The Implementation of SIRK Methods for Differential Algebraic Equations

Erik Hamran Nilsen

February 19, 1997

Department of Mathematical Sciences
The Norwegian University of Science and Technology

Abstract

The aim of this report is to survey some aspects of implicit differential equations, differential algebraic equations, and their numerical solution. We will formulate an algorithm to solve implicitly defined differential systems by singly implicit Runge-Kutta methods, and see how estimates of the local error can be computed. Then we will give some convergence results for Runge-Kutta methods applied to differential algebraic systems.

As a part of the project we have implemented an experimental version of the stiff ODE solver STRIDE by J. C. Butcher, K. Burrage and F. H. Chipman. The new code is capable of solving systems in implicit form and some differential algebraic systems. We have used the code to verify the convergence results and the local error estimates for SIRK methods. Furthermore we have tested the experimental code on some real applications.

Preface

This report is a part of my studies at the Department of Mathematical Sciences, the Norwegian University of Technology and Science in Trondheim, Norway. All the implementation of computer programs, and the writing of this report, has been done at the University of Auckland, New Zealand.

I am very grateful to prof. John C. Butcher, who gladly accepted to supervise my work and arrange the practical details for my stay in Auckland. I would also like to thank Anne Kværnø who has been my supervisor in Norway, and Syvert P. Nørsett who introduced me to John. Special thanks go to all the members of the "Runge-Kutta club" in Auckland, for their help and advice, and for the weekly workshops and lunches.

Erik Hamran Nilsen, Auckland, February 19, 1997

Contents

Chapter 1

Introduction

Numerical solution of differential algebraic equations is a relatively new field within mathematics. Only a few existing numerical solvers are capable of handling such systems, even though differential algebraic equations arise naturally in many applications. The aim of this project has been to implement an experimental solver of differential algebraic equations, based on *singly implicit Runge-Kutta* methods.

In this chapter we will give a brief description of relevant problems, and then some basic theoretical aspects of numerical methods and stability follows. In the next chapter we will explain details about singly implicit Runge-Kutta methods and their implementation, especially for implicitly defined problems. We will also show how an estimate of the local error can be calculated. Chapter 3 gives convergence results for Runge-Kutta methods applied to differential algebraic equations, and discusses some difficulties that arise when we want to estimate the local error for these problems. Then we will give some numerical results in chapter 4. We have verified the convergence results for SIRK methods applied to differential algebraic equations, and tested the error estimates we found in chapter 2. We have also given some test results for SIRK methods applied to real applications. Chapter 5 contains conclusions and discussions of the report and the implementation we have done.

1.1 Differential Algebraic Equations

Consider the general system of first order differential equations

$$\Phi(x'(t), x(t), t) = 0, \tag{1.1}$$

This system is generally not equivalent to an ordinary differential equation (ODE)

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

We can achieve this form only if $\partial\Phi/\partial x'$ is non-singular (implicit function theorem). Otherwise we will call equation (1.1) a *differential algebraic equation*

(DAE).

We use the name *differential algebraic* because many DAEs can be split into two parts:

$$\begin{aligned} x' = f(x,y,t) &= 0, && (1.2)\\ G(x,y,t) &= 0, && (1.3)\end{aligned}$$

where algebraic constraints appear explicitly. The DAE can then be regarded as a vector-field on the manifold

$$S := \{[x,y,t] : G(x,y,t) = 0\}.$$

Generally, the algebraic constraints can not be written explicitly as in (1.3).

Differential algebraic equations are usually harder to solve than ODEs, and a measure of the numerical difficulties are given by the *index* of the system.

1.1.1 The Differential Index

To motivate the definition of differential index, we first consider the linear constant coefficient DAE

$$Ax' + Bx = q(t). \tag{1.4}$$

We will use the ideas of [10] to analyze this system.

We define the *regular matrix pencil* as the ordered pair of matrices $\{A, B\}$, where the polynomial $p(\lambda) = \det(\lambda A + B)$ does not vanish identically.

The following theorem by Weierstrass makes it possible to transform the above equation to a simpler form:

Theorem 1.1 *Let $\{A, B\}$ be a regular matrix pencil. Then $\{A, B\}$ can be transformed by the regular matrices E and F into $\{\widetilde{A}, \widetilde{B}\}$, where*

$$\begin{aligned} \widetilde{A} &= EAF = \begin{pmatrix} I & 0 \\ 0 & N \end{pmatrix}, \\ \widetilde{B} &= EBF = \begin{pmatrix} W & 0 \\ 0 & I \end{pmatrix}. \end{aligned}$$

The matrix N is a block diagonal matrix $N = \operatorname{diag}(N_1, \ldots, N_K)$, where each N_i is of the form

$$N_i = \begin{pmatrix} 0 & 1 & & \\ & 0 & \ddots & \\ & & \ddots & 1 \\ & & & 0 \end{pmatrix}$$

and W can be assumed to be in Jordan canonical form.

If we change coordinates to $\widetilde{x}(t) = F^{-1}x(t)$, we can write equation (1.4) as

$$\widetilde{A}\widetilde{x}'(t) + \widetilde{B}\widetilde{x}(t) = \widetilde{q}(t) \tag{1.5}$$

where $\widetilde{q}(t) = Eq(t)$.

The system (1.5) is now decoupled into

$$u'(t) + Wu(t) = p(t), \tag{1.6}$$
$$Nv'(t) + v(t) = r(t). \tag{1.7}$$

This system is called the *Kronecker normal form* of equation (1.4). The first part is an ordinary differential equation, and the second part decouples into k smaller systems

$$N_i v_i'(t) + v_i(t) = r_i(t), \qquad i = 1, \ldots, k. \tag{1.8}$$

Each of the systems is of the form (with dimension $m = m_i$)

$$\left.\begin{array}{rcl} z_2' + z_1 &=& d_1(t) \\ &\vdots& \\ z_m' + z_{m-1} &=& d_{m-1}(t) \\ z_m &=& d_m(t) \end{array}\right\}. \tag{1.9}$$

The last equation defines z_m, and the other components can be obtained recursively by insertion and differentiation. Then z_1 will depend on the $(m-1)$-fold derivative of z_m. Since differentiation is an unstable numerical procedure, the largest m_i is a measure of the numerical difficulty of solving the problem. The largest m_i is said to be the index of problem (1.4).

The above definition of index can be extended to non-linear problems. Following [1] we consider the general non-linear DAE

$$\Phi(x'(t), x(t), t) = 0 \tag{1.10}$$

We assume that f is sufficiently smooth and define the system

$$\left.\begin{array}{rcl} \Phi(x', x, t) &=& 0 \\ \dfrac{d}{dt}\Phi(x', x, t) &=& 0 \\ &\vdots& \\ \dfrac{d^m}{dt^m}\Phi(x', x, t) &=& 0 \end{array}\right\}. \tag{1.11}$$

Definition 1.2 *The* differential index *of the DAE (1.10) is the smallest integer m, such that the system (1.11) can be solved for x' (if such an integer m exists).*

Example 1.1 *Consider a DAE on the form*

$$\begin{array}{rcl} u' &=& f(u, v), \\ 0 &=& g(u, v), \end{array}$$

where $g_v(u,v)$ is non-singular. We take the first derivative of the constraint equation

$$\frac{d}{dt}g(u,v) = g_u u' + g_v v' = 0$$

and the system can be written as an explicit ODE

$$\begin{aligned} u' &= f(u,v), \\ v' &= g_v^{-1}(u,v)g_u(u,v)f(u,v). \end{aligned}$$

We see that following the above procedure, it is possible to reduce a DAE to an explicit ODE. But a solution to this ODE is not necessarily a solution to the original DAE. There are no algebraic constraints to keep the solution on a certain manifold, and numerical solutions will typically drift away.

1.1.2 The Perturbation Index

In [9] another definition of index is introduced. The idea is to measure the sensitivity of the solution with respect to perturbations of the problem.

Definition 1.3 (Def. 1.1 in [9]) *Equation (1.1) has perturbation index m_p along a solution $x(t)$ on $[0,\bar{t}\,]$ if m_p is the smallest integer such that, for all functions $\widehat{x}(t)$ having a defect*

$$\Phi(\widehat{x}', \widehat{x}, t) = \delta(t),$$

there exists on $[0,\bar{t}\,]$ an estimate

$$||\widehat{x}(t) - x(t)|| \leq C\left(||\widehat{x}(0) - x(0)|| + \max_{0\leq\xi\leq t}||\delta(\xi)|| + \ldots + \max_{0\leq\xi\leq t}||\delta^{(m_p-1)}(\xi)||\right)$$

whenever $\delta(t)$ is sufficiently small.

Considering each component of x individually, it is possible to talk about the index of a single component.

The definition is valid also for index 0, if we interpret $\delta^{(-1)}(\xi)$ as the integral

$$\int_0^\xi \delta(u)\mathrm{d}u.$$

In other words, the system (1.1) has perturbation index 0 if

$$||\widehat{x}(t) - x(t)|| \leq C\left(||\widehat{x}(0) - x(0)|| + \max_{0\leq\xi\leq t}||\int_0^\xi \delta(u)\mathrm{d}u||\right).$$

It is easy to see that for the linear constant coefficient DAE (1.4), the perturbation index coincides with the differential index. The system (1.9) gives

$$\left.\begin{array}{rcl} \Delta z_2' + \Delta z_1 & = & \delta_1(t) \\ & \vdots & \\ \Delta z_m' + \Delta z_{m-1} & = & \delta_{m-1}(t) \\ \Delta z_m & = & \delta_m(t) \end{array}\right\} \tag{1.12}$$

where $\Delta z_i = \widehat{z}_i - z_i$, and δ_j, $j = 1, \ldots, m$, are components of the defect $\delta(t)$. Here z_1 has the highest index $(m-1)$, because

$$\begin{array}{rcl} ||\Delta z_1(t)|| & = & ||\delta_1(t) - \delta_2^{(1)}(t) + \ldots (-1)^{m-1}\delta_m^{(m-1)}(t)|| \\ & \leq & ||\delta_1(t)|| + ||\delta_2^{(1)}(t)|| + \ldots + ||\delta_m^{(m-1)}(t)|| \\ & \leq & \max\limits_{0\leq\xi\leq t} ||\delta_1(\xi)|| + \max\limits_{0\leq\xi\leq t} ||\delta_2^{(1)}(\xi)|| + \ldots + \max\limits_{0\leq\xi\leq t} ||\delta_m^{(m-1)}(\xi)||. \end{array}$$

It was believed for some time that the differential and perturbation indices can differ at most by 1, but Campbell and Gear found a counter-example in 1995. (See [8] p. 461.)

1.2 Singular Perturbation Problems

There is a close relationship between differential algebraic equations and certain kind of stiff ordinary differential equations called *singularly perturbed problems*. Following [9] we consider the system

$$\begin{array}{rcl} y' & = & f(y,z), \\ \epsilon z' & = & g(y,z), \end{array} \tag{1.13}$$

where $\epsilon > 0$ is a small parameter. If certain conditions apply, the solution is known to possess an ϵ-expansion on any fixed interval, outside an initial transient phase.

$$\begin{array}{rcl} y(t) & = & y_0(t) + \epsilon y_1(t) + \epsilon^2 y_2(t) + \ldots + \epsilon^N y_N(t) + \mathcal{O}(\epsilon^{N+1}), \\ z(t) & = & z_0(t) + \epsilon z_1(t) + \epsilon^2 z_2(t) + \ldots + \epsilon^N z_N(t) + \mathcal{O}(\epsilon^{N+1}). \end{array} \tag{1.14}$$

Inserting (1.14) into (1.13), and comparing powers of ϵ leads to the systems

$$\begin{array}{rcl} y_0' & = & f(y_0, z_0), \\ 0 & = & g(y_0, z_0), \end{array}$$

for first order approximations, and

$$\begin{array}{rcl} y_1' & = & f_z(y_0, z_0)y_1 + f_z(y_0, z_0)z_1, \\ z_0' & = & g_y(y_0, z_0)y_1 + g_z(y_0, z_0)z_1, \end{array}$$

for second order approximations. The first of these systems is an index 1 DAE, and so is the second if we assume that y_0 and z_0 have already been computed. But the two systems together form an index 2 DAE. Continuing to higher orders of ϵ we can obtain systems of arbitrary high index.

