        global_est ... maximum of norm of estimations of global error
210   !        ratio ... step size-update ratio
211   !        old_h ... last used step size
212   !        old_est ... estimated error of solution of last iteration
213   !        global_y ... maximum of norm of estimations of solution
214   !        old_y ... maximum of norm of estimations of last iteration
215   !        oop ... exponent for ratio in error estimation ( 1/(order) )
216      REAL(KIND = dp), ALLOCATABLE :: yfield(:,:),yfieldw(:,:),yfieldb(:,:)
217   !        yfield ... store all approximations to interpolate the solution
218   !        yfieldw ... approximations workspace to improve solution
219   !        yfieldb ... store basic approximations for IDeC
220   ! ------------------------------------------------------------------------
221   ! Code
222   ! ------------------------------------------------------------------------
223      IF (debug_output) WRITE(*,'(A)') '  ---> ivp_ieuleridec called'
224
225   ! check the input arguments and set up the optional arguments
226      IF (PRESENT(number_iter)) number_iter = 0
227      ifail = -1                ! check interval
228      IF ( ((h>0.0_dp).AND.(b<= a)) .OR. ((h<0.0_dp).AND.(a<=b)) ) RETURN
229      ifail = -2                ! check smallest step size
230      IF (PRESENT(min_stepsize)) THEN
231         h_min = min_stepsize   ! min stepsize was supplied
232      ELSE
233         h_min = default_min_stepsize ! use default
234      END IF
235      IF ( (h_min < 0.0_dp)  .OR. &
236         (h_min <= EPSILON(1.0_dp)) ) RETURN ! too small
237      IF (h_min > ABS(b-a)) RETURN ! too large
238      ifail = -3                ! check starting step size
239      IF (h == 0.0_dp) RETURN
```

```
240      IF (ABS(h) < h_min) RETURN ! too small, or use h = SIGN(h_min,h)
241      IF (ABS(h) > ABS(b-a)) RETURN ! too large, or use h = SIGN(b-a,h)
242      ifail = -4                 ! check global tolerance
243      IF (global_tol <= 0.0_dp) RETURN
244      IF ( (global_rel < 0.0_dp) .OR. (global_rel > 1.0_dp) ) RETURN
245      IF (.not. linear) THEN
246         ifail = -5              ! check Newton-tolerance
247         IF (newton_tol <= 0.0_dp) RETURN
248         ! IF (newton_tol > global_tol) RETURN ! should at least be exact
249                                 ! but this should rather be a warning, not
250                                 ! an error, thus we do not check it
251      END IF
252      ifail = -6                 ! check order and number of iterations
253      IF (order_method < 1) RETURN
254      idec_iter = order_method - 1
255      oop = 1.0_dp               ! the order estimation is for order-1
256      IF (order_method > 1) oop = 1.0_dp/(order_method-1)
257      IF (PRESENT(order_poly)) THEN
258         pord = order_poly       ! order was supplied
259         ! IF (pord < order_method) RETURN ! should not be, maybe for tests
260      ELSE
261         pord = order_method     ! use default
262      END IF
263      IF (PRESENT(use_damping)) THEN
264         damping = use_damping
265      ELSE
266         damping = default_damping ! use default
267      END IF
268      IF (PRESENT(use_stepcontrol)) THEN
269         stepcontrol = use_stepcontrol
270      ELSE
271         stepcontrol = default_stepcontrol ! use default
272      END IF
273      IF ((stepcontrol).AND.(order_method<2)) RETURN ! we do not have an
274                                 ! estimation for order 1, as we need at
275                                 ! least 1 IDeC iteration, yielding order 2
276
277    ! verbose the used parameters to the debug output if wished
278      IF (debug_output) THEN
279         WRITE(*,*) ' * ivp: ',a,' to ',b,' (linear',linear,')'
280         WRITE(*,'(A,I1,A,I1,A,I1,A)') '  * method: order ',order_method, &
281                ' (polynomial ',pord,' with ',idec_iter,' IDeC iterations)'
282         WRITE(*,'(A,E14.8,A,E14.8,A)') '  * stepsize: initial ',h, &
283                ' (minimal ',h_min,')'
284         WRITE(*,*) ' * control: global tolerance ', global_tol, &
285                ' (step size control',stepcontrol,')'
286         WRITE(*,*) ' * Newton: tolerance ', newton_tol, ' (with damping', &
287                damping,')'
288      END IF ! debug_output
289
290    ! set up internal variables
291      ifail = 0
292      iter = 0
293      ya = y                     ! store initial value y(a)
294      old_est = HUGE(1.0_dp)
295      old_y = 1.0_dp
296
297    ! do the big step size/tolerance iteration until the maximum of the
298    ! estimated global error is below the given tolerance
299      DO                         ! big iteration over step size/tolerance
300         iter = iter + 1
301
302    ! calculate amount of necessary steps and adjust h to distribute grid
303    ! points equally spaced on [a,b], where the number of steps must be a
304    ! multiple of pord. h itselt is the real step size, thus h*pord is the
305    ! desired size of the big steps used by Zadunaisky's technique.
```

```
306         maxsteps = CEILING((b-a)/(h*pord)/(1+EPSILON(1.0_dp)))
307                                ! use h*(1+EPSILON) to avoid a further step
308                                ! due to round-off error in h
309         IF (maxsteps < 1) maxsteps = 1 ! one big step
310         h = (b-a)/maxsteps/pord ! in any case we have hnew <= h
311         old_h = h              ! save old stepsize
312         IF (debug_output) WRITE(*,'(A,I1,A,I5,A,E14.8)') '  Iteration: ',&
313            iter,' performing ', maxsteps,' steps with step size ',h
314
315   ! now check new adjusted step size again towards parameters
316         IF (ABS(h) < h_min) THEN
317            IF (iter == 1) THEN ! we did not do anything yet
318               ifail = -3       ! initial step size was too small
319            ELSE                ! iter > 1, so a solution is available
320               ifail = 1        ! could not complete because of h < h_min
321               global_tol = old_est ! that is what we got in last iteration
322               IF (old_y /= 0.0_dp) THEN
323                  global_rel = old_est/old_y ! relative global error
324               ELSE
325                  global_rel = old_est
326               END IF
327               h = old_h        ! with the last used step size
328                                ! solution has the values of the last run
329              iter = iter -1    ! the last iteration failed
330            END IF
331            EXIT                ! skip all further iterations
332         END IF ! ABS(h) < h_min
333
334   ! drop the last solution and inialize new soultion field at base of heap
335         IF (PRESENT(solution)) THEN
336            IF (ASSOCIATED(solution)) DEALLOCATE(solution) ! drop last
337            ALLOCATE(solution(0:maxsteps*pord,SIZE(y)))
338         END IF
339   ! allocate workspace after solution, thus must be freed before
340         ALLOCATE(yfield(0:maxsteps*pord,SIZE(y)),yfieldw(0:pord,SIZE(y)),&
341                  yfieldb(0:maxsteps*pord, SIZE(y)))
342
343   ! calculate the basic approximation using implicit Euler's method
344         x = a
345         step = 0
346         y = ya                 ! initialise initial values
347         yfield(step,:) = y     ! store first value, the initial value
348         global_y = MAXVAL(ABS(y)) ! norm of y(a), set for idec_iter=0
349         global_est = 0.0_dp ! y(a) is exact, thus error is 0
350         DO j = 1, maxsteps     ! loop for big steps
351            DO pi = 1, pord     ! loop for little, real steps in polynomial
352               IF (linear) THEN
353                  y = impl_lin_step(h,.false.)
354               ELSE
355                  IF (damping) THEN
356                     y = impl_nlin_damp_step(h,.false.)
357                  ELSE
358                     y = impl_nlin_step(h,.false.)
359                  END IF
360                  IF (ifail /= 0) THEN ! stop integration due to error
361                     CALL error_in_loop()
362                     RETURN
363                  END IF ! ifail /= 0
364               END IF
365               x = x + h        ! go to next grid point
366               step = step + 1
367               yfield(step,:) = y ! store basic approximation
368            END DO ! DO pi over little steps
369         END DO ! DO j over steps
370         yfieldb = yfield       ! store the basic approximation of order 1
371
```