1.3 Numerical Methods

We will from now on assume that the reader has some knowledge about methods for solving ordinary differential equations. This section contains only a brief description of some methods, and defines the notation and some results we will need later. Several textbooks on numerical methods for ODEs are available, for example [5], [4], [7] and [8].

1.3.1 Runge-Kutta Methods

Consider the initial value problem

$$y' = f(y,t), \qquad y(t_0) = y_0, \tag{1.15}$$

and assume that we know an approximation y_n to the solution $y(t)$ at the point $t = t_n$. We can apply an s-stage implicit Runge-Kutta method to obtain an approximation y_{n+1} at $t_{n+1} = t_n + h$,

$$Y_i = y_n + h \sum_{j=1}^{s} a_{ij} Y_j' \ , \quad i = 1, \ldots, s, \tag{1.16}$$

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i Y_i' \ . \tag{1.17}$$

Here we have

$$Y_i' = f(Y_i, t_n + c_i h) \ , \quad i = 1, \ldots, s \ . \tag{1.18}$$

For every time-step we have to solve a nonlinear system of algebraic equations.

The RK-method is characterized by an $s \times s$ matrix A, and two vectors b and c, which we arrange in a tableau:

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array} = \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array} \tag{1.19}$$

The order of the method applied to (1.15) is p if the difference between the exact and the numerical solution is

$$y(t_0 + h) - y_1 = \mathcal{O}(h^{p+1}).$$

An important tool for construction of higher order implicit Runge-Kutta methods are the *simplifying assumptions*.

Definition 1.4 *The* simplifying assumptions $B(p)$, $C(\eta)$ *and* $D(\xi)$ *are defined by:*

$$\begin{aligned}
B(p) \quad &: \quad \sum_{i=1}^{s} b_i c_i^{q-1} = \frac{1}{q} \qquad q = 1, \dots, p, \\
C(\eta) \quad &: \quad \sum_{j=1}^{s} a_{ij} c_j^{q-1} = \frac{c_i^q}{q} \qquad i = 1, \dots, s, \quad q = 1, \dots, \eta, \\
D(\xi) \quad &: \quad \sum_{i=1}^{s} b_i c_i^{q-1} a_{ij} = \frac{b_j}{q}(1 - c_j^q) \qquad j = 1, \dots, s, \quad q = 1, \dots, \xi.
\end{aligned}$$

The formulations of $B(p)$ and $C(\eta)$ are equivalent to results from integration theory. The $B(p)$ condition can be formulated as

$$\sum_{j=1}^{s} b_j P(c_j) = \int_0^1 P(c)\mathrm{d}c \ , \tag{1.20}$$

for any polynomial P of degree less than p, and $C(\eta)$ is equivalent to the condition

$$\sum_{j=1}^{s} a_{ij} P(c_j) = \int_0^{c_i} P(c)\mathrm{d}c \ , \quad i = 1, \dots, s \ , \tag{1.21}$$

for any polynomial P of degree less than η.

The following theorem connects the simplifying assumptions to the order of a Runge-Kutta method and is proven in [7].

Theorem 1.5 (Butcher 1964) *If the coefficients* b_i*,* c_i*,* a_{ij} *of an RK-method satisfy* $B(p)$*,* $C(\eta)$ *and* $D(\xi)$ *with* $p \leq \eta + \xi + 1$ *and* $p \leq 2\eta + 2$*, then the method is of order* p*.*

Stability

If we apply a Runge-Kutta method to the test equation

$$y' = \lambda y \ , \quad y(0) = y_0,$$

the solution y_1 at $t = h$ can be written as

$$y_1 = R(\lambda h) y_0.$$

The function $R(\lambda h)$ is called the *stability function* of the method, and the region where $|R(\lambda h)| < 1$ is called the *stability region* ($\lambda h = z$ is a number in the complex plane). A method is A-stable if the stability region includes the negative half plane, $\mathbb{C}^- = \{z = x + iy; \quad x < 0\}$. If we in addition have

$$\lim_{z \to \infty} R(z) = 0$$

we will call the method L-stable. A weaker condition than A-stability is often useful, and we say that a method is $A(\alpha)$-stable if the sector

$$S_\alpha = \{z; \quad |\arg(-z)| \leq \alpha, \quad z \neq 0\}$$

is contained in the stability region.

For Runge-Kutta methods $R(z)$ is a rational function, given by the following theorem.

Theorem 1.6 *The Runge-Kutta method given by (1.19) has stability function*

$$R(z) = 1 + zb^T(I - zA)^{-1}e \ ,$$

where

$$e = [1, \ldots, 1]^T \, .$$

Proof. When $f(y,t) = \lambda y$ (1.17) reads

$$y_1 = y_0 + h\lambda b^T[Y_1, \ldots, Y_s]^T,$$

and (1.16) gives

$$[Y_1, \ldots, Y_s]^T = y_0 e + h\lambda A[Y_1, \ldots, Y_s]^T.$$

If we Solve the last equation for $[Y_1, \ldots, Y_s]^T$, and insert it into the first, we obtain the wanted result. □

By applying Cramer's rule to the linear system, it is possible to write the stability function as

$$R(z) = \frac{\det(I - zA + zeb^T)}{\det(I - zA)}. \tag{1.22}$$

1.3.2 Multistep Methods

A general multistep method applied to the system $y' = f(y,t)$ has the form

$$\sum_{i=0}^{k} \alpha_i y_{n-i} = h \sum_{i=0}^{k} \beta_i f_{n-i},$$

where an approximation y_n of the exact solution $y(t_n)$ is computed from numerical approximations from previous steps, $y_{n-1}, \ldots, y_{n-k}$ and the derivatives $f_{n-i} = f(y_{n-i}, t_{n-i})$.

The most popular multistep method for stiff ODEs, is the backward differentiation formula (BDF), where $\beta_1 = \ldots = \beta_k = 0$ and $\beta_0 \neq 0$. This method can be written

$$\sum_{i=0}^{k} \alpha_i y_{n-i} = hf_n,$$

where we have to find the coefficients $\alpha_0, \ldots, \alpha_k$. The idea is to interpolate $y_{n-k}, \ldots, y_n$ by a polynomial q of degree k, and in addition require that the polynomial should satisfy the differential equation at $t = t_n$. For $k = 1$ we have

$$q(t_{n-1} + rh) = y_{n-1} + (y_n - y_{n-1})r.$$

Inserting $q(t_n)$ into $y' = f(y,t)$ we get

$$q'(t_{n-1} + h) = y_n - y_{n-1} = f(y_n, t_n),$$

which is the backward Euler formula. Similar formulas can be obtained for any value of k (see [7]).

We can easily adapt the method to equations of the form $\Phi(y', y, t) = 0$ by

$$\Phi\left(\frac{1}{h}\sum_{i=0}^{k} \alpha_i y_{n-i}, y_n, t_n\right) = 0.$$

For $k = 1, 2$ the BDF formulas are A-stable, and for $k = 3, \ldots, 6$ the methods are $A(\alpha)$ stable, but α is quite small for $k = 5, 6$. The BDF formulas corresponding to $k > 6$ are unstable.

Implementation of efficient variable step-size codes is difficult, but several good BDF codes are already available. Still, only a few are capable of solving equations on implicit form and DAEs. A limitation of BDF is that the first step always has to be a backward Euler step, since no previous approximations are available. For many index 3 problems, backward Euler will diverge. A possible way to overcome this problem is to use a Runge-Kutta method as a starting method.

Chapter 2

Singly Implicit Runge-Kutta Methods

2.1 Reducing the Cost

For each new step to be calculated with an s-stage implicit Runge-Kutta method, it is necessary to solve a nonlinear algebraic system of size $s \times m$, where m is the number of unknown variables. We will in this chapter show how a special family of Runge-Kutta methods can be implemented at a lower cost. We will mainly follow the work of Burrage [2] and Butcher [6]. We assume that modified Newton-Raphson iterations are used to solve the nonlinear system.

2.1.1 The Nonlinear System

Using tensor products and a new variable $Z_i = Y_i - y_n$, we can write an implicit RK-method as

$$\begin{aligned} Z &= h(A \otimes I_m)F, \\ y_{n+1} &= y_n + h(b^T \otimes I_m)F, \end{aligned}$$

where

$$Z = \begin{bmatrix} Z_1 \\ \vdots \\ Z_s \end{bmatrix}, \qquad F = \begin{bmatrix} F_1 \\ \vdots \\ F_s \end{bmatrix}.$$

If the differential equation is given as (1.15), we have

$$F_i = f(y_n + Z_i),$$

and we get the nonlinear system

$$Z - h(A \otimes I_m) \begin{bmatrix} f(y_n + Z_1) \\ \vdots \\ f(y_n + Z_s) \end{bmatrix} = 0$$

of $s \times m$ equations to obtain the stage values Z for each integration step. For simplicity we have assumed that the differential system is autonomous. Usually we apply Newton-Raphson iterations to solve the system, and if we update Z by $Z - \delta$ for each iteration, we get the linear system

$$\begin{bmatrix} I_m - ha_{11}J_1 & \cdots & -ha_{1s}J_s \\ \vdots & & \vdots \\ -ha_{s1}J_1 & \cdots & I_m - ha_{ss}J_s \end{bmatrix} \delta = Z - h(A \otimes I_m) \begin{bmatrix} f(y_n + Z_1) \\ \vdots \\ f(y_n + Z_s) \end{bmatrix}, \tag{2.1}$$

where $J_i = f'(y_n + Z_i)$. Jacobian evaluations and factorizations are costly, so in practice we approximate the J_i's by a single Jacobian J, which we keep constant over several steps, as long as the rate of convergence is acceptable.

2.1.2 Singly Diagonal Implicit Methods

If A is a lower triangular matrix the system above can be solved at a considerably lower cost. Instead of factorizing an $s \times m$ matrix, we can now divide the linear system into s parts, each of size m. If we further assume that A has a single s-fold eigenvalue λ, all diagonal elements will be equal, and the same factorization can be used for all s systems. The system above then becomes

$$\begin{bmatrix} I_m - h\lambda J & 0 & \cdots & 0 \\ -ha_{21}J & I_m - h\lambda J & & \vdots \\ \vdots & & \ddots & 0 \\ -ha_{s1}J & \cdots & -ha_{s,s-1}J & I_m - h\lambda J \end{bmatrix} \delta = Z - h(A \otimes I_m) \begin{bmatrix} f(y_n + Z_1) \\ f(y_n + Z_2) \\ \vdots \\ f(y_n + Z_s) \end{bmatrix}.$$

Systematic studies of these methods can be found in [11] and [12]. We will refer to the methods as *singly diagonal implicit Runge-Kutta methods* (SDIRK). Implementation of fully implicit Runge-Kutta methods requires $s^3m^3/3$ operations for the LU-factorization, since the size of the matrix is $s \times m$. The cost of back-substitution is s^2m^2. For SDIRK methods the cost is reduced significantly, especially when s is large. Factorizing one matrix of size m requires $m^3/3$ operations and performing s back-substitutions, another sm^2 operations. But there are some drawbacks. The maximum order for SDIRK methods is $s+1$, and the stage order is only 1, because the first stage is a backward Euler step. Furthermore it is difficult to construct high order methods.