```
372  ! start IDeC iteration, we do at least one iteration, as is order >= 2
373       DO idec = 1, idec_iter
374
375  ! calculate idec-th approximations for the neighbouring problem
376          x = a               ! initialise initial values
377          step = 0
378          y = yfield(step,:)
379          yfieldw(0,:) = y
380          global_y = MAXVAL(ABS(y)) ! norm of y(a)
381          global_est = 0.0_dp ! y(a) is exact, thus error is 0
382          DO j = 1, maxsteps  ! loop for big Zadunaisky steps
383             feldi = step
384             DO pi = 1, pord  ! loop for little, real steps in polynomial
385                IF (linear) THEN
386                   y = impl_lin_step(h,.true.)
387                ELSE
388                   IF (damping) THEN
389                      y = impl_nlin_damp_step(h,.true.)
390                   ELSE
391                      y = impl_nlin_step(h,.true.)
392                   END IF
393                   IF (ifail /= 0) THEN ! stop integration due to error
394                      CALL error_in_loop()
395                      RETURN
396                   END IF ! ifail /= 0
397                END IF
398                x = x + h     ! go to next grid-points
399                step = step + 1
400                est = y-yfieldb(step,:) ! error estimation for this y
401                yfieldw(pi,:) = yfield(step,:) - est ! update solution
402                global_y = MAX(global_y,MAXVAL(ABS(y))) ! get maximum
403                global_est = MAX(global_est,MAXVAL(ABS(est))) ! get max
404             END DO ! DO pi over little steps
405             yfield(step-pord:step-1,:) = yfieldw(0:pord-1,:)
406             yfieldw(0,:) = yfieldw(pord,:) ! last point becomes first
407          END DO ! DO j over steps
408          yfield(step,:) = yfieldw(pord,:)
409
410          ! alternatively we could use the estimation at b,
411          ! global_est = MAXVAL(ABS(y-yfieldb(step,:)))
412          ! but we are interested in BVPs, thus we take the maximas
413
414          IF (debug_output) WRITE(*,'(A,I1,A,I1,A,E14.8)') &
415             '  Iteration: ', iter, ' in IDeC iteration ', idec, &
416             ' max.error est. ', global_est
417
418          IF ( (stepcontrol) .AND. &
419               (global_est < global_tol + global_y * global_rel) ) EXIT
420          ! solution is accurate before we used all our idec iterations
421       END DO ! DO idec over iterations
422
423  ! store the last, most accurate approximation, which is is of order pord
424       y = yfield(step,:)     ! y approximates y(b)
425       IF (PRESENT(solution)) solution = yfield ! the last whole solution
426       DEALLOCATE(yfield,yfieldw,yfieldb) ! freeing data leaving solution
427
428  ! check if step size was small enough to produce a correct error
429  ! estimation, i.e. if the error estimation is smaller than the solution,
430  ! global error/solution < 100 percent
431       IF (global_est > relative_error_factor*global_y) THEN
432          ifail = 5           ! global error is 100 percent
433          global_tol = global_est ! we got this with last step size
434          IF (global_y /= 0.0_dp) THEN
435             global_rel = global_est/global_y ! relative error
436          ELSE
437             global_rel = global_est
```

```
438            END IF
439            EXIT                ! we also have a solution, but useless
440         END IF ! global_est > global_y
441         IF (debug_output) WRITE(*,'(A,I1,A,E14.8,A,E14.8)') &
442            '  Iteration: ', iter, ' error estimation ', global_est, &
443            ' local error est. ', global_est/global_y
444
445   ! check the global error, accept solution or refine the step size
446         IF ( (global_est <= global_tol + global_y * global_rel) .OR. &
447              (.not. stepcontrol) ) THEN
448            global_tol = global_est ! save estimation to return it
449            IF (global_y /= 0.0_dp) THEN
450               global_rel = global_est/global_y ! relative error
451            ELSE
452               global_rel = global_est
453            END IF
454            EXIT                ! OK, we succeeded and have a solution
455                                ! without step control do just 1 iteration
456         ELSE                   ! reject solution, calculate new step size
457            IF (global_est >= old_est) THEN
458               ifail = 3        ! error is not decreasing
459               global_tol = global_est ! we got this with last step size
460               IF (global_y /= 0.0_dp) THEN
461                  global_rel = global_est/global_y ! relative error
462               ELSE
463                  global_rel = global_est
464               END IF
465                                ! we also have a solution
466               EXIT
467            END IF
468            old_h = h           ! store old h and old error estimation
469            old_est = global_est
470            old_y = global_y
471
472   ! calculate the new stepsize upon our successfull error estimation
473            ratio = (global_est / (global_tol+global_y*global_rel) )**oop
474            ! the order of method is order_method, but the global error
475            ! estimation is for order_method-1, as derived by the last
476            ! iteration, which is of course used for the solution.
477            IF (ratio < min_update_stepsize) ratio = min_update_stepsize
478            IF (ratio > max_update_stepsize) ratio = max_update_stepsize
479            h = h/ratio*safety_factor_stepsize ! reduced step size
480            IF (debug_output) WRITE(*,'(A,I1,A,E14.8,A,E14.8)') &
481               '  Iteration: ', iter, ' update ratio ', ratio, &
482               ' yields new step size ', h
483         END IF ! global_est <= global_tol
484      END DO ! iter
485      IF (PRESENT(number_iter)) number_iter = iter
486      IF (debug_output) WRITE(*,'(A)') '  <--- ivp_ieuleridec returned'
487
488      RETURN
489   ! ----------------------------------------------------------------------
490   ! Local subroutines
491   ! ----------------------------------------------------------------------
492      CONTAINS
493
494   SUBROUTINE error_in_loop()
495   ! --- Function and Algorithm ---
496   ! Perform cleanup for errors during the do-loops, i.e. errors in the
497   ! implicit Euler method.
498   ! --- Arguments ---
499      IMPLICIT NONE
500   ! --- Code ---
501      DEALLOCATE(yfield,yfieldw,yfieldb) ! freeing data
502      IF (PRESENT(solution)) DEALLOCATE(solution)
503      IF (iter > 1) THEN        ! an old solution is available,
```

```
                              ! but we have no solution vector for that
      global_tol = old_est    ! we got in last iteration
      IF (old_y /= 0.0_dp) THEN
         global_rel = old_est/old_y ! relative error
      ELSE
         global_rel = old_est
      END IF
      h = old_h               ! with the last used step size
      iter = iter -1          ! the last iteration failed
      IF (PRESENT(number_iter)) number_iter = iter
   END IF
   RETURN
END SUBROUTINE error_in_loop

FUNCTION lagrang()
! Version 1.01, last modified 07-09-1998.
! --- Function and Algorithm ---
! Evaluate the polynomial sum_{j=0}^pord omega_j*f_j (lagrange-poly)
! interpolating yfield on [a,a+pord*h] at a given x. As x consists just
! of grid points for Implicit Euler,just return the value of
! yfield(step+1).
! required temporary storage: SIZE(y)
! --- Arguments ---
   IMPLICIT NONE
   REAL(KIND = dp) :: lagrang(SIZE(y))
!   REAL(KIND = dp), INTENT(in) :: x
!        x ... posititon the evaluation of lagrange-poly is required
! --- Global variables ---
!        step, yfield(:,:), y (size only)
! --- Code ---
   lagrang = yfield(step+1,:)
   RETURN
END FUNCTION lagrang

FUNCTION lagran2(x)
! Version 1.00, last modified 14-01-1998.
! --- Function and Algorithm ---
! Evaluate the polynomial sum_{j=0}^pod omega_j*f_j (lagrange-poly)
! interpolating yfield on [a,a+pord*h] at x. This is the general
! purpose method, which is not used by the main method.
! required temporary storage: 2*SIZE(y)+5
! --- Arguments ---
   IMPLICIT NONE
   REAL(KIND = dp) :: lagran2(SIZE(y))
   REAL(KIND = dp), INTENT(in) :: x
!        x ... posititon the evaluation of lagrange-poly is required
! --- Global variables ---
!        yfield(:,:), feldi, pord, h, y (size only)
! --- Local variables ---
   REAL(KIND = dp) :: u, w, ph(SIZE(y))
!        u ... x-value shifted to [0, pord*h]
!        w ... auxiliary data
!        ph ... value of lagrange-polynomial at x
   INTEGER(KIND = i4b) :: j, k
!        j, k ... loop counters
! --- Code ---
   u = x - a - h*feldi       ! move interval [a+...,...] to [0,...]
   ph = 0.0_dp               ! initialise sum of omega_j*f_j
   DO j = 0, pord            ! calculate sum omega_j*f_j
      w = 1.0_dp
      DO k = 0, pord ! calculate product omega_j (ord_poly fctrs)
         IF (j /= k) w = w * (u-h*k)/(j-k)
      END DO
      ph = ph + w*yfield(feldi+j,:) ! add omega_j*f_j
   END DO
   lagran2 = ph/h**pord
```