2.2 Singly Implicit Methods

Singly implicit methods (SIRK) are fully implicit methods, where the matrix A has only one single eigenvalue λ. The advantage of these methods is that A can be transformed to Jordan canonical form by the similar transformation

$$T^{-1}AT = \widetilde{A} = \lambda \begin{bmatrix} 1 & & & 0 \\ -1 & 1 & & \\ & \ddots & \ddots & \\ 0 & & -1 & 1 \end{bmatrix}. \tag{2.2}$$

We can exploit this property by transforming the variables in (2.1). Let $\widehat{X}$ denote the transformed value of X,

$$\widehat{X} = (T^{-1} \otimes I)X \tag{2.3}$$

where X can be replaced by any of the variables Y, Z, F and δ. Inserting transformed variables for Z, F and δ in (2.1) and multiplying by $(T^{-1} \otimes I_m)$ gives

$$\begin{aligned}(T^{-1} \otimes I_m)\left[(I_s \otimes I_m) - h(A \otimes J)\right](T \otimes I_m)\widehat{\delta} \\ = (T^{-1} \otimes I_m)\left[(T \otimes I_m)\widehat{Z} - h(A \otimes I_m)(T \otimes I_m)\widehat{F}\right],\end{aligned}$$

which simplifies to

$$\left[(I_s \otimes I_m) - h(\widetilde{A} \otimes J)\right]\widehat{\delta} = \widehat{Z} - h(\widetilde{A} \otimes J)\widehat{F}. \tag{2.4}$$

Implementation will require the same operation count as SDIRK methods for the factorization and back-substitution. In addition, another ms^2 is required for each transformation of the variables. Like SDIRK methods the maximum order we can obtain is $s+1$, but the stage order can be s. Despite the additional cost of SIRK compared to SDIRK methods, we will choose SIRK methods for our implementation. The reason is that for differential algebraic equations, and also for very stiff ordinary differential equations, high stage order is important. Furthermore it is easy to construct SIRK methods of arbitrary high order with good stability properties, as we shall see in the next sections.

Properties of Laguerre Polynomials

In the following sections we will need some properties of Laguerre polynomials. We can define the n-degree Laguerre polynomial as

$$L_n(x) = \sum_{i=0}^{n} \binom{n}{i} \frac{(-x)^i}{i!}. \tag{2.5}$$

It is easy to verify that the Laguerre polynomial $L_n(x)$ of degree $n \geq 1$, and its derivative are generated by the recursion

$$L'_n(x) = L'_{n-1}(x) - L_{n-1}(x), \tag{2.6}$$
$$L_n(x) = L_{n-1}(x) + \frac{x}{n} L'_n(x), \tag{2.7}$$

where $L_0(x) = 1$ and $L'_0(x) = 0$.

A property we will need later is

$$\int_0^\mu L_n(x)\mathrm{d}x = -\frac{\mu}{n+1} L'{}_{n+1}(\mu), \tag{2.8}$$

which can easily be verified rewriting (2.6) as

$$L_n(x) = L'_n(x) - L'_{n+1}(x),$$

and (2.7) as

$$L_n(x) - L_{n+1}(x) = -\frac{x}{n+1} L'_{n+1}(x).$$

Integrating the first equation and then combining it with the second, we have the result.

2.2.1 Construction of Methods

We will from now on consider only SIRK methods which are collocation methods, so we assume that the collocation points $c_1, \ldots, c_s$ are distinct, and the method satisfies $C(s)$ and $B(s)$.

Following Butcher [5] we can determine the method completely from a choice of s and the eigenvalue λ:

Theorem 2.1 *For a singly implicit Runge-Kutta method, where the A matrix has the eigenvalue λ and $c = [c_1, \ldots, c_s]$, $c_1/\lambda, c_2/\lambda, \ldots, c_s/\lambda$ are the zeros of the s-degree Laguerre polynomial L_s, given by (2.5).*

Proof. Let c^k denote the vector $[c_1^k, \ldots, c_s^k]^T$. Then $C(s)$ can be written

$$Ac^{k-1} = \frac{c^k}{k}, \quad k = 1, \ldots, s.$$

Then

$$Ac^k = A^2 k c^{k-1} = A^3 k(k-1)c^{k-2} = \ldots = A^{k+1}(k)!c^0,$$

and

$$A^k c^0 = \frac{c^k}{k!}.$$

Applying the Cayley-Hamilton theorem (see [5] sect. 104) to A we obtain

$$\frac{1}{\lambda^s}(A - I\lambda)^s c^0 = \sum_{i=0}^{s} \begin{pmatrix} s \\ i \end{pmatrix} \left(\frac{-1}{\lambda}\right)^i A^i c^0 = \sum_{i=0}^{s} \frac{(-1)^i}{i!} \begin{pmatrix} s \\ i \end{pmatrix} \left(\frac{c}{\lambda}\right)^i = 0.$$

For each component we have $L_s(c_i/\lambda) = 0$. □

The transformation matrices T and T^{-1} in (2.2) can be expressed in terms of Laguerre polynomials and its zeros,

$$T = \begin{bmatrix} L_0(\xi_1) & \cdots & L_{s-1}(\xi_1) \\ \vdots & & \vdots \\ L_0(\xi_s) & \cdots & L_{s-1}(\xi_s) \end{bmatrix}, \tag{2.9}$$

and

$$T^{-1} = \begin{bmatrix} \frac{\xi_1 L_0(\xi_1)}{s^2 L_{s-1}(\xi_1)^2} & \cdots & \frac{\xi_s L_0(\xi_s)}{s^2 L_{s-1}(\xi_s)^2} \\ \vdots & & \vdots \\ \frac{\xi_1 L_{s-1}(\xi_1)}{s^2 L_{s-1}(\xi_1)^2} & \cdots & \frac{\xi_s L_{s-1}(\xi_s)}{s^2 L_{s-1}(\xi_s)^2} \end{bmatrix}, \tag{2.10}$$

where $\xi_1, \ldots, \xi_s$ are the zeros of L_s. The proof makes use of some basic properties of Laguerre polynomials and is given in [6].

2.2.2 Choosing the Parameter

Using (1.22) it is easy to see that the stability function of a SIRK method has the form

$$R(z) = \frac{P(z)}{(1 - z\lambda)^s}, \tag{2.11}$$

where $P(z)$ is a polynomial of degree $m \leq s$.

Following [5] p. 246, we look at $R(z)$ as a *restricted Padé approximation* to the exponential function

$$|R(z) - \exp(z)| = \mathcal{O}(z^{m+1}). \tag{2.12}$$

We then insert (2.11) into (2.12). The result is

$$P(z) = (1 - z\lambda)^s \exp(z) + \mathcal{O}(z^{m+1}). \tag{2.13}$$

Then we want to write the term $(1 - z\lambda)^s \exp(z)$ as a Taylor series in z. The following lemma is proved in [5].

Lemma 2.2 *Let $L_s^{(n)}(x)$ denote the n-th derivative of the s-degree Laguerre polynomial. For negative n we define recursively $L_s^{(n)}(x) = \int_0^x L_s^{(n+1)}(u)du$, $L_s^{(0)} = L_s$. Then we have*

$$(1 - z\lambda)^s \exp(z) = (-1)^s \sum_{i=0}^{\infty} \lambda^i L_s^{(s-i)}(1/\lambda) z^i. \tag{2.14}$$

Using the result we find an expression for $R(z)$,

$$R(z) = \frac{(-1)^s \sum_{i=0}^{m} \lambda^i L_s^{(s-i)}(1/\lambda) z^i}{(1 - z\lambda)^s},$$

and the error term

$$\exp(z) - R(z) = (-1)^s \lambda^{m+1} L_s^{(s-1-m)}(1/\lambda) z^{m+1} + \mathcal{O}(z^{m+2}).$$

Optimizing the Order

If we have $m = s$ the error constant can be written

$$\begin{aligned}(-1)^s \lambda^{m+1} L_s^{(-1)}(1/\lambda) &= (-1)^s \lambda^{m+1} \int_0^{1/\lambda} L_s(u) \mathrm{d}u \\ &= (-1)^{s+1} \frac{\lambda^m}{s+1} L'_{s+1}(1/\lambda).\end{aligned}$$

This means that if $1/\lambda$ is chosen as a zero of L'_{s+1}, we obtain order $s+1$. A-stability is possible for $s = 1$, 2, 3 and 5, but stability at infinity is incompatible with order $s+1$.

Stability at Infinity

Instead of optimizing the order, we can use the parameter λ to make the method stable at infinity, in other words choose λ such that $R(z) \to 0$ as $z \to \infty$. If $1/\lambda$ equals a zero of L_s the z^s term of $P(z)$ vanishes, and since the denominator has degree s, we have $R(\infty) = 0$.

We still have s different choices of λ, and we will use this to make the stability region reasonable large. For orders $s = 1, \ldots, 6$ and 8 we can obtain A-stability together with stability at infinity (L-stability). Following [6], we require only $A(\alpha)$ where $\alpha \geq 1.35$. Then we have the following choices for $\lambda = 1/\xi_i$, where ξ_i is the i-th zero of L_s:

stages	choices of Laguerre zero
$s = 1$	$i = 1$
$s = 2, 3$	$i = 1, 2$
$s = 4, 5$	$i = 1, 2, 3$
$s = 6, 7$	$i = 1, 2, 3, 4$
$s = 8, 9, 10$	$i = 1, 2, 3, 4, 5$

Numerical experiments show that the error constant is minimized when i has the greatest possible value.

We see that even when we choose λ as the maximum allowed Laguerre zero from the table, some of the collocation points will lie far outside the interval $[t_n, t_n + h]$. For example for the fifth order method we have $c_1 = 0.073$, $c_2 = 0.393$, $c_3 = 1$, $c_4 = 1.970$ and $c_5 = 3.515$. This appears to be one of the main drawbacks for singly implicit methods.

2.3 Solving Equations in Implicit Form

Details on how to implement SIRK methods and error estimates efficiently for explicit ordinary differential equations, using transformed variables, can be found in [6]. We will in this section discuss implementation details for equations in the form

$$\Phi(y', y, t) = 0. \tag{2.15}$$

Runge-Kutta methods applied to this problem leads to the system of algebraic equations

$$\begin{aligned} Y_i &= y_n + h \sum_{j=1}^{s} a_{ij} F_j \ , \quad i = 1, \ldots, s, \\ 0 &= \Phi(F_k, Y_k) \ , \quad k = 1, \ldots, s, \end{aligned}$$

which we solve for either Y_i or F_i. For simplicity of notation we have assumed autonomous system. The numerical approximation is computed by

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i F_i,$$

as usual.

2.3.1 Solving the Non-Linear System

If we assume that A is invertible we can eliminate F_i. Then we have

$$F = \begin{bmatrix} F_1 \\ \vdots \\ F_s \end{bmatrix} = \frac{1}{h}(A^{-1} \otimes I_m)Z,$$

and the nonlinear system becomes

$$0 = \Phi(\frac{1}{h} \sum_{j=1}^{s} w_{ij} Z_j, y_n + Z_i) \ , \quad i = 1, \ldots, s, \tag{2.16}$$

where w_{ij} is the (i,j)-th element of A^{-1}. Applying modified Newton-Raphson iterations to this system leads to the linear system

$$\left((I \otimes J) + \frac{1}{h}(A^{-1} \otimes M)\right) \delta = \begin{bmatrix} \Phi(F_1, y_n + Z_1) \\ \vdots \\ \Phi(F_s, y_n + Z_s) \end{bmatrix},$$

where $J = \partial\Phi/\partial y$ and $M = \partial\Phi/\partial y'$. If our Runge-Kutta method is a SIRK method, we can transform the linear system to a form similar to (2.4). We substitute F, Z and δ with the corresponding transformed variables $\widehat{F}$, $\widehat{Z}$ and $\widehat{\delta}$ and multiply the system from left with $(T^{-1} \otimes I_m)$. This leads to

$$\left((I \otimes J) + \frac{1}{H}(\widehat{A}^{-1} \otimes M)\right) \widehat{\delta} = (T^{-1} \otimes I_m) \begin{bmatrix} \Phi(F_1, y_n + Z_1) \\ \vdots \\ \Phi(F_s, y_n + Z_s) \end{bmatrix}, \tag{2.17}$$

where the matrix $\widehat{A}^{-1}$ is

$$\widehat{A}^{-1} = \begin{bmatrix} 1 & & & 0 \\ -1 & 1 & & \\ & \ddots & \ddots & \\ 0 & & -1 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 1 & & \vdots \\ \vdots & & \ddots & 0 \\ 1 & 1 & \cdots & 1 \end{bmatrix}$$

and $H = h\lambda$. The system is now separated into s parts, each of the form

$$G\widehat{\delta}_i = \widehat{\Phi}(F_i, y_n + Z_i) - \frac{1}{H} M \sum_{j=1}^{i-1} \widehat{\delta}_j, \tag{2.18}$$

where G is the iteration matrix

$$G = \left[J + \frac{1}{H} M\right].$$

An algorithm for performing a modified Newton-Raphson iteration k, given starting values Z^0, is:

1. Compute derivatives $F^k = \frac{1}{H}(A^{-1} \otimes I_m) Z^k$.
2. Compute the residuals $\Phi_i^k = \Phi(F_i^k, y_n + Z_i^k, t_n + c_i h)$, $i = 1, \ldots, s$.
3. Transform the residuals
$$[\widehat{\Phi}_1^k, \ldots, \widehat{\Phi}_s^k]^T = (T^{-1} \otimes I_m)[\Phi_1^k, \ldots, \Phi_s^k]^T .$$
4. Then, for $i = 1, \ldots, s$
   (a) Calculate the right-hand side of the linear system
   $$\rho_i^k = \widehat{\Phi}_i^k - \frac{1}{H} M \sum_{j=1}^{i-1} \widehat{\delta}_j^k.$$
   (b) Solve the system $G\widehat{\delta}_i^k = \rho_i^k$.
5. Untransform the solution
$$[\delta_1^k, \ldots, \delta_s^k]^T = (T \otimes I_m)[\widehat{\delta}_s^k, \ldots, \widehat{\delta}_s^k]^T.$$
6. Update the stage values $Z_i^{k+1} = Z_i^k - \delta_i^k$, $i = 1, \ldots, s$.

If we use a SIRK method with $1/\lambda = \xi_\nu$, where ξ_ν is the ν-th zero of the s-degree Laguerre polynomial, we know that the method is stiffly accurate. Then, if the iterations converge, a numerical solution is given by

$$y_{n+1} = y_n + Z_\nu^k, \tag{2.19}$$

for some $k \geq 1$.

Stopping Criterion

We use the ideas of [8] chapter IV.8. The rate of convergence Θ for the modified Newton-Raphson iteration can be estimated by

$$\Theta_k = ||\Delta Z^{k+1}||/||\Delta Z^k|| \ , \quad k \geq 0, \tag{2.20}$$

where

$$\Delta Z^k = \begin{bmatrix} Z_1^{k+1} - Z_1^k \\ \vdots \\ Z_s^{k+1} - Z_s^k \end{bmatrix},$$

and $||\cdot||$ is usually the same norm used for the local error estimation. If $\Theta < 1$, the iteration converges. For contractions on Banach spaces we have the result

$$||Z^{k+1} - Z^*|| \leq \frac{\Theta}{1-\Theta}||\Delta Z^k||,$$

where Z^* is the solution of the nonlinear system (2.16). We therefore stop the iteration when

$$\eta_k||\Delta Z^k|| \leq \kappa \ , \quad \eta_k = \frac{\Theta_k}{1-\Theta_k},$$

where $\kappa > 0$ is a suitable constant. If we assume that the requested tolerance is hidden in the norm, experiments say that κ should be in the interval $[10^{-2}, 1]$. The computation above requires at least two iterations. Normally we will stop after only one iteration if $||\Delta Z^1||$ is sufficiently small.

Iteration should be given up if $\Theta_k \geq 1$ for any k. Usually we allow only a certain number of iterations, say $k_{\max}$. The size of $||\Delta Z^{k_{\max}-1}||$ can be estimated by

$$\begin{aligned} ||\Delta Z^{k_{\max}-1}|| &= \Theta_{k_{\max}-1} \cdots \Theta_k \Theta_{k-1} \cdots \Theta_0 ||\Delta Z^0|| \\ &\approx \Theta_k^{k_{\max}-k-1}||\Delta Z^k||, \end{aligned} \tag{2.21}$$

where we have assumed that

$$\Theta_k \approx \Theta_{k+1} \approx \ldots \approx \Theta_{k_{\max}-1}.$$

Then we can estimate the size of $||Z^{k_{\max}} - Z^*||$ after k iterations by the formula

$$\frac{\Theta_k^{k_{\max}-k}}{1-\Theta_k}||\Delta Z^k|| \ . \tag{2.22}$$

If this value is greater than κ we should give up the iteration process and try again with an updated Jacobian or a smaller step-size.

Starting Values

The starting values $Z_i^0 = 0$, $i = 1, \ldots, s$ are the easiest choice, and will usually work because $Z_i = y_0 + \mathcal{O}(h)$. But using information from the preceding step, we can find better estimates. Since the method is a collocation method of order s we know that there is a polynomial q of degree $s-1$,

$$q(c) = y(t_0 + ch) - y_0,$$

which interpolates Z_i at the points c_i. We can rewrite Z_i in terms of transformed stage derivatives $\widehat{F}_1, \ldots, \widehat{F}_s$ as

$$\begin{aligned} Z_i &= h\sum_{j=1}^{s} a_{ij} F_j \\ &= h\sum_{j=1}^{s}\sum_{k=1}^{s} a_{ij} L_{k-1}(c_i/\lambda)\widehat{F}_k. \end{aligned}$$

Using the $C(s)$ condition and property (2.8) we can write the coefficient of $\widehat{F}_k$ as

$$\begin{aligned} h\sum_{j=1}^{s} a_{ij} L_{k-1}(c_i/\lambda) &= h\int_0^{c_j} L_{k-1}(c/\lambda)\mathrm{d}c \\ &= -h\frac{c_i}{k}L_k'(c_i/\lambda). \end{aligned}$$

The resulting expression for Z_i is

$$Z_i = -h\sum_{k=1}^{s}\frac{c_i}{k}L_k'(c_i/\lambda)\widehat{F}_k, \tag{2.23}$$

and then $q(c)$ has to be

$$q(c) = -h\sum_{k=1}^{s}\frac{c}{k}L_k'(c/\lambda)\widehat{F}_k. \tag{2.24}$$

Inserting the values

$$1 + \frac{h_{\mathrm{new}}}{h_{\mathrm{old}}}c_i, \quad i = 1, \ldots, s,$$

provides starting values Z_i^0 for the Newton-Raphson iteration in the next step. The formula can also be used for interpolation of output values.

2.3.2 Avoiding Unnecessary Jacobian Evaluations

Since Jacobian evaluation and LU-decomposition is the most costly task of a step, we may want to use the same matrix G for several successive integration steps. The problem with an automatic step-size control code is that the value of

$H = h\lambda$ changes. Using an idea of [3] we replace the Newton-Raphson iteration (2.17) with the system

$$\frac{1}{c}\begin{bmatrix} \widetilde{G} & 0 & \cdots & 0 \\ \frac{1}{\widetilde{H}}\widetilde{M} & \widetilde{G} & & \vdots \\ \vdots & & \ddots & 0 \\ \frac{1}{\widetilde{H}}\widetilde{M} & & \cdots & \widetilde{G} \end{bmatrix} (\widehat{Z}^k - \widehat{Z}^{k+1}) = (T^{-1} \otimes I_m) \begin{bmatrix} \Phi(F_1, y_n + Z_1) \\ \vdots \\ \Phi(F_s, y_n + Z_s) \end{bmatrix},$$

where $\widetilde{G}$, $\widetilde{M}$ and $\widetilde{H}$ are values of G, M and H from a previous step. Our aim is to find a constant c to speed up the convergence. We separate the system into s parts and write each system as a fixed point iteration,

$$\widehat{Z}_i^{k+1} = \widehat{Z}_i^k - c\widetilde{G}^{-1}\widehat{\Phi}(\frac{1}{h}\sum_{j=1}^{s} w_{ij} Z_j, y_n + Z_i) + \frac{1}{\widetilde{H}}\widetilde{G}^{-1}\sum_{j=1}^{i-1} M\widehat{\delta}_j.$$

Taking the derivative with respect to $\widehat{Z}_i^k$ of the right hand side gives

$$B = I - c\widetilde{G}^{-1}G.$$

The spectral radius of B is a measure of the convergence rate. We insert the linear equation $y' - \mu y = 0$, and get

$$G = \frac{1}{H} - \mu \quad \text{and} \quad \widetilde{G} = \frac{1}{\widetilde{H}} - \mu.$$

Then the spectral radius of B equals

$$\bar{\mu} = 1 - c\frac{\frac{1}{H} - \mu}{\frac{1}{\widetilde{H}} - \mu}.$$

We then choose c to minimize the maximum over all values of μ in the left half plane, and get

$$c = \frac{2}{\widetilde{H}/H + 1}.$$

The corresponding $\bar{\mu}$ is

$$\bar{\mu} = \frac{1 - \widetilde{H}/H}{1 + \widetilde{H}/H}\ . \tag{2.25}$$

We can use the equation (2.25) to decide when to evaluate a new Jacobian. For STRIDE++ a value of 0.25 is used as an upper limit.

2.4 Estimating the Error

We want to be able to estimate the local error for SIRK methods, and use the estimate to find a suitable step-size h for the next step. One possibility is to use an s-stage p-th order method embedded in an $s+1$-stage method of order $p+1$.

The difference of these two methods can be used as an estimate of the local error. We use the p-th order method to advance the solution, and a step-size guess for the next step is given by

$$h_{\text{new}} = \beta h_{\text{old}} \left(\frac{1}{E}\right)^{\frac{1}{p+1}}, \tag{2.26}$$

where E is the norm of the estimated error. We have assumed that the user supplied tolerances are hidden in the norm. The coefficient β is a safety factor $0 < \beta \le 1$.

We will assume that we have chosen the eigenvalue λ such that the SIRK method is stable at infinity and has order s. Then we try to find an embedded $s+1$ order method of the form

$$\begin{array}{c|cccc}
c_1 & a_{11} & \cdots & a_{1s} & 0 \\
\vdots & \vdots & & \vdots & 0 \\
c_s & a_{s1} & \cdots & a_{ss} & 0 \\
c_{s+1} & r_1 & \cdots & r_s & \lambda \\
\hline
 & \widehat{b}_1 & \cdots & \widehat{b}_s & \widehat{b}_{s+1}
\end{array}, \tag{2.27}$$

where a_{ij} are the same elements of the A matrix, as in last section, and λ is the single eigenvalue of A. Following Butcher [6] we will express the error estimate by transformed stage derivatives $\widehat{F}_i$, $i = 1, \ldots, s$.