```
570      RETURN
571   END FUNCTION lagran2
572
573   FUNCTION difflagr(x)
574   ! Version 1.01, last modified 07-09-1998.
575   ! --- Function and Algorithm ---
576   ! Evaluate the derivative of the polynomial sum_{j=0}^pord omega_j*f_j
577   ! (lagrange-polynomial) interpolating yfield on [a,a+pord*h] at x=k*h.
578   ! required temporary storage: 2*SIZE(y)+7
579   ! --- Arguments ---
580      IMPLICIT NONE
581      REAL(KIND = dp) :: difflagr(SIZE(y))
582      REAL(KIND = dp), INTENT(in) :: x
583   !        x ... value the evaluation is required
584   ! --- Global variables ---
585   !        a, h, feldi, pord, yfield(:,:), y(size only)
586   ! --- Local variables ---
587      REAL(KIND = dp) :: w, w2, phs(SIZE(y))
588   !        u ... x-value shifted to [0,pord*h]
589   !        w, w2 ... auxiliary data
590   !        phs ... value of (lagrange-polynomial)' at x
591      INTEGER(KIND = i4b) :: i, j, k, l
592   !        i ... index of evaluation point in yfield
593   !        j, k, l, u ... loop counter
594   ! --- Code ---
595      i = nint((x-a-h*feldi)/h) ! move interval to [0,...]
596      phs = 0.0_dp              ! initialise sum of omega'_j*f_j
597
598      DO j = 0, pord            ! calculate sum omega'_j*f_j(pord+1 sum.)
599         w = 0.0_dp             ! initialise sum for omega'_j
600         DO k = 0, pord         ! calculate sum omega'_j (pord sumands)
601            IF ((j /= k).and.((i == j).or.(i == k))) THEN
602               w2 = 1.0_dp      ! initialise k-th product in omega'_j
603               DO l = 0, pord   ! calculate k-th product in omega'_j
604                  IF ((j /= l).and.(l /= k)) w2 = w2*(i-l)
605               END DO
606               w = w + w2       ! add components of omega'_j
607            END IF
608         END DO
609
610         DO k = 0, pord         ! calculate the constant factor of omega_j
611            IF (j /= k ) w = w/(j-k)
612         END DO
613         phs = phs + w*yfield(feldi+j,:) ! add omega'_j*f_j
614      END DO
615      difflagr = phs/h
616      RETURN
617   END FUNCTION difflagr
618
619   FUNCTION expl_lin_step(h, zadu)
620   ! Version 1.00, last modified 07-09-1998.
621   ! --- Function and Algorithm ---
622   ! Perform one explicit Euler step for fcn or neighbouring problem.
623   ! required temporary storage: 2*SIZE(y)+2
624   ! --- Arguments ---
625      IMPLICIT NONE
626      REAL(KIND = dp) :: expl_lin_step(SIZE(y))
627      REAL(KIND = dp), INTENT(in) :: h
628   !        h ... step size to perform implicit Euler step
629      LOGICAL, INTENT(in) :: zadu
630   !        zadu ... use Zadunaisky polynomials for neighbouring problem
631   ! --- Global variables ---
632   !        fcn, x, y
633   !        for zadu indirect usage of yfield(:,:), feldi, pord
634   ! --- Local variables ---
635      REAL(KIND = dp) :: w(SIZE(y)), lag(SIZE(y))
```

```
636  !       w ... inhomogeous part of fcn
637  !       lag ... explicit evaluation of lagran2 for IBM AIX XL Fortran
638  ! --- Called software components ---
639  !       direct ... for zadu difflag2
640  ! --- Code ---
641     w = y + h*fcn(x,y)
642     IF (zadu) THEN
643  !     w = w + h*(difflagr(x) - fcn(x, lagrang(x)))
644        lag = lagrang()
645        lag = difflagr(x) - fcn(x, lag)
646        w = w + h*lag
647     END IF
648     expl_lin_step = w         ! new solution for y
649     RETURN
650  END FUNCTION expl_lin_step
651
652  FUNCTION impl_lin_step(h, zadu)
653  ! Version 1.01, last modified 03-09-1998.
654  ! --- Function and Algorithm ---
655  ! Perform one implicit Euler step for linear or neighbouring problem.
656  ! required temporary storage: SIZE(y)^2+4*SIZE(y)+4
657  ! --- Arguments ---
658     IMPLICIT NONE
659     REAL(KIND = dp) :: impl_lin_step(SIZE(y))
660     REAL(KIND = dp), INTENT(in) :: h
661  !       h ... step size to perform implicit Euler step
662     LOGICAL, INTENT(in) :: zadu
663  !       zadu ... use Zadunaisky polynomials for neighbouring problem
664  ! --- Global variables ---
665  !       fcn, dfcn, x, y
666  !       for zadu indirect usage of yfield(:,:), feldi, pord, h
667  ! --- Local variables ---
668     REAL(KIND = dp) :: Jac(SIZE(y),SIZE(y)), w(SIZE(y)), &
669                        u, lag(SIZE(y))
670  !       Jac ... Jacobian for given x
671  !       w ... inhomogenous part of fcn
672  !       u ... auxiliary data
673  !       lag ... explicit evaluation of lagrang for IBM AIX XL Fortran
674     INTEGER(KIND = i4b) :: indx(SIZE(y)), i
675  !       indx ... pivoting indices for LU-decomposition
676  !       i ... loop counter
677  ! --- Called software components ---
678  !       direct ... ludcmp, lubksb, for zadu difflagr
679  ! --- Code ---
680  ! set up linear system Jac*y1=w for implicit Euler
681     w = 0.0_dp
682     u = x + h
683     w = y + h*fcn(u,w)        ! inhomogenous component of problem
684     IF (zadu) THEN
685  !     w = w + h*(difflagr(u) - fcn(u, lagrang(u)))
686        lag = lagrang()
687        lag = difflagr(u) - fcn(u, lag)
688        w = w + h*lag
689     END IF
690     Jac = -h*dfcn(u,y)        ! homogenous component of problem
691                               ! y dummy argument because linear
692     DO i = 1, SIZE(y)         ! loop over diagonal, add identity
693        Jac(i,i) = Jac(i,i) + 1.0_dp
694     END DO ! DO i over diagonal
695
696  ! solve linear system Jac*y1=w, destroying Jac and yielding y1 in w
697     CALL ludcmp(Jac, indx)
698     CALL lubksb(Jac, indx, w)
699
700     impl_lin_step = w         ! y1 = new solution for y
701     RETURN
```

```
END FUNCTION impl_lin_step

FUNCTION impl_nlin_step(h, zadu)
! Version 1.01, last modified 03-09-1998.
! --- Function and Algorithm ---
! Perform one Implicit Euler step for non-linear fcn or the neighbouring
! problem using simple Newton iteration.
! required temporary storage: SIZE(y)^2+5*SIZE(y)+7
! --- Arguments ---
   IMPLICIT NONE
   REAL(KIND = dp) :: impl_nlin_step(SIZE(y))
   REAL(KIND = dp), INTENT(in) :: h
!       h ... step size to perform implicit Euler step
   LOGICAL, INTENT(in) :: zadu
!       zadu ... use Zadunaisky polynomials for neighbouring problem
! --- Global variables ---
!       fcn, dfcn, x, y, Newton_tol, constant max_newton_steps
!       for zadu indirect usage of yfield(:,:), feldi, pord, h
! --- Local variables ---
   REAL(KIND = dp) :: Jac(SIZE(y),SIZE(y)), w(SIZE(y)), &
                      y1(SIZE(y)), u, nrmF, nrmdeltak, lag(SIZE(y))
!       Jac ... Jacobi-matrix for given x
!       w ... fcn call
!       nrmF ... norm of residual
!       nrmdeltak ... norm of Newton-increment (deltak)
!       y1 ... approximation of step
!       u ... auxiliary data
!       lag ... explicit evaluation of lagran2 for IBM AIX XL Fortran
   INTEGER(KIND = i4b) :: indx(SIZE(y)), i, index
!       indx ... pivoting indices for LU-decomposition
!       i ... loop counter
!       index ... iteration count in Newton-iteration
! --- Called software components ---
!       direct ... ludcmp, lubksb, for zadu difflagr
! --- Code ---
! get initial approximation for y1
   y1 = y
   u = x + h

! start Newton-iteration
   index = 0
   DO
      index = index + 1

! set up linear system Jac*y1^(i) = w for Newton-increment
      w = y + h*fcn(u, y1) - y1 ! calculate Newton-residual
      IF (zadu) THEN
!        w = w + h*(difflagr(u) - fcn(u, lagrang(u)))
         lag = lagrang()
         lag = difflagr(u) - fcn(u, lag)
         w = w + h*lag
      END IF
      nrmf = MAXVAL(ABS(w))  ! Newton-residual for this approximation
      Jac = h*dfcn(u, y1)    ! Jacobi-matrix
      DO i = 1, SIZE(y)      ! loop over diagonal, subtract identity
         Jac(i,i) = Jac(i,i) - 1.0_dp
      END DO ! DO i over diagonal

! solve linear system, destroying Jac and yielding delta y1 in w
      CALL ludcmp(Jac, indx)
      CALL lubksb(Jac, indx, w)