2.4.1 Computing the Error Estimate

For the error estimate we need the value of Z_{s+1}, which is given by iteration on the linear system

$$\left[J + \frac{1}{H}M\right]\delta_{s+1} = \Phi(F_{s+1}, y_n + Z_{s+1}) - \frac{1}{h}M\sum_{i=1}^{s}\omega_{s+1,i}\delta_i, \tag{2.28}$$

where ω_{ij} is the (i,j) element of the matrix

$$\begin{bmatrix} A & 0 \\ r^T & \lambda \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & 0 \\ -r^T A^{-1}/\lambda & 1/\lambda \end{bmatrix}.$$

We assume that $\delta_1, \ldots, \delta_s$ are small and neglect the last term of (2.28). F_{s+1} is given from the Runge-Kutta matrix as

$$F_{s+1} = \frac{1}{\lambda}\left(\frac{Z_{s+1}}{h} - \sum_{i=1}^{s} r_i F_i\right).$$

Then we try to express the sum $\sum_{i=1}^{s} r_i F_i$ in terms of transformed variables. Using the transformation matrix (2.9) we can write

$$\sum_{i=1}^{s} r_i F_i \quad = \quad \sum_{i=1}^{s}\sum_{j=1}^{s} r_i L_{j-1}(\xi_i)\widehat{F}_j$$

$$= \sum_{j=1}^{s}\left(\sum_{i=1}^{s} r_i L_{j-1}(\xi_i)\right)\widehat{F}_j.$$

The $C(s)$ condition for stage $s+1$ implies

$$\sum_{i=1}^{s} r_i p(c_i) + \lambda p(c_{s+1}) = \int_0^{c_{s+1}} p(c)\mathrm{d}c, \tag{2.29}$$

for any polynomial p of degree less than s. Using this, the coefficient of $\widehat{F}_j$ can be written as

$$\begin{aligned}\sum_{i=1}^{s} r_i L_{j-1}(\xi_i) &= \int_0^{c_{s+1}} L_{j-1}(c/\lambda)\mathrm{d}c - \lambda L_{j-1}(c_{s+1}/\lambda)\\ &= -c_{s+1}\frac{1}{j}L'_j(c_{s+1}/\lambda) - \lambda L_{j-1}(c_{s+1}/\lambda),\end{aligned}$$

where we have used (2.29) with $p = L_{j-1}$ and the property (2.8).

In a similar way we can find the coefficients $\widehat{b}_j$, $j = 1, \ldots, s+1$. The $B(s+1)$ condition is equivalent to

$$\sum_{j=1}^{s+1} \widehat{b}_j p(c_j) = \int_0^1 p(c)\mathrm{d}c, \tag{2.30}$$

where p is any polynomial of degree less than $s+1$. We first find $\widehat{b}_{s+1}$ by setting $p(c) = L_s(c/\lambda)$ into (2.30). Since c_j is the j-th zero of L_s we have

$$\begin{aligned}\widehat{b}_{s+1} L_s(c_{s+1}/\lambda) &= \int_0^1 L_s(c/\lambda)\mathrm{d}c\\ &= -\frac{1}{s+1}L'_{s+1}(1/\lambda).\end{aligned}$$

Then we express the sum $h\sum_{i=1}^{s}\widehat{b}_i F_i$ in terms of transformed values $\widehat{F}_j$, $j = 1, \ldots, s$. The result is

$$h\sum_{i=1}^{s}\sum_{j=1}^{s}\widehat{b}_i L_{j-1}(\xi_i)\widehat{F}_j.$$

Using (2.30) again, we find the coefficient of $\widehat{F}_j$ as

$$\begin{aligned}h\sum_{i=1}^{s}\widehat{b}_i L_{j-1}(\xi_i) &= \int_0^1 L_{j-1}(c/\lambda)\mathrm{d}c - h\widehat{b}_{s+1}L_{j-1}(c_{s+1}/\lambda)\\ &= -h\frac{1}{j}L'_j(1/\lambda) - h\widehat{b}_{s+1}L_{j-1}(c_{s+1}/\lambda).\end{aligned}$$

We also need the sum $h\sum_{i=1}^{s} b_i F_i = h\sum_{i=1}^{s}\sum_{j=1}^{s} b_i L_{j-1}(\xi_i)\widehat{F}_j$. The same calculation as above yields

$$h\sum_{i=1}^{s} b_i L_{j-1}(\xi_i) = -h\frac{1}{j}L'_j(1/\lambda),$$

by the $B(s)$ condition.

All the coefficients in the error estimating method are now determined by the choice of the parameter c_{s+1}. The value $c_{s+1} = 0$ seems to be a good choice, because then c_{s+1} is distinct from the other c_i's and it makes the coefficients simpler to evaluate.

The error estimate then takes the form

$$\begin{aligned} \widehat{y}_n - y_n &= h\sum_{j=1}^{s}(\widehat{b}_j - b_j)F_j + h\widehat{b}_{s+1}F_{s+1} \\ &= h\widehat{b}_{s+1}\left(F_{s+1} - \sum_{j=1}^{s}\widehat{F}_j\right) \qquad (2.31) \\ &= -\frac{L'_{s+1}(1/\lambda)}{\lambda(s+1)}Z_{s+1}, \qquad (2.32) \end{aligned}$$

where Z_{s+1} is computed from the Newton-Raphson iteration

$$\begin{aligned} G\delta_{s+1} &= \Phi\left(\frac{1}{H}(Z_{s+1}^k + H\sum_{j=1}^{s}\widehat{F}_j), y_n + Z_{s+1}^k\right) \\ Z_{s+1}^{k+1} &= Z_{s+1}^k - \delta_{s+1}, \quad k \geq 0. \end{aligned}$$

Starting value for the iteration should be chosen as $Z_{s+1}^0 = 0$. Numerical experiments show that normally one iteration is enough to get a value of Z_{s+1} within the tolerance. A simple formula for the error estimate is then

$$E = \alpha_s G^{-1}\Phi(\sum_{j=1}^{s}\widehat{F}_j, y_n), \qquad (2.33)$$

where

$$\alpha_s = -\frac{L'_{s+1}(1/\lambda)}{\lambda(s+1)}.$$

2.4.2 Stability of the Error Estimate

The error estimating method can be calculated as

$$\widehat{y}_{n+1} = y_{n+1} + \alpha_s Z_{s+1}.$$

We want this method to have good stability properties, and the stability at infinity is of special interest for differential algebraic equations. Inserting the test equation for linear stability $y' = \mu y$, we get for the error estimating stage

$$\Phi(F_{s+1}, y_n + Z_{s+1}) = F_{s+1} - \mu(y_n + Z_{s+1}) = 0.$$

The Runge-Kutta method gives

$$Z_{s+1} = h\sum_{j=1}^{s} r_j F_j + h\lambda F_{s+1} = \sum_{j=1}^{s} e_j Z_j + h\lambda F_{s+1},$$

where $e_1, \ldots, e_s$ are some constants given by the Runge-Kutta matrix. Eliminating F_{s+1} from these two equations we obtain

$$Z_{s+1} = \frac{\sum_{j=1}^{s} e_j Z_j + h\lambda\mu y_n}{1 - h\lambda\mu}. \tag{2.34}$$

The internal stages $Z_1, \ldots, Z_s$, can be expressed by the internal stability functions $R_j(c_j h\mu)$, $j = 1, \ldots, s$, as

$$y_n + Z_j = R_j(c_j h\mu) y_n.$$

Equation (2.34) can now be written as

$$Z_{s+1} = \frac{(\sum_{j=1}^{s} e_j (R_j(c_j h\mu) - 1) + h\mu\lambda) y_n}{1 - h\mu\lambda}.$$

When $h\mu \to \infty$, R_j will approach 0 for all $j = 1, \ldots, s$. Therefore $Z_{s+1} \to -1$ as $h\mu \to \infty$.

Further Improvement of the Estimate

First we consider ordinary differential equations on explicit form $y' = f(y, t)$. Following [8] p.134 (an idea of Shampine) we introduce the new error estimate

$$\widetilde{E} = (I - h\lambda J)^{-1} E, \tag{2.35}$$

where E is the error estimate given by (2.33). This does not change the order of the estimate when h is small, but if we insert the test equation $y' = \mu y$ we get

$$\widetilde{E} = \frac{E}{1 - h\lambda\mu},$$

which approaches zero for large values of $h\mu$.

We can generalize the idea to equations on the form

$$0 = My' - f(y, t) \tag{2.36}$$

by formally replacing f by $M^{-1} f$ and J by $(-M^{-1} J)$. The resulting expression for the error estimate is

$$D = (I + HM^{-1} J)^{-1} E = (M + HJ)^{-1} ME. \tag{2.37}$$

The estimate is generalized to systems on the form $\Phi(y', y, t) = 0$ using $M = \partial\Phi / \partial y'$. Since we have already computed the LU-decomposition of the matrix $[(1/H)M + J]$, the computation of the error estimate requires only an extra back-substitution.

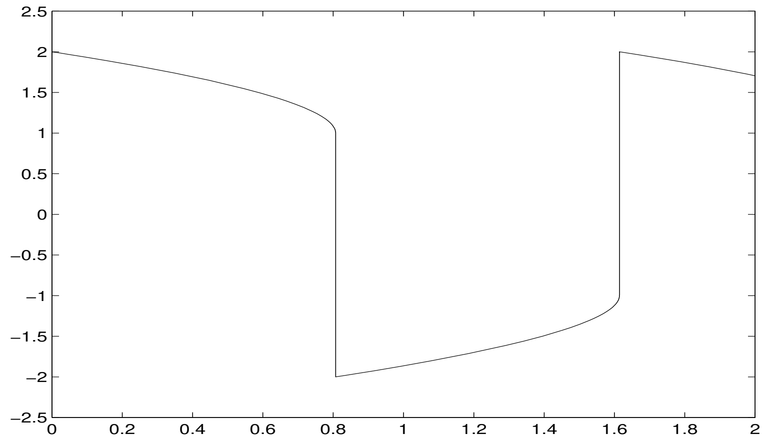

Figure 2.1: The solution component y_1 of the van der Pol equation (2.38).

2.4.3 Numerical Study of the Error Estimates

We have used an experimental version of STRIDE to plot the different error estimates as functions of the step-size h. The test equation is the van der Pol equation,

$$\begin{aligned} y_1' &= y_2, \\ \epsilon y_2' &= (1 - y_1^2)y_2 - y_1, \end{aligned} \tag{2.38}$$

with initial conditions $y_1(0) = 2$, $y_2(0) = -0.6$ and parameter $\epsilon = 10^{-6}$. The method used is a 4-th order SIRK implementation with variable step-size, and relative and absolute tolerances equal to 10^{-4}. Figure (2.1) shows the solution of the component y_1.

In figure (2.2) we have plotted the norm of the following error estimates:

1. The error estimate (2.33) is indicated by [+]. The computation requires one function evaluation and one back-substitution.

2. The error estimate (2.37) is indicated by [o]. This is computationally the most expensive estimate. It requires one function evaluation and two back-substitutions.

3. A cheap way to estimate the error is to approximate F_{s+1} in (2.31) by the derivative at the end of the last step. This can easily be calculated from the last stage values using the Runge-Kutta matrix,

$$F_{s+1} \approx \sum_{j=1}^{s} w_{\nu j} Z_j,$$

where $w_{\nu j}$ is the (ν, j) element of A^{-1}, and ν is the same as in (2.19).

Using (2.31) this alternative error estimate becomes

$$\widetilde{E} = \alpha_s H(F_{s+1} - \sum_{j=1}^{s} \widehat{F}_j). \tag{2.39}$$

No function evaluations or back-substitutions are necessary. This estimate is marked with $[\cdot]$ in the figure.