! iterate solution and exit if accurate
      y1 = y1 - w
      nrmdeltak = MAXVAL(ABS(w)) ! size of Newton-increment
      IF ((nrmf < Newton_tol) .AND. (nrmdeltak < Newton_tol)) EXIT
```

```
768
769         IF (index > max_newton_steps) THEN
770            ifail = 2           ! error, too many iterations
771            impl_nlin_step = 0.0_dp
772            RETURN
773         END IF
774      END DO ! Newton-loop
775
776      impl_nlin_step = y1       ! iterated y1 = new solution for y
777      RETURN
778   END FUNCTION impl_nlin_step
779
780   FUNCTION impl_nlin_damp_step(h, zadu)
781   ! Version 1.03, last modified 18-11-1998.
782   ! --- Function and Algorithm ---
783   ! Perform one Implicit Euler step for non-linear fcn or the neighbouring
784   ! problem using the damped Newton's method.
785   ! required temporary storage: SIZE(y)^2+8*SIZE(y)+8
786   ! --- Bibliography ---
787   ! Deuflhard,
788   ! 'Numerische Mathematik: Eine algorithmisch orientierte Einfhrung',
789   ! --- Arguments ---
790      IMPLICIT NONE
791      REAL(KIND = dp) :: impl_nlin_damp_step(SIZE(y))
792      REAL(KIND = dp), INTENT(in) :: h
793   !        h ... step size to perform implicit Euler step
794      LOGICAL, INTENT(in) :: zadu
795   !        zadu ... use Zadunaisky polynomials for neighbouring problem
796   ! --- Global variables ---
797   !        fcn, dfcn, x, y, newton_tol, constant max_newton_steps
798   !        for zadu indirect usage of yfield(:,:), feldi, pord, h
799   ! --- Local variables ---
800      REAL(KIND = dp) :: Jac(SIZE(y),SIZE(y)), res(SIZE(y)), lambda, &
801                         delta(SIZE(y)), y1(SIZE(y)), y2(SIZE(y)), u, &
802                         lag(SIZE(y)), nrmdeltak, w(SIZE(y))
803   !        Jac ... Jacobian for given x
804   !        res ... Newton-residual
805   !        delta ... Newton-increment
806   !        nrmdeltak ... norm of Newton-increment (delta)
807   !        y1 ... approximation of step
808   !        y2 ... approximation of step used by damping
809   !        u,w ... auxiliary data, work
810   !        lag ... explicit evaluation of lagrang
811   !        lambda ... last damping factor
812      INTEGER(KIND = i4b) :: indx(SIZE(y)), i, index, lindex
813   !        indx ... pivoting indices for LU-decomposition
814   !        i ... loop counter
815   !        index ... iteration count in Newton-iteration
816   !        lindex ... iteration count in damping loop
817   ! --- Called software components ---
818   !        direct ... ludcmp, lubksb, for zadu difflagr
819   ! --- Code ---
820      y1 = y                    ! initial approximation for y1
821      u = x + h                 ! target point
822      lambda = dumping_factor_newton/2.0_dp ! initial damping factor
823
824   ! calculate 1st Newton-residual
825      res = y+h*fcn(u, y1) - y1 ! calculate Newton-residual
826      IF (zadu) THEN
827   !     res = res + h*(difflagr(u) - fcn(u, lagrang(u)))
828         lag = lagrang()
829         lag = difflagr(u) - fcn(u, lag)
830         res = res + h*lag
831      END IF
832
833   ! start Newton-iteration
```

```
834      index = 0
835      DO
836         index = index + 1
837
838   ! set up linear system Jac*y1^(i) = delta for Newton-increment
839         delta = res
840         Jac = h*dfcn(u, y1)    ! Jacobian
841         DO i = 1, SIZE(y)      ! loop over diagonal, subtract identity
842            Jac(i,i) = Jac(i,i) - 1.0_dp
843         END DO ! DO i over diagonal
844
845   ! solve linear system, destroying Jac and yielding delta_y1 in delta
846         CALL ludcmp(Jac, indx)
847         CALL lubksb(Jac, indx, delta)
848
849   ! exit iteration if accurate
850         nrmdeltak = MAXVAL(ABS(delta)) ! size of Newton-increment
851         IF ((MAXVAL(ABS(res)) < newton_tol) .AND. &
852             (nrmdeltak < newton_tol*(1.0_dp+MAXVAL(ABS(y1))))) EXIT
853
854   ! update solution and iterate damping factors
855         lambda = MIN(1.0_dp,lambda*2.0_dp)
856         lindex = 0
857         DO
858            lindex = lindex + 1
859            y2 = y1 - lambda*delta
860
861   ! calculate new Newton-residual
862            res = y+h*fcn(u, y2) - y2 ! calculate Newton-residual
863            IF (zadu) res = res + h*lag
864
865   ! check "monotonic test" of the residuals by the increments
866            w = res             ! simplified Newton-increment
867            CALL lubksb(Jac, indx, w)
868
869            IF (MAXVAL(ABS(w)) <= (1.0_dp-lambda/2.0_dp)*nrmdeltak) THEN
870               y1 = y2          ! "monotonic test" is true
871               EXIT             ! therefore take this solution
872            END IF
873
874   ! shorten increment's length by 2
875            lambda = lambda/2.0_dp
876            IF (lindex > max_lambda_steps) THEN
877               ifail = 4        ! error, residual did not decrease
878               impl_nlin_damp_step = 0.0_dp
879               RETURN
880            END IF
881         END DO ! lambda dumping
882         IF (index > max_newton_steps) THEN
883            ifail = 2           ! error, too many iterations
884            impl_nlin_damp_step = 0.0_dp
885            RETURN
886         END IF
887      END DO ! Newton-loop
888
889      impl_nlin_damp_step = y1       ! iterated y1 = new solution for y
890      RETURN
891   END FUNCTION impl_nlin_damp_step
892
893   ! ------------------------------------------------------------------------
894   ! End of ivp_ieuleridec
895   ! ------------------------------------------------------------------------
896   END SUBROUTINE ivp_ieuleridec
897
898   END MODULE ieulidec
```

A.1.5 Auxiliary Routines

```
1  ! ------------------------------------------------------------------------
2  ! This is part of the 'Solve Singular BVP' (SOLVSING) package developed
3  ! by W.Auzinger, O.Koch, P.Kofler, E.Weinmueller,
4  ! Department of Applied Mathematics and Numerical Analysis,
5  ! Vienna University of Technology, Vienna, Austria.
6  ! ------------------------------------------------------------------------
7  MODULE linalg
8  ! --- Function and Algorithm ---
9  ! This module provides routines for doing linear algebra. The routines
10 ! were taken from 'Numerical Recipes in Fortran 90' and slightly
11 ! modified. The routines cover the calculating of normes and condition
12 ! numbers (with brute force), LU-decomposition and the inversion of a
13 ! matrix and QR-decomposition.
14 ! This is a general purpose module, so some methods provided here are
15 ! not used in the package.
16 ! --- Compiling ---
17 ! This module has to be compiled second, following the types module. For
18 ! the usage of the different subroutines see t_linalg.f90.
19 ! --- Bibliography ---
20 ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
21 ! 'Numerical Recipes in Fortran 90', Cambridge Univ.Press(1996).
22 ! ------------------------------------------------------------------------
23 ! Constants
24 ! ------------------------------------------------------------------------
25    USE TYPES                 ! predefined data-types
26    IMPLICIT NONE
27    INTERFACE swap            ! enable argument dependant method calls
28       MODULE PROCEDURE swap_i,swap_d,swap_dv,swap_dm, &
29          masked_swap_ds,masked_swap_dv,masked_swap_dm
30    END INTERFACE
31    PRIVATE
32    PUBLIC :: swap, ludcmp, lubksb, lubksbm, qrdcmp, qrsolv, qrdecomp, &
33              vIabs, v2abs, v1abs, mIabs, m1abs, m1absinv, L1cond, &
34              outerprod
35 ! ------------------------------------------------------------------------
36 ! Subroutines
37 ! ------------------------------------------------------------------------
38 CONTAINS
39
40 FUNCTION vIabs(v)
41 ! Version 1.00, last modified 15-08-2000.
42 ! --- Function and Algorithm ---
43 ! Calculate the Inifinity-norm of a given vector.
44 ! --- Arguments ---
45    IMPLICIT NONE
46    REAL(KIND = dp) :: vIabs
47    REAL(KIND = dp), INTENT(in) :: v(:)
48 !       v ... vector to calculate the Infinity-norm for
49 ! --- Code ---
50    vIabs = MAXVAL(ABS(v))
51    RETURN
52 END FUNCTION vIabs
53
54 FUNCTION v2abs(v)
55 ! Version 1.00, last modified 18-09-1998.
56 ! --- Function and Algorithm ---
57 ! Calculate the L2-norm of a given vector, i.e. sqrt(x_1^2+...+x_n^2).
58 ! --- Bibliography ---
59 ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
60 ! 'Numerical Recipes in Fortran 90', Cambridge Univ.Press(1996), p.1008.
61 ! --- Arguments ---
62    IMPLICIT NONE
63    REAL(KIND = dp) :: v2abs
64    REAL(KIND = dp), INTENT(in) :: v(:)
```