4. We can use the idea of Shampine on (2.39) to get an error estimate with better stability properties. This yields

$$[M + HJ]^{-1} M(\alpha_s H(F_{s+1} - \sum_{j=1}^{s} \widehat{F}_j)), \tag{2.40}$$

and requires one back-substitution only. The estimate is indicated by $[\times]$.

We have also included an estimate of the real error, computed by STRIDE++ using relative and absolute tolerances equal to 10^{-16}. This "exact" error is represented by the solid line. The horizontal dashed line indicates the tolerance, and the vertical dashed line represent the step-size chosen by the automatic step-size selection mechanism in STRIDE++. We have used the scaled norm

$$\|E\| = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{E_i}{w_i}\right)^2\right]^{\frac{1}{2}}, \tag{2.41}$$

where m is the number of variables and the weights w_i is given by the relative tolerances r_i, the absolute tolerance a_i and the i-th solution component y_i as

$$w_i = r_i|y_i| + a_i.$$

This means that if the norm of the error is greater than 1, the tolerance is exceeded.

The error shows the characteristic "bump" for singularly perturbed problems (see [8]). We see that the error estimate (2.37) is significantly sharper than the other estimates, but it seems to slightly underestimate the error.

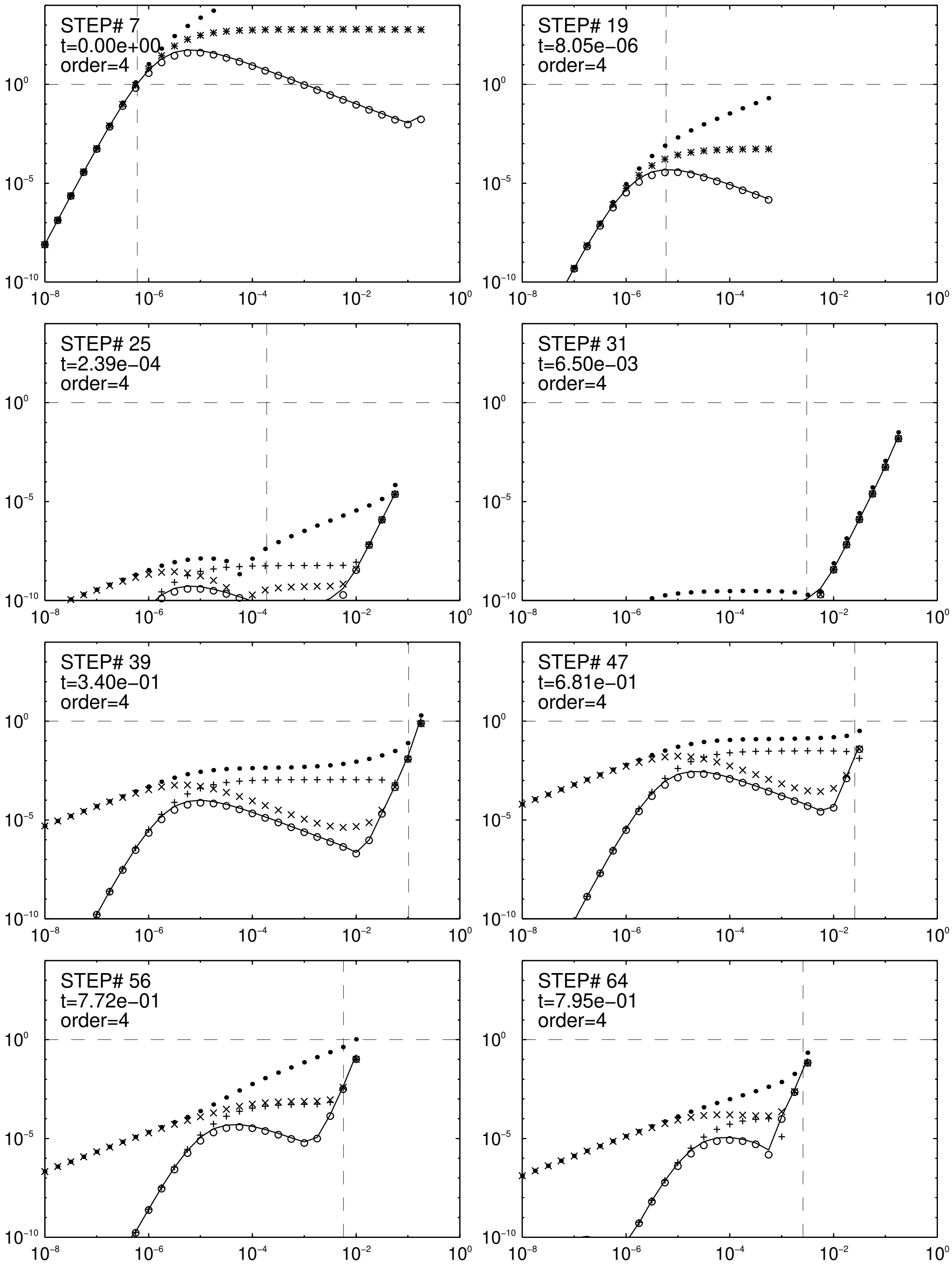

Figure 2.2: Local error estimates for the van der Pol equation.

Chapter 3

Extension to DAEs

In this chapter we will give some convergence results for Runge-Kutta methods applied to DAEs. We will also explain some difficulties that occur when we are using embedded methods for error estimation, and propose some ideas to overcome the difficulties.

We will use the results of Hairer, given in [9] and [8].

3.1 Differential Algebraic Equations of Index 1

We consider the differential algebraic system

$$\begin{aligned} y' &= f(y,z), \\ 0 &= g(y,z), \end{aligned} \tag{3.1}$$

and assume that

$$g_z \text{ has a bounded inverse.} \tag{3.2}$$

Then the system (3.1) has index 1, and by the implicit function theorem we can solve the constraint equation for $z = G(y)$.

We then apply an implicit Runge-Kutta method to the system (3.1). Using the approach from last chapter, we get from equation (2.16) the system

$$\frac{1}{h}\sum_{j=1}^{s} w_{ij}(Y_{nj} - y_n) = f(Y_{ni}, Z_{ni}), \tag{3.3}$$

$$0 = g(Y_{ni}, Z_{ni}). \tag{3.4}$$

where w_{ij} is the (i,j)-th element of A^{-1}. When (Y_{ni}, Z_{ni}) is close to the solution curve, we should be able to solve the system, and the new approximation is computed from

$$y_{n+1} = y_n + \sum_{i=1}^{s} b_i Y'_{ni}, \tag{3.5}$$

$$z_{n+1} = z_n + \sum_{i=1}^{s} b_i Z'_{ni}, \tag{3.6}$$

where Y'_{ni} and Z'_{ni} are given by the Runge-Kutta matrix as

$$Y'_{ni} = \frac{1}{h}\sum_{j=1}^{s} w_{ij}(Y_{nj} - y_n), \tag{3.7}$$

$$Z'_{ni} = \frac{1}{h}\sum_{j=1}^{s} w_{ij}(Z_{nj} - z_n). \tag{3.8}$$

This approach is called the *direct approach.*

There is an alternative approach to solve the system, where we replace the equation (3.6) by

$$z_{n+1} = G(y_{n+1}). \tag{3.9}$$

This is equivalent to solving the ODE

$$y' = f(y, G(y)), \tag{3.10}$$

and we will refer to this method as the *indirect approach.*

The indirect approach has the advantage that the solution will always lie on the manifold $g(y,z) = 0$. This is not necessarily the case for the direct approach, but some special methods called *stiffly accurate* satisfy

$$a_{si} = b_i \quad \text{for i} = 1, \ldots, \text{s},$$

which means that $y_{n+1} = Y_{ns}$ and $z_{n+1} = Z_{ns}$. Equation (3.4) then implies that (3.9) is satisfied, and the indirect and direct approaches are equivalent.

3.1.1 Convergence

For the indirect approach the numerical solution of (3.1) is equivalent to the solution of the ODE (3.10). Therefore the order of both y and z components are equal to the classical order p.

The direct approach has order p for the y component, because the calculation of y_{n+1} is equivalent for the direct and indirect approaches. From [8] chap. VI, we have the following result for the z component:

Theorem 3.1 *Assume that the system (3.1) satisfies (3.2) in a neighbourhood of the solution $(y(t), z(t))$ and that the initial values are consistent. Consider a Runge-Kutta method of classical order p and stage order q, with an invertible coefficient matrix A and stability function $R(h\lambda)$. Then the numerical solution of $z(t)$ has global error*

$$z_n - z(t_n) = \mathcal{O}(h^r),$$

where

a) $r = p$ *for stiffly accurate methods,*

b) $r = \min(p, q+1)$ *if* $-1 \leq R(\infty) < 1$,

c) $r = \min(p-1, q)$ *if* $R(\infty) = +1$.

d) If $|R(\infty)| > 1$, *the numerical solution diverges.*

Proof. Part a) is discussed above. For the remaining cases we note that the $B(p)$ and $C(q)$ conditions imply that

$$z(t_{n+1}) = z(t_n) + h \sum_{i=1}^{s} b_i z'(t_n + c_i h) + \mathcal{O}(h^{p+1}), \tag{3.11}$$

$$z(t_n + c_i h) = z(t_n) + h \sum_{j=1}^{s} a_{ij} z'(t_n + c_j h) + \mathcal{O}(h^{q+1}). \tag{3.12}$$

Since A is invertible, we can compute $z'(t_n + c_i)$ from (3.12) and insert it into (3.11). This yields

$$z(t_{n+1}) = \rho z(t_n) + b^T A^{-1} \widehat{Z}_n + \mathcal{O}(h^{p+1}) + \mathcal{O}(h^{q+1}), \tag{3.13}$$

where $\widehat{Z}_n = [z(t_n + c_1 h), \ldots, z(t_n + c_s h)]^T$ and $\rho = 1 - b^T A^{-1} e = R(\infty)$. From (3.8) and (3.6) we can express the numerical solution as

$$z_{n+1} = \rho z_n + b^T A^{-1} Z_n \tag{3.14}$$

where $Z_n = [Z_{n1}, \ldots, Z_{ns}]^T$. Then we compute the difference $\Delta z_{n+1} = z_{n+1} - z(t_{n+1})$ between the numerical and the exact solution. Defining $\Delta Z_n = Z_n - \widehat{Z}_n$ we can write

$$\Delta z_{n+1} = \rho \Delta z_n + b^T A^{-1} \Delta Z_n + \mathcal{O}(h^{p+1}) + \mathcal{O}(h^{q+1}). \tag{3.15}$$

We then have to use the y component to estimate ΔZ_n. Since $Y_{ni} = y(t_n + c_i h) + \mathcal{O}(h^p) + \mathcal{O}(h^{q+1})$, extraction of Z_{ni} from (3.4) gives

$$Z_{ni} = G(Y_{ni}) = G(y(t_n + c_i h)) + \mathcal{O}(h^p) + \mathcal{O}(h^{q+1}). \tag{3.16}$$

Introducing $r = \min(p, q+1)$ we have

$$\Delta z_{n+1} = \rho \Delta z_n + \delta_{n+1}\ , \quad \delta_{n+1} = \mathcal{O}(h^r). \tag{3.17}$$

Repeated insertion of this formula gives

$$\Delta z_{n+1} = \sum_{i=1}^{n} \rho^{n-i} \delta_i, \tag{3.18}$$

because $\Delta z_0 = 0$. The theorem then follows for $\rho \neq -1$. For $\rho = -1$, Δz_n is a sum of differences $\delta_{j+1} - \delta_j$. But δ_{n+1} is on the form $\delta_{n+1} = d(t_n) h^r + \mathcal{O}(h^{r+1})$ and therefore $\delta_{j+1} - \delta_j = \mathcal{O}(h^{r+1})$. □

From the theorem above we see that a SIRK method of s stages and order $p = s$ applied to the index one system (3.1) has local error $\mathcal{O}(h^{s+1})$ and global error $\mathcal{O}(h^s)$ for all components. The error estimating method (2.27) described in the last chapter has local error $\mathcal{O}(h^{s+2})$.