```
65  !       v ... vector to calculate L2-norm for
66  ! --- Code ---
67     v2abs = SQRT(DOT_PRODUCT(v,v))
68     RETURN
69  END FUNCTION v2abs
70
71  FUNCTION v1abs(v)
72  ! Version 1.00, last modified 21-10-1998.
73  ! --- Function and Algorithm ---
74  ! Calculate the L1-norm of a given vector, i.e. $|x_1|+... +|x_n|$.
75  ! --- Arguments ---
76     IMPLICIT NONE
77     REAL(KIND = dp) :: v1abs
78     REAL(KIND = dp), INTENT(in) :: v(:)
79  !       v ... vector to calculate L1-norm for
80  ! --- Code ---
81     v1abs = SUM(ABS(v))
82     RETURN
83  END FUNCTION v1abs
84
85  FUNCTION mIabs(m)
86  ! Version 1.00, last modified 16-08-2000.
87  ! --- Function and Algorithm ---
88  ! Calculate the Infinity-norm of a given matrix.
89  ! --- Arguments ---
90     IMPLICIT NONE
91     REAL(KIND = dp) :: mIabs
92     REAL(KIND = dp), INTENT(in) :: m(:,:)
93  !       m ... matrix to calculate Infinity-norm for
94  ! --- Code ---
95     mIabs = MAXVAL(ABS(m))
96     RETURN
97  END FUNCTION mIabs
98
99  FUNCTION m1abs(m)
100 ! Version 1.00, last modified 21-10-1998.
101 ! --- Function and Algorithm ---
102 ! Calculate the L1-norm of a given matrix.
103 ! --- Arguments ---
104    IMPLICIT NONE
105    REAL(KIND = dp) :: m1abs
106    REAL(KIND = dp), INTENT(in) :: m(:,:)
107 !       m ... matrix to calculate L1-norm for
108 ! --- Code ---
109    m1abs = MAXVAL(SUM(ABS(m),dim=2))
110    RETURN
111 END FUNCTION m1abs
112
113 FUNCTION m1absinv(m)
114 ! Version 1.00, last modified 21-04-1999.
115 ! --- Function and Algorithm ---
116 ! Calculate the L1-norm of the inverse of a given matrix. This method
117 ! is just for testing to calculate the L1-condition number. It uses
118 ! brute force by calculating the inverse and features no optimizations.
119 ! --- Arguments ---
120    IMPLICIT NONE
121    REAL(KIND = dp) :: m1absinv
122    REAL(KIND = dp), INTENT(in) :: m(:,:)
123    REAL(KIND = dp), ALLOCATABLE :: Aqr(:,:), R(:,:)
124    INTEGER(KIND = i4b), ALLOCATABLE :: luindx(:)
125    INTEGER(KIND = i4b) :: n, i
126    n = SIZE(m,1)
127
128    ALLOCATE(Aqr(n,n),R(n,n),luindx(n))
129    R = m
130    Aqr = 0.0_dp              ! eye(dimp)
```

```
131      DO i = 1, n
132         Aqr(i,i) = 1.0_dp
133      END DO
134      CALL ludcmp(R,luindx)
135      CALL lubksbm(R,luindx,Aqr) ! calculate inverse matrix in Aqr
136      m1absinv = m1abs(Aqr)     ! calculate norm of that matrix
137      DEALLOCATE(luindx, R, Aqr)
138      RETURN
139   END FUNCTION m1absinv
140
141   FUNCTION L1cond(m)
142   ! Version 1.01, last modified 21-04-1999.
143   ! --- Function and Algorithm ---
144   ! Calulate the L1-condition number of a given matrix. This method
145   ! is just for testing. It uses brute force by calculating the inverse
146   ! and features no optimizations.
147   ! --- Arguments ---
148      IMPLICIT NONE
149      REAL(KIND = dp) :: L1cond
150      REAL(KIND = dp), INTENT(in) :: m(:,:)
151      L1cond = m1abs(m)*m1absinv(m)
152      RETURN
153   END FUNCTION L1cond
154
155   !MODULAR SUBROUTINE swap(v1, v2)
156   ! Version 1.00, last modified 10-01-1998.
157   ! --- Function and Algorithm ---
158   ! Swap two arguments. Dependant of the arguments, different versions of
159   ! this method are used. The most common usage is to swap two vectors.
160   ! --- Bibliography ---
161   ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
162   ! 'Numerical Recipes in Fortran 90', Cambridge Univ.Press(1996), p.991.
163   SUBROUTINE swap_i(a,b)
164   INTEGER(KIND = i4b), INTENT(in out) :: a,b
165   INTEGER(KIND = i4b) :: dum
166      dum=a
167      a=b
168      b=dum
169      RETURN
170   END SUBROUTINE swap_i
171   SUBROUTINE swap_d(a,b)
172   REAL(KIND = dp), INTENT(in out) :: a,b
173   REAL(KIND = dp) :: dum
174      dum=a
175      a=b
176      b=dum
177      RETURN
178   END SUBROUTINE swap_d
179   SUBROUTINE swap_dv(a,b)
180   REAL(KIND = dp), DIMENSION(:), INTENT(in out) :: a,b
181   REAL(KIND = dp), DIMENSION(SIZE(a)) :: dum
182      dum=a
183      a=b
184      b=dum
185      RETURN
186   END SUBROUTINE swap_dv
187   SUBROUTINE swap_dm(a,b)
188   COMPLEX(KIND = dp), DIMENSION(:,:), INTENT(in out) :: a,b
189   COMPLEX(KIND = dp), DIMENSION(size(a,1),size(a,2)) :: dum
190      dum=a
191      a=b
192      b=dum
193      RETURN
194   END SUBROUTINE swap_dm
195   SUBROUTINE masked_swap_ds(a,b,mask)
196   REAL(KIND = dp), INTENT(in out) :: a,b
```

```
197   LOGICAL, INTENT(IN) :: mask
198   REAL(KIND = dp) :: swp
199      if (mask) then
200         swp=a
201         a=b
202         b=swp
203      end if
204      RETURN
205   END SUBROUTINE masked_swap_ds
206   SUBROUTINE masked_swap_dv(a,b,mask)
207   REAL(KIND = dp), DIMENSION(:), INTENT(in out) :: a,b
208   LOGICAL, DIMENSION(:), INTENT(IN) :: mask
209   REAL(KIND = dp), DIMENSION(size(a)) :: swp
210      where (mask)
211         swp=a
212         a=b
213         b=swp
214      end where
215      RETURN
216   END SUBROUTINE masked_swap_dv
217   SUBROUTINE masked_swap_dm(a,b,mask)
218   REAL(KIND = dp), DIMENSION(:,:), INTENT(in out) :: a,b
219   LOGICAL, DIMENSION(:,:), INTENT(IN) :: mask
220   REAL(KIND = dp), DIMENSION(size(a,1),size(a,2)) :: swp
221      where (mask)
222         swp=a
223         a=b
224         b=swp
225      end where
226      RETURN
227   END SUBROUTINE masked_swap_dm
228
229   FUNCTION outerprod(v1, v2)
230   ! Version 1.00, last modified 10-01-1998.
231   ! --- Function and Algorithm ---
232   ! Calculate the matrix of the outer product of two given vectors,
233   ! optimised for parallel processing. This method is called by the LU-
234   ! and the QR-decompositions.
235   ! --- Bibliography ---
236   ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
237   ! 'Numerical Recipes in Fortran 90', Cambridge Univ.Press(1996), p.1000.
238   ! --- Arguments ---
239      IMPLICIT NONE
240      REAL(KIND = dp), INTENT(in) :: v1(:), v2(:)
241   !        v1, v2 ... vectors to calculate the outer product with
242      REAL(KIND = dp) :: outerprod(SIZE(v1),SIZE(v2))
243   ! --- Code ---
244      outerprod = SPREAD(v1, dim=2, ncopies=SIZE(v2)) * &
245                  SPREAD(v2, dim=1, ncopies=SIZE(v1))
246      RETURN
247   END FUNCTION outerprod
248
249   SUBROUTINE ludcmp(A, indx)
250   ! Version 1.00, last modified 10-01-1998.
251   ! --- Function and Algorithm ---
252   ! Replace the given matrix A by the LU-decomposition of a rowwise
253   ! permutation of A.
254   ! --- Bibliography ---
255   ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
256   ! 'Numerical Recipes in Fortran 90', Cambridge Univ.Press(1996), p.1016.
257   ! ------------------------------------------------------------------------
258   ! Arguments
259   ! ------------------------------------------------------------------------
260      IMPLICIT NONE
261      REAL(KIND = dp), INTENT(in out) :: A(:,:)
262   !        A ... (in) matrix to decompose (out) LU-decomposition
```

```
263      INTEGER(KIND = i4b), INTENT(out) :: indx(:)
264   !        indx ... row permutation by partial pivoting
265   ! ------------------------------------------------------------------------
266   ! Local variables
267   ! ------------------------------------------------------------------------
268      REAL(KIND = dp) :: vv(SIZE(A,1))
269   !        vv ... implicit scaling of each row
270      INTEGER(KIND = i4b) :: j, n, imax
271   !        j ... loop counter
272   !        n ... dimension of system
273   !        imax ... pivot row
274   ! ------------------------------------------------------------------------
275   ! Code
276   ! ------------------------------------------------------------------------
277      n = SIZE(A,1)
278      IF ((n /= SIZE(A,2)) .OR. (n /= SIZE(indx)) ) THEN
279   !     error in input dimensions (not checked)
280      END IF
281
282      vv = MAXVAL(ABS(A), dim=2)
283      IF ((ANY(vv == 0.0_dp)) .OR. &
284          (ANY(vv < MAXVAL(vv)*EPSILON(1.0_dp))) ) THEN
285         WRITE(*,*) ' singular matrix in LUDCMP '
286         STOP
287      END IF
288      vv = 1.0_dp/vv
289      DO j = 1, n
290         imax = (j-1)+imaxloc(vv(j:n)*ABS(A(j:n,j))) ! find pivot row
291         IF (j /= imax) THEN
292            CALL swap(A(imax,:), A(j,:))
293            vv(imax) = vv(j)
294         END IF
295         indx(j) = imax
296         IF (A(j,j) == 0.0_dp) A(j,j) = TINY(1.0_dp)
297         A(j+1:n,j) = A(j+1:n,j) / A(j,j) ! divide by pivot element
298         A(j+1:n,j+1:n) = A(j+1:n,j+1:n)-outerprod(A(j+1:n,j), A(j,j+1:n))
299      END DO
300      RETURN
301   ! ------------------------------------------------------------------------
302   ! Local subroutines
303   ! ------------------------------------------------------------------------
304      CONTAINS
305
306   FUNCTION imaxloc(v)
307   ! Version 1.00, last modified 10-01-1998.
308   ! --- Function and Algorithm ---
309   ! Return the location of the largest element of a given vector.
310   ! --- Bibliography ---
311   ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
312   ! 'Numerical Recipes in Fortran 90', Cambridge Univ.Press(1996), p.993.
313   ! --- Arguments ---
314      IMPLICIT NONE
315      INTEGER(KIND = i4b) :: imaxloc
316      REAL(KIND = dp), INTENT(in) :: v(:)
317   !        v ... vector to find location
318   ! --- Local variables ---
319      INTEGER(KIND = i4b) :: imax(1)
320   !        imax ... vector of length 1 to use Fortran function MAXLOC
321   ! --- Code ---
322      imax = MAXLOC(v(:))
323      imaxloc = imax(1)
324      RETURN
325   END FUNCTION imaxloc
326   END SUBROUTINE ludcmp
327
328   SUBROUTINE lubksb(A, indx, b)
```

```
329   ! Version 1.00, last modified 10-01-1998.
330   ! --- Function and Algorithm ---
331   ! Solve a linear equation system A*x=b with the LU-decomposition and
332   ! permutation vector at hand (to be called after LUDCMP).
333   ! --- Bibliography ---
334   ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
335   ! 'Numerical Recipes in Fortran 90', Cambridge Univ.Press(1996), p.1017.
336   ! ---------------------------------------------------------------------
337   ! Arguments
338   ! ---------------------------------------------------------------------
339      IMPLICIT NONE
340      REAL(KIND = dp), INTENT(in) :: A(:,:)
341   !       A ... LU-decomposition matrix
342      INTEGER(KIND = i4b), INTENT(in) :: indx(:)
343   !       indx ... row permutation by partial pivoting
344      REAL(KIND = dp), INTENT(in out) :: b(:)
345   !       b ... (in) right side of LES, (out) solution of LES
346   ! ---------------------------------------------------------------------
347   ! Local variables
348   ! ---------------------------------------------------------------------
349      REAL(KIND = dp) :: summ
350   !       summ ... auxiliary data
351      INTEGER(KIND = i4b) :: i, n, ii, ll
352   !       i ... loop counter
353   !       n ... dimension of system
354   !       ii, ll ... auxiliary indices
355   ! ---------------------------------------------------------------------
356   ! Code
357   ! ---------------------------------------------------------------------
358      n = SIZE(A,1)
359   !  IF ((n /= SIZE(A,2)) .OR. (n /= SIZE(indx)) .OR. (n /= SIZE(b))) THEN
360   !     error in input dimensions (not checked)
361   !  END IF
362
363      ii = 0
364      DO i = 1, n
365         ll = indx(i)
366         summ = b(ll)
367         b(ll) = b(i)
368         IF (ii /= 0) THEN
369            summ = summ - DOT_PRODUCT(A(i,ii:i-1), b(ii:i-1))
370         ELSE IF (summ /= 0.0_dp) THEN
371            ii = i
372         END IF
373         b(i) = summ
374      END DO
375      DO i = n,1,-1
376         b(i) = (b(i) - DOT_PRODUCT(A(i,i+1:n), b(i+1:n)))/A(i,i)
377      END DO
378      RETURN
379   END SUBROUTINE lubksb
380
381   SUBROUTINE lubksbm(A, indx, B)
382   ! Version 1.00, last modified 01-10-1998.
383   ! --- Function and Algorithm ---
384   ! Solve a matrix linear equation system A*X=B with the LU-decomposition
385   ! and permutation vector at hand (to be called after LUDCMP). This just
386   ! loops over the collumns of B.
387   ! --- Bibliography ---
388   ! based on "lubksb" W.H.Press,S.A.Teukolsky,W.T.Vetterling,B.P.Flannery
389   ! ---------------------------------------------------------------------
390   ! Arguments
391   ! ---------------------------------------------------------------------
392      IMPLICIT NONE
393      REAL(KIND = dp), INTENT(in) :: A(:,:)
394   !       A ... LU-decomposition matrix
```

```
395      INTEGER(KIND = i4b), INTENT(in) :: indx(:)
396   !        indx ... row permutation by partial pivoting
397      REAL(KIND = dp), INTENT(in out) :: B(:,:)
398   !        b ... (in) right sides of the LES, (out) solution of the LES
399   ! ----------------------------------------------------------------------
400   ! Local variables
401   ! ----------------------------------------------------------------------
402      REAL(KIND = dp) :: summ
403   !        summ ... auxiliary data
404      INTEGER(KIND = i4b) :: i, n, ii, ll, j
405   !        i ... loop counter
406   !        n ... dimension of system
407   !        ii, ll ... auxiliary indices
408   ! ----------------------------------------------------------------------
409   ! Code
410   ! ----------------------------------------------------------------------
411      n = SIZE(A,1)
412   !  IF ((n /= SIZE(A,2)) .OR. (n /= SIZE(indx)) .OR. &
413   !                                                 (n /= SIZE(B,1))) THEN
414   !     error in input dimensions (not checked)
415   !  END IF
416
417      DO j = 1, SIZE(B,2)
418         ii = 0
419         DO i = 1, n
420            ll = indx(i)
421            summ = b(ll,j)
422            b(ll,j) = b(i,j)
423            IF (ii /= 0) THEN
424               summ = summ - DOT_PRODUCT(A(i,ii:i-1), b(ii:i-1,j))
425            ELSE IF (summ /= 0.0_dp) THEN
426               ii = i
427            END IF
428            b(i,j) = summ
429         END DO ! i = 1, n
430         DO i = n,1,-1
431            b(i,j) = (b(i,j) - DOT_PRODUCT(A(i,i+1:n), b(i+1:n,j)))/A(i,i)
432         END DO ! i = 1, n
433      END DO ! j = 1, SIZE(B,2)
434      RETURN
435   END SUBROUTINE lubksbm
436
437   SUBROUTINE qrdcmp(A, c, d, sing)
438   ! Version 1.00, last modified 18-09-1998.
439   ! --- Function and Algorithm ---
440   ! Calculate the QR-decomposition of a given quadratic matrix A.
441   ! --- Bibliography ---
442   ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
443   ! 'Numerical Recipes in Fortran 90', Cambridge Univ.Press(1996), p.1039.
444   ! ----------------------------------------------------------------------
445   ! Arguments
446   ! ----------------------------------------------------------------------
447      IMPLICIT NONE
448      REAL(KIND = dp), INTENT(in out) :: A(:,:)
449   !        A ... (in) matrix to decompose (out) R from QR-decomposition
450      REAL(KIND = dp), INTENT(out) :: c(:), d(:)
451   !        c ... factors of the n-1 Householder matrices building Q
452   !        d ... diagonal elements of R
453      LOGICAL, INTENT(out) :: sing
454   !        sing ... true if singularity is encountered
455   ! ----------------------------------------------------------------------
456   ! Local variables
457   ! ----------------------------------------------------------------------
458      REAL(KIND = dp) :: scale, sigma
459   !        scale ... scale of lower part of a collumn
460   !        sigma ... sign
```