3.1.2 Local Error Estimation

Local error estimations can be calculated as the difference between two Runge-Kutta solutions of different order. We now consider the differential algebraic system (3.1) of index 1, and non-consistent initial values (y_0, z_0). Then the behavior of the error estimate is quite different from what we observe for ordinary differential equations. The following theorem explains the behaviour, and shows why it is important for both Runge-Kutta methods to have the property $R(\infty) = 0$.

Theorem 8.1 of [9] says:

Theorem 3.2 *Consider the differential algebraic system (3.1) and (3.2) with initial values satisfying*

$$||(g_z^{-1}g)(y_0, z_0)|| \leq \delta \ . \tag{3.19}$$

Let (y_1, z_1) and $(\widehat{y}_1, \widehat{z}_1)$ be the solutions of two different Runge-Kutta methods. Then the difference satisfies

$$\begin{aligned} y_1 - \widehat{y}_1 &= \mathcal{O}(h^{p+1}), & (3.20)\\ z_1 - \widehat{z}_1 &= (\rho - \widehat{\rho})[(g_z^{-1}g)(y_0, z_0) + \mathcal{O}(\delta^2)] + \mathcal{O}(h^{\eta+1}), & (3.21) \end{aligned}$$

where $\rho = R(\infty)$ and the terms $\mathcal{O}(h^{p+1})$ and $\mathcal{O}(h^{\eta+1})$ are the differences of the local errors corresponding to consistent initial values.

Proof. If $g(y_0, z_0)$ is small enough, there is a $\tilde{z}_0$ which is close to z_0 and satisfies $g(y_0, \tilde{z}_0) = 0$ (by Implicit Function Theorem). Let $(y(t), z(t))$ be the solution of (3.1) which passes through $(y_0, \tilde{z}_0)$ and $(y_1, \tilde{z}_1)$ the first Runge-Kutta solution when it is applied to $(y_0, \tilde{z}_0)$.

From the proof in the last section and equation (3.15), we can write the difference

$$z_1 - z(x_0 + h) = \rho(z_0 - \tilde{z}_0) + \mathcal{O}(h^{k+1}), \tag{3.22}$$

where $k = \min(q, p)$. The corresponding expression for $\widehat{z}_1$ is

$$\widehat{z}_1 - z(x_0 + h) = \widehat{\rho}(z_0 - \tilde{z}_0) + \mathcal{O}(h^{l+1}). \tag{3.23}$$

Subtracting (3.23) from (3.22) gives

$$z_1 - \widehat{z}_1 = (\rho - \widehat{\rho})(z_0 - \tilde{z}_0) + \mathcal{O}(h^{\eta+1}) \ , \quad \eta = \min(k, l). \tag{3.24}$$

Then we compute the Taylor expansion

$$-g(y_0, z_0) = g_z(y_0, z_0)(\tilde{z}_0 - z_0) + \mathcal{O}(||\tilde{z}_0 - z_0||^2),$$

and get

$$z_0 - \tilde{z}_0 = (g_z^{-1}g)(y_0, z_0) + \mathcal{O}(\delta^2).$$

If we insert this into (3.24) we get the result of the theorem. □

3.2 Higher Index Differential Algebraic Equations

We will in this section state some convergence results for Runge-Kutta methods applied to index 2 and 3 problems.

3.2.1 Differential Algebraic Equations of Index 2

Consider the differential algebraic system

$$\begin{aligned} y' &= f(y,z), \\ 0 &= g(y), \end{aligned} \tag{3.25}$$

where

$$g_y(y)f_z(y,z) \quad \text{is invertible} \tag{3.26}$$

in a neighbourhood of the exact solution. Then (3.25) is an index 2 problem. We say that the initial values y_0, z_0 are *consistent* if

$$g(y_0) = 0 \ , \quad g_y(y_0)f(y_0,z_0) = 0 \tag{3.27}$$

The direct approach leads to the algebraic system

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i Y_i', \tag{3.28}$$

$$z_{n+1} = z_n + h\sum_{i=1}^{s} b_i Z_i', \tag{3.29}$$

where

$$Y_i' = f(Y_i, Z_i), \qquad 0 = g(Y_i), \tag{3.30}$$

and the internal stages are given by

$$Y_i = y_n + h\sum_{j=1}^{s} a_{ij} Y_j', \tag{3.31}$$

$$Z_i = y_n + h\sum_{j=1}^{s} a_{ij} Z_j'. \tag{3.32}$$

In [9] it is proved that the nonlinear system actually has a locally unique solution when the Runge-Kutta matrix A is invertible.

Local Error and Convergence

Definition 4.3 in [8] chapter VII, defines two projections which are useful for the study of the system (3.25).

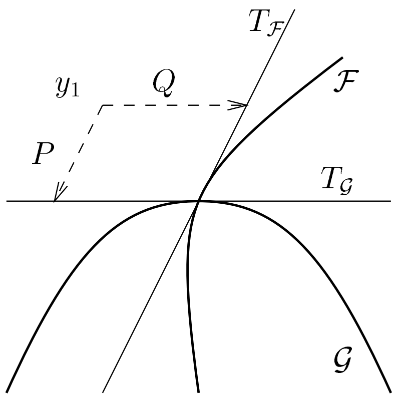

Figure 3.1: Projections P and Q.

Definition 3.3 *For values of* (y_0, z_0) *where* $(g_y f_z)^{-1}(y_0, z_0)$ *has an inverse, we define the projections*

$$Q = (f_z(g_y f_z)^{-1} g_y)(y_0, z_0)\ , \quad P = I - Q\ . \tag{3.33}$$

These matrices have important geometric interpretations for (3.25). Let us define the manifolds $\mathcal{G}$ and $\mathcal{F}$ and their tangent spaces $T_{\mathcal{G}}$ and $T_{\mathcal{F}}$:

$$\begin{aligned} \mathcal{G} &= \{y; g(y) = 0\}, & T_{\mathcal{G}} &= \ker(g_y(y_0)), \\ \mathcal{F} &= \{f(y_0, z); z \text{ arbitrary}\}, & T_{\mathcal{F}} &= \mathrm{Im}(f_z(y_0, z_0)), \end{aligned}$$

where $y_0 \in \mathcal{G}$ and z_0 is chosen so that $f(y_0, z_0)$ lies in $T_{\mathcal{F}}$. $\ker(A)$ denotes the null-space of a matrix A, and $\mathrm{Im}(A)$ the column-space (image) of A.

By (3.26) these two spaces are transversal and together they form the y-space. Then we consider an arbitrary point y_1 in the y-space, and try to find a matrix which projects y_1 to $y_f \in \mathcal{T}_{\mathcal{F}}$ *parallel* to the space $T_{\mathcal{G}}$. We know that y_f must be a linear combination of the columns in f_z, in other words $y_f = f_z \eta$. The residual vector $y_1 - f_z \eta$ has to lie in the space $T_{\mathcal{G}}$, which is the null-space of g_y, and we have $g_y(y_1 - f_z \eta) = 0$. Solving for η gives $\eta = (g_y f_z)^{-1} f_z y_1$. Thus

$$y_f = f_z (g_y f_z)^{-1} g_y y_1 = Q y_1\ .$$

The conclusion is that P projects onto $T_{\mathcal{G}}$ parallel to $T_{\mathcal{F}}$ and Q projects onto $T_{\mathcal{F}}$ parallel to $T_{\mathcal{G}}$ (see fig (3.1)).

Assume *consistent* initial values y_0, z_0, and let y_1, z_1 be the Runge-Kutta solution after one step. We denote the local error by

$$\delta y = y_1 - y(t+h) \quad \text{and} \quad \delta z = z_1 - z(t+h).$$

Proof of the following theorem can be found in [8], section VII.4.

Theorem 3.4 *Assume that the coefficient matrix $A = (a_{ij})$ of the Runge-Kutta matrix is invertible, and that the method satisfies $B(p)$ and $C(q)$, $p \geq q$. Then we have the result*

$$\begin{array}{rclrcl} \delta y &=& \mathcal{O}(h^{q+1}), & P(t)\delta y &=& \mathcal{O}(h^{\eta}), \\ \delta z &=& \mathcal{O}(h^{q}), & & & \end{array}$$

where $\eta = \min(p+1, q+2)$, and $P(t)$ is the projection given by definition (3.3) evaluated at $(y(t), z(t))$. If the method is stiffly accurate we can find a better estimate for the y-component,

$$\delta y = \mathcal{O}(h^{\eta}).$$

We might expect the global error to be one order lower than the local error, but using the projections P and Q, Hairer has found the following results:

Theorem 3.5 (Theorem 4.5 and 4.6 in [8], chap. VII) *Assume that (3.26) holds in a neighbourhood of the solution $(y(t), z(t))$ of (3.25), and that the initial values are consistent. If the Runge-Kutta matrix A is invertible, $|R(\infty)| < 1$, and the local error satisfies*

$$\delta y(t) = \mathcal{O}(h^{r}), \qquad P(t)\delta y(t) = \mathcal{O}(h^{r+1}),$$

$$\delta z(t) = \mathcal{O}(h^{k}),$$

then the global error for the y-components is $\mathcal{O}(h^{r})$, and the global error for the z-components is $\mathcal{O}(h^{k})$.

For most Runge-Kutta methods, sharper results can be found by comparing the Taylor expansions of the numerical and the exact solution. This leads to special order conditions for equations of the form (3.25). (See section 5 in [9].) But for the SIRK methods from last chapter, the results of theorem (3.5) are sufficiently sharp. An s-stage SIRK method satisfies theorem (3.4) with $q = s$ and $\eta = s + 1$, and hence theorem (3.5) with $r = k = s$. The order of convergence is s for all components.

3.2.2 Differential Algebraic Equations of Index 3

Consider the system

$$\begin{array}{rcl} y' &=& f(y, z), \\ z' &=& k(y, z, w), \\ 0 &=& g(y), \end{array} \tag{3.34}$$

which is an index 3 system when

$$g_y(y) f_z(y, z) k_w(y, z, w) \quad \text{has an inverse.} \tag{3.35}$$

Consistent initial values of this system are (y_0, z_0, w_0) which satisfy

$$g = 0, \quad g_y f = 0, \quad g_{yy}(f, f) + g_y f_y f + g_y f_z k = 0.$$

For the index 3 case we will only present a convergence result from [9], again without proof.

Theorem 3.6 (Theorem 6.4 in [9]) *Suppose that (3.35) holds in a neighbourhood of the solution $(y(t), z(t), w(t))$ of (3.34), and that the initial conditions are consistent. Assume that the Runge-Kutta matrix A is invertible, $|R(\infty)| < 1$ and the conditions $B(p)$, $C(q)$ with $p \geq q + 1$ and $q \geq 2$. Then we have the following estimates for the global error:*

$$y_n - y(t_n) = \mathcal{O}(h^q), \quad z_n - z(t_n) = \mathcal{O}(h^q), \quad w_n - w(t_n) = \mathcal{O}(h^{q-1}),$$

for $t_n = nh \leq \mathrm{Const}.$

The assumption $p \geq q + 1$, means that our SIRK methods satisfy the theorem only for $q = s - 1$, which gives order $s - 1$ for the y and z components, and $s - 2$ for the w component. But numerical experiments in the next chapter, indicates that the order might actually be one order higher for all components.