```
461      INTEGER(KIND = i4b) :: j, n
462   !        j ... loop counter
463   !        n ... dimension of system
464   ! ------------------------------------------------------------------
465   ! Code
466   ! ------------------------------------------------------------------
467      n = SIZE(A,1)
468   !  IF ( (n /= SIZE(A,2)) .OR. (n /= SIZE(c)) .OR. (n /= SIZE(d)) ) THEN
469   !     error in input dimensions (not checked)
470   !  END IF
471      sing = .false.
472      c = 0.0_dp
473      d = 0.0_dp
474
475      DO j = 1,n-1
476         scale = MAXVAL(ABS(A(j:n,j)))
477         IF ( (scale == 0.0_dp) .OR. &
478            (scale < MAXVAL(ABS(A))*EPSILON(1.0_dp))) THEN ! singular case
479            sing = .true.
480            c(j) = 0.0_dp
481            d(j) = 0.0_dp
482         ELSE                    ! form Qj and Qj A
483            A(j:n,j) = A(j:n,j) / scale
484            sigma = SIGN(v2abs(A(j:n,j)),A(j,j))
485            A(j,j) = A(j,j) + sigma
486            c(j) = sigma * A(j,j)
487            d(j) = -scale * sigma
488            A(j:n,j+1:n) = A(j:n,j+1:n) - outerprod(A(j:n,j), &
489                           MATMUL(A(j:n,j), A(j:n,j+1:n))) / c(j)
490         END IF
491      END DO
492      d(n) = A(n,n)
493      IF ( (d(n) == 0.0_dp) .OR. &
494           (ABS(d(n)) < MAXVAL(ABS(A))*EPSILON(1.0_dp)) ) sing = .true.
495      RETURN
496   END SUBROUTINE qrdcmp
497
498   SUBROUTINE qrsolv(A, c, d, b)
499   ! Version 1.00, last modified 18-09-1998.
500   ! --- Function and Algorithm ---
501   ! Solve a linear *regular* equation system A*x=b with QR-decomposition
502   ! (to be called after QRDCMP).
503   ! --- Bibliography ---
504   ! W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery,
505   ! 'Numerical Recipes in Fortran 90', Cambridge Univ.Press(1996), p.1040.
506   ! ------------------------------------------------------------------
507   ! Arguments
508   ! ------------------------------------------------------------------
509      IMPLICIT NONE
510      REAL(KIND = dp), INTENT(in) :: A(:,:), c(:), d(:)
511   !        A ... R from QR-decomposition
512   !        c ... factors of the n-1 Householder matrices building Q
513   !        d ... diagonal elements of R
514      REAL(KIND = dp), INTENT(in out) :: b(:)
515   !        b ... (in) right side of LES, (out) solution of LES
516   ! ------------------------------------------------------------------
517   ! Local variables
518   ! ------------------------------------------------------------------
519      REAL(KIND = dp) :: tau
520   !        tau ... auxiliary factor
521      INTEGER(KIND = i4b) :: j, n
522   !        j ... loop counter
523   !        n ... dimension of system
524   ! ------------------------------------------------------------------
525   ! Code
526   ! ------------------------------------------------------------------
```

```
527      n = SIZE(A,1)
528   !  IF ((n /= SIZE(A,2)) .OR. (n /= SIZE(c)) .OR. (n /= SIZE(d)) &
529   !                       .OR. (n /= SIZE(b))) THEN
530   !     error in input dimensions (not checked)
531   !  END IF
532
533      DO j = 1,n-1              ! form, Q^T b
534         tau = DOT_PRODUCT(A(j:n,j), b(j:n)) / c(j)
535         b(j:n) = b(j:n) - tau * A(j:n,j)
536      END DO
537      b(n) = b(n) / d(n)
538      DO j = n-1,1,-1
539         b(j) = (b(j) - DOT_PRODUCT(A(j,j+1:n), b(j+1:n))) / d(j)
540      END DO
541      RETURN
542   END SUBROUTINE qrsolv
543
544   SUBROUTINE qrdecomp(A, Q)
545   ! Version 1.01, last modified 22-09-1998.
546   ! --- Function and Algorithm ---
547   ! Calculate the QR-decomposition of a given nxm matrix A.
548   ! --- Bibliography ---
549   ! based on 'qrdcmp' W.H.Press,S.A.Teukolsky,W.T.Vetterling,B.P.Flannery
550   ! ------------------------------------------------------------------------
551   ! Arguments
552   ! ------------------------------------------------------------------------
553      IMPLICIT NONE
554      REAL(KIND = dp), INTENT(in out) :: A(:,:)
555   !        A ... (in) nxm matrix to decompose (out) nxm matrix R
556      REAL(KIND = dp), INTENT(out) :: Q(SIZE(A,1),SIZE(A,1))
557   !        Q ... nxn matrix Q from QR-decomposition
558   ! ------------------------------------------------------------------------
559   ! Local variables
560   ! ------------------------------------------------------------------------
561      REAL(KIND = dp) :: Qj(SIZE(A,1),SIZE(A,1)), u(SIZE(A,1))
562   !        Qj ... householder matrices
563   !        u ... vector axis of householder matrix
564      REAL(KIND = dp) :: scale, sigma, c
565   !        scale ... scale of lower part of a collumn
566   !        sigma ... sign
567   !        c ... factor of the Householder matrix building Q
568      INTEGER(KIND = i4b) :: i, j, n, m, k
569   !        i, j ... loop counter
570   !        n, m, k ... dimension of system
571   ! ------------------------------------------------------------------------
572   ! Code
573   ! ------------------------------------------------------------------------
574      n = SIZE(A,1)
575      m = SIZE(A,2)
576      k = MIN(n-1,m)
577      Q = 0.0_dp                ! set Q as product of Householder matrices
578      DO i = 1,n
579         Q(i,i) = 1.0_dp
580      END DO
581
582      DO j = 1,k
583         scale = MAXVAL(ABS(A(j:n,j)))
584   !      IF ( (scale == 0.0_dp) .OR. &
585   !         (scale < MAXVAL(ABS(A))*EPSILON(1.0_dp))) THEN ! singular case
586         IF (scale == 0.0_dp) THEN ! singular case
587            A(j:n,j) = 0.0_dp   ! calculate R
588         ELSE
589            IF (j > 1) u(1:j-1) = 0.0_dp ! axis for Householder matrix
590            u(j:n) = A(j:n,j) / scale
591            sigma = SIGN(v2abs(u(j:n)),u(j))
592            u(j) = u(j) + sigma
```

```
593           c = sigma * u(j)
594
595           Qj = 0.0_dp         ! generate Q_j
596           DO i = 1,n
597              Qj(i,i) = 1.0_dp
598           END DO
599           Qj(j:n,j:n) = Qj(j:n,j:n) - outerprod(u(j:n),u(j:n)) / c
600           Q = MATMUL(Q,Qj)    ! accumulate Q_j into Q
601
602           A(j:n,j+1:m) = A(j:n,j+1:m) - outerprod(u(j:n), &
603                          MATMUL(u(j:n), A(j:n,j+1:m))) / c ! update A
604           A(j+1:n,j) = 0.0_dp ! calculate R
605           A(j,j) = -scale * sigma
606        END IF
607     END DO
608  !  IF (k == n-1) A(n,n) = A(n,n)
609     RETURN
610  END SUBROUTINE qrdecomp
611
612  END MODULE linalg
```

```
1   ! ------------------------------------------------------------------------
2   ! This is part of the 'Solve Singular BVP' (SOLVSING) package developed
3   ! by W.Auzinger, O.Koch, P.Kofler, E.Weinmueller,
4   ! Department of Applied Mathematics and Numerical Analysis,
5   ! Vienna University of Technology, Vienna, Austria.
6   ! ------------------------------------------------------------------------
7   MODULE types
8   ! --- Function and Algorithm ---
9   ! This module provides just the kind (type) constants used throughout
10  ! the entire package. Default precision is double precision. To change
11  ! the precision replace 1.0D0 by 1.0 for single or 1.0Q0 for quadruple
12  ! precision. All further variables are defined as "REAL(KIND = dp)", all
13  ! constants are defined as "1.0_dp".
14  ! --- Compiling ---
15  ! This module has to be compiled first. For the usage see t_types.f90.
16  ! ------------------------------------------------------------------------
17  ! Constants
18  ! ------------------------------------------------------------------------
19     IMPLICIT NONE
20     INTEGER, PARAMETER :: dp = KIND(1.0D0), &
21                           i4b = SELECTED_INT_KIND(9)
22  !        dp ... working precision real kind number
23  !        i4b ... working precision integer kind number
24     LOGICAL, PARAMETER :: debug_output = .false.
25  !        debug_output ... do some verbose text output what is going on
26
27  END MODULE types
```

```
1   ! ------------------------------------------------------------------------
2   ! This is part of the 'Solve Singular BVP' (SOLVSING) package developed
3   ! by W.Auzinger, O.Koch, P.Kofler, E.Weinmueller,
4   ! Department of Applied Mathematics and Numerical Analysis,
5   ! Vienna University of Technology, Vienna, Austria.
6   ! ------------------------------------------------------------------------
7   MODULE dynvect
8   ! 'DYNamic expanding VECTor' (DYNVECT)
9   ! Version 1.01, last modified 16-09-2000.
10  ! --- Function and Algorithm ---
11  ! This module provides dynamic vectors similar the Java implementation
12  ! of the class 'java.util.Vector' (1.62 99/04/22), which is Copyright
13  ! by Sun Microsystems, Inc., U.S.A. All rights reserved. But the
14  ! functionality is less than in Java, only resetting, adding and copying
15  ! to an array is supplied. As needed for RKPair, dynamic storage for a
16  ! vector and a real scalar is implemented.
17  ! --- Compiling ---
```

```
18 ! This module has to be compiled second, following the types module.
19 ! --- Bibliography ---
20 ! M.Metcalf, J.Reid,
21 ! 'Fortran 90 Explained', Oxford University Press (1990),
22 ! L.Boynton, J.Payne, 'java.util.Vector.java', JDK1.22, Sun.
23 ! ------------------------------------------------------------------------
24 ! Constants
25 ! ------------------------------------------------------------------------
26    USE TYPES                 ! predefined data-types
27    IMPLICIT NONE
28    INTEGER(KIND = i4b), PARAMETER :: vsize_init = 100, &
29                                      vsize_mult = 2
30 !       vsize_init ... initial size of the dynamic arrays
31 !       vsize_mult ... multiplier for the size if it has to grow
32
33 ! ------------------------------------------------------------------------
34 ! Global variables
35 ! ------------------------------------------------------------------------
36    REAL(KIND = dp), POINTER :: vectors(:,:), scalars(:)
37 !       vectors ... array for accumulating vectors
38 !       scalars ... array for accumulating scalars
39    INTEGER(KIND = i4b) :: vsize, vsize_max
40 !       vsize ... size of the used fields of vectors and scalars
41 !       vsize_max ... size of complete array of vectors and scalars
42
43    PRIVATE
44    PUBLIC :: dv_init, dv_capacity, dv_size, dv_addelement, dv_toarray, &
45              dv_removeallelements, dv_release
46
47 ! ------------------------------------------------------------------------
48 ! Subroutines
49 ! ------------------------------------------------------------------------
50 CONTAINS
51
52 SUBROUTINE dv_init(dim)
53 ! --- Function and Algorithm ---
54 ! Set up the data for the dynamic vector to the specified dimension.
55 ! This functionality is similar to Java's 'new java.util.Vector()'.
56 ! --- Arguments ---
57    IMPLICIT NONE
58    INTEGER(KIND = i4b), INTENT(in) :: dim
59 !       dim ... dimension of the vectors to be stored in vectors.
60 ! --- Code ---
61    IF (ASSOCIATED(vectors)) DEALLOCATE(vectors) ! drop old values
62    IF (ASSOCIATED(scalars)) DEALLOCATE(scalars) ! drop old values
63    vsize_max = vsize_init
64    vsize = 0             ! there are no values stored
65    ALLOCATE(vectors(0:vsize_max-1,dim))
66    ALLOCATE(scalars(0:vsize_max-1))
67    vectors = 0.0_dp
68    scalars = 0.0_dp
69    RETURN
70 END SUBROUTINE dv_init
71
72 FUNCTION dv_capacity()
73 ! --- Function and Algorithm ---
74 ! Returns the current capacity of this vector, i.e. the length of its
75 ! internal data arary.
76 ! --- Arguments ---
77    IMPLICIT NONE
78    INTEGER(KIND = i4b) :: dv_capacity
79 ! --- Code ---
80    dv_capacity = vsize_max
81    RETURN
82 END FUNCTION dv_capacity
83
```

```
84  FUNCTION dv_size()
85  ! --- Function and Algorithm ---
86  ! Returns the number of components in this vector.
87  ! --- Arguments ---
88     IMPLICIT NONE
89     INTEGER(KIND = i4b) :: dv_size
90  ! --- Code ---
91     dv_size = vsize
92     RETURN
93  END FUNCTION dv_size
94
95  SUBROUTINE dv_addelement(x,y)
96  ! --- Function and Algorithm ---
97  ! Adds a scalar and a vector-value to the end of this dynamic vector
98  ! increasing its size by one.. If the array is too small to hold the
99  ! next value, it is incresed.
100 ! --- Arguments ---
101    IMPLICIT NONE
102    REAL(KIND = dp), INTENT(in) :: x, y(:)
103 !        x ...  the scalar to store next in scalars
104 !        y ... the value to store next in vectors
105 ! --- Local variables ---
106    REAL(KIND = dp), POINTER :: temps(:), tempv(:,:)
107 !        temp ... temporary for holding data while doubling the arrays
108    INTEGER(KIND = i4b) :: new_size
109 !        new_size ... new (incresed) size of the dynamic vector
110 ! --- Code ---
111    IF (vsize >= vsize_max) THEN ! no space is left in dynamic vector
112       new_size = vsize_max*vsize_mult
113
114       ! increse the space of vectors
115       ALLOCATE(tempv(0:vsize_max-1,SIZE(y))) ! create temp storage
116       tempv = vectors           ! copy values to temporary storage
117       DEALLOCATE(vectors)       ! drop small storage
118       ALLOCATE(vectors(0:new_size-1,SIZE(y))) ! allocate more
119       vectors(0:vsize_max-1,:) = tempv ! copy values from temp storage
120       vectors(vsize_max:new_size-1,:) = 0.0_dp
121       DEALLOCATE(tempv)      ! drop temporary storage
122
123       ! increse the space of scalars
124       ALLOCATE(temps(0:vsize_max-1)) ! create temp storage
125       temps = scalars        ! copy values to temporary storage
126       DEALLOCATE(scalars)    ! drop small storage
127       ALLOCATE(scalars(0:new_size-1)) ! allocate more
128       scalars(0:vsize_max-1) = temps ! copy values from temp storage
129       scalars(vsize_max:new_size-1) = 0.0_dp
130       DEALLOCATE(temps)      ! drop temporary storage
131
132       vsize_max = new_size   ! we allocated more storage
133    END IF
134    vectors(vsize,:) = y
135    scalars(vsize) = x
136    vsize = vsize + 1         ! we added a new element to the next column
137    RETURN
138 END SUBROUTINE dv_addelement
139
140 SUBROUTINE dv_toarray(grid,solution)
141 ! --- Function and Algorithm ---
142 ! Returns the arrays containing scalars and vectors all of the elements
143 ! in this dynamic Vector in the correct order and in its exact size.
144 ! --- Arguments ---
145    IMPLICIT NONE
146    REAL(KIND = dp), INTENT(out) :: grid(:)
147    REAL(KIND = dp), INTENT(out) :: solution(:,:)
148 !        grid ... the field where the scalars are put
149 !        solution ... the field where the vectors are put
```

```
150  ! --- Code ---
151     ! no range check is done here !
152     grid = scalars(0:vsize-1)
153     solution = vectors(0:vsize-1,:)
154     RETURN
155  END SUBROUTINE dv_toarray
156
157  SUBROUTINE dv_removeallelements()
158  ! --- Function and Algorithm ---
159  ! Removes all of the elements from this Vector. The Vector will be
160  ! empty after this call returns
161  ! --- Arguments ---
162     IMPLICIT NONE
163  ! --- Code ---
164     vsize = 0                ! there are no values stored
165     vectors = 0.0_dp
166     scalars = 0.0_dp
167     RETURN
168  END SUBROUTINE dv_removeallelements
169
170  SUBROUTINE dv_release()
171  ! --- Function and Algorithm ---
172  ! return the used allocated arrays to the system.
173  ! --- Arguments ---
174     IMPLICIT NONE
175  ! --- Code ---
176     IF (ASSOCIATED(vectors)) DEALLOCATE(vectors) ! drop old values
177     IF (ASSOCIATED(scalars)) DEALLOCATE(scalars) ! drop old values
178     vsize_max = 0
179     vsize = 0
180     RETURN
181  END SUBROUTINE dv_release
182
183  END MODULE dynvect
```