3.2.3 Local Error Estimation

The Index 2 Case

As usual we consider the difference of two different Runge-Kutta methods giving numerical solutions (y_1, z_1) and $(\widehat{y}_1, \widehat{z}_1)$ from inconsistent initial conditions (y_0, z_0).

An expression for the difference can be found by comparing both RK-solutions against the exact solution which passes through consistent initial values $(\tilde{y}_0, \tilde{z}_0)$ sufficiently close to (y_0, z_0). We define $\tilde{y}_0$ by

$$g(\tilde{y}_0) = 0, \qquad P(\tilde{y}_0 - y_0) = 0.$$

Then we can find $\tilde{z}_0$ from

$$(g_y f)(\tilde{y}_0, \tilde{z}_0).$$

If we assume that

$$||((g_y f_z)^{-1} g)(y_0, z_0)|| \leq h\delta, \quad ||((g_y f_z)^{-1} g_y f)(y_0, z_0)|| \leq \theta,$$

where h, δ and θ are sufficiently small, theorem 8.2 in [9] gives the estimates

$$\begin{aligned} y_1 - \widehat{y}_1 &= (\rho - \widehat{\rho})(f_z (g_y f_z)^{-1} g)(y_0, z_0) + \mathcal{O}(h^2\delta + h\delta^2) + \mathcal{O}(h^p), && (3.36) \\ z_1 - \widehat{z}_1 &= -(\sigma - \widehat{\sigma})\frac{1}{h}((g_y f_z)^{-1} g)(y_0, z_0) + (\rho - \widehat{\rho})((g_y f_z)^{-1} g_y f)(y_0, z_0) \\ &\quad + \mathcal{O}(h\delta + \delta^2 + \theta^2) + \mathcal{O}(h^q), && (3.37) \end{aligned}$$

where $\rho = R(\infty) = 1 - b^T A^{-1}e$, $\sigma = b^T A^{-2}e$, and the terms $\mathcal{O}(h^p)$ and $\mathcal{O}(h^q)$ are the corresponding differences for consistent initial values.

As for index 1 problems it is important that $\rho = \widehat{\rho} = 0$, but for the z components there is another term that causes difficulties. When $\sigma \neq \widehat{\sigma}$ the difference grows like $\mathcal{O}(1/h)$ as the step-size decreases. This is a dangerous property for automatic step-size implementations. Following [9] we propose to use the error estimate

$$||y_1 - \widehat{y}_1|| + h||z_1 - \widehat{z}_1||, \tag{3.38}$$

but even then we risk that the term $-(\sigma - \widehat{\sigma})((g_y f_z)^{-1}g)(y_0, z_0)$ is greater than the tolerance. The simplest way to avoid difficulties is to discard the z component completely when we compute the error estimate.

The Index 3 Case

Similar results for the index 3 system (3.34) are given in theorem 8.3 [9]. Then we have to assume that the inconsistent initial values (y_0, z_0, w_0) satisfy

$$\begin{aligned} ||(g_y f_z k_w)^{-1} g|| &\leq h^2\delta, \\ ||(g_y f_z k_w)^{-1} g_y f|| &\leq h\theta, \\ ||(g_y f_z k_w)^{-1}(g_{yy}(f,f) + g_y f_y f + g_y f_z k)|| &\leq \mu, \end{aligned}$$

where h, δ, θ and μ are sufficiently small and all functions are evaluated at (y_0, z_0, w_0). Then the difference between the two Runge-Kutta solutions satisfies

$$\begin{aligned} y_1 - \widehat{y}_1 &= (\rho - \widehat{\rho}) f_z k_w (g_y f_z k_w)^{-1} g \\ &\quad + \mathcal{O}(h^3\delta + h^3\theta + h^2\delta^2 + h^2\theta^2 + h^2\delta\theta) + \mathcal{O}(h^p), \\ z_1 - \widehat{z}_1 &= -(\sigma - \widehat{\sigma})\frac{1}{h} k_w (g_y f_z k_w)^{-1} g \\ &\quad + (\rho - \widehat{\rho}) k_w (g_y f_z k_w)^{-1} g_y f \\ &\quad + \mathcal{O}(h^2\delta + h^2\theta + h\delta^2 + h\theta^2 + h\delta\theta) + \mathcal{O}(h^q), \\ w_1 - \widehat{w}_1 &= -(\tau - \widehat{\tau})\frac{1}{h^2}(g_y f_z k_w)^{-1} g \\ &\quad -(\sigma - \widehat{\sigma})\frac{1}{h}(g_y f_z k_w)^{-1} g_y f \\ &\quad + (\rho - \widehat{\rho})(g_y f_z k_w)^{-1}(g_{yy}(f,f) + g_y f_y f + g_y f_z k) \\ &\quad + \mathcal{O}(h\delta + h\theta + \delta^2 + \theta^2 + \delta\theta) + \mathcal{O}(h^{q-1}), \end{aligned}$$

where $\rho = R(\infty) = 1 - b^T A^{-1}e$, $\sigma = b^T A^{-2}e$, $\tau = b^T A^{-3}e$, and the terms $\mathcal{O}(h^p)$, $\mathcal{O}(h^q)$ and $\mathcal{O}(h^{q-1})$ are the differences corresponding to consistent initial values.

For implementation we have to rescale both the z and w components and the error estimate becomes

$$||y_1 - \widehat{y}_1|| + h||z_1 - \widehat{z}_1|| + h^2||w_1 - \widehat{w}_1||,$$

or we can discard both the z and w components.

Chapter 4

Numerical Results

4.1 Verification of Convergence Results

In this section we will verify the convergence results for SIRK methods applied to index 2 and 3 problems. The test equation we will use is the mathematical pendulum.

4.1.1 The Test Equation

Differential algebraic equations arise frequently in modelling systems of rigid bodies. A simple example of such a system is the mathematical pendulum. In Cartesian coordinates (x, y) the equations modelling the motion, can be written as

$$\begin{aligned} x' &= u, \\ y' &= v, \\ mu' &= -\mu x, \\ mv' &= -\mu y - mg, \\ 0 &= x^2 + y^2 - l^2, \end{aligned} \tag{4.1}$$

where (u, v) is the velocity vector, m is the mass of an infinitely small ball at the end of a massless bar of length l, μ is the tension in the bar and g the gravitational constant. The system (4.1) is an index 3 differential algebraic equation.

In this simple example it is easy to eliminate the constraints from the system and write it as an ordinary differential equation,

$$\theta'' + \frac{g}{l}\sin(\theta) = 0, \tag{4.2}$$

where $x = l\cos(\theta)$ and $y = -l\sin(\theta)$. But for more complicated systems it might be very difficult to find an ODE representation. Therefore it makes sense to solve systems of the form (4.1) directly.

The easiest way to formulate (4.1) as an index 2 problem is to replace the constraint equation by its derivative,

$$0 = xu + yv. \tag{4.3}$$

The geometrical interpretation of (4.3) is that the velocity vector has to be tangential to the manifold given by the constraint $0 = x^2 + y^2 - l^2$, but it does not force the numerical solution to lie in this manifold. We therefore use the index 2 representation from [9],

$$\begin{aligned} x' &= u - x\eta \\ y' &= v - y\eta \\ mu' &= -x\mu \\ mv' &= -y\mu - g \\ 0 &= x^2 + y^2 - l^2 \\ 0 &= xu + yv, \end{aligned} \tag{4.4}$$

where one addition variable η has been introduced.

4.1.2 Results

We have applied SIRK methods of different order to the index 2 and 3 formulations of the pendulum problem. For simplicity, and without loss of generality, we have used $m = 1$, $g = 1$, $l = 1$ and consistent initial values

$$x(0) = 1, \qquad y(0) = u(0) = v(0) = \mu(0) = \eta(0) = 0.$$

The solution of the x, y, u and v components are plotted in figure (4.1). The result is computed from the index 2 formulation (4.4), by STRIDE++ using relative and absolute tolerances equal to 10^{-4}.

A reliable reference solution can be computed from the ordinary differential equation (4.2). This equation can be solved using Elliptic integrals, but we find it more convenient to solve it numerically with a very low tolerance. We then write the equation as

$$\begin{aligned} \theta_1' &= \theta_2, \\ \theta_2' &= -\sin(\theta_1). \end{aligned} \tag{4.5}$$

The initial conditions are

$$\theta_1(0) = \frac{\pi}{2}, \qquad \theta_2(0) = 0.$$

The unknown variables of (4.4) can be calculated by

$$\begin{aligned} x &= \sin(\theta_1), \\ y &= -\cos(\theta_1), \\ u &= \cos(\theta_1)\theta_2, \\ v &= \sin(\theta_1)\theta_2, \\ \mu &= \theta_2^2 + \cos(\theta_1), \\ \eta &= 0. \end{aligned}$$

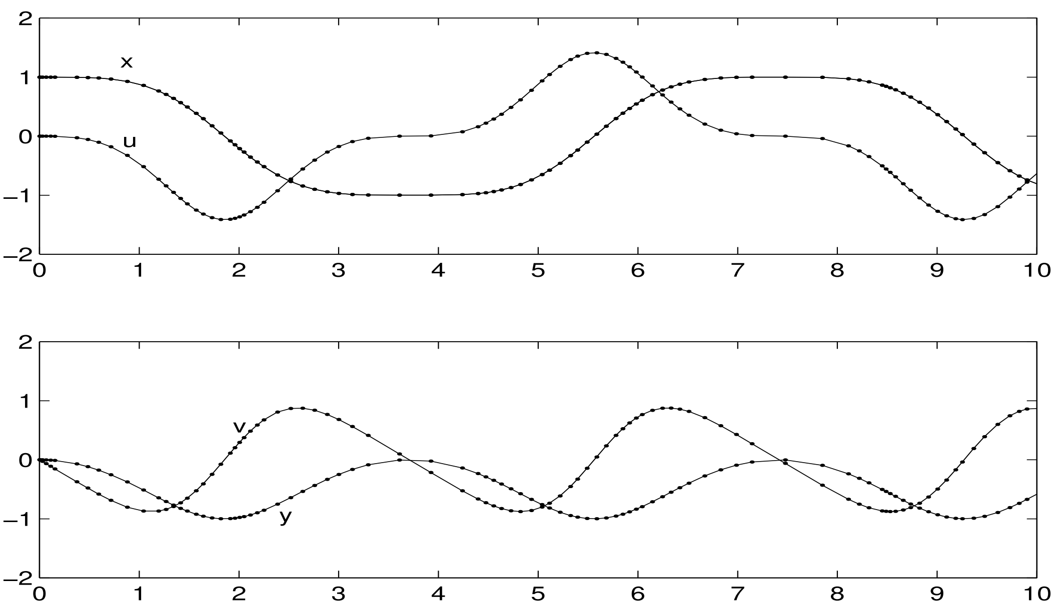

Figure 4.1: Solution of the pendulum problem (4.4).

Figure (4.2) shows estimates of the global error as functions of the step-size, for 2nd and 4th order SIRK methods applied to the index 2 formulation (4.4). The values are obtained by repeated integration from $t = 0$ to $t = 10$ for different fixed step-sizes. In accordance with the theorems (3.4) and (3.5) we observe order equal to the classical order s for all components.

We have also applied different SIRK methods to the index 3 formulation (4.1), and the estimated global error for the 2nd and 4th order methods are plotted in figure (4.3). Since SIRK methods do not satisfy the assumption $p \geq q + 1$ in theorem (3.6), we can expect global error of magnitude $\mathcal{O}(h^{s-1})$ for the y and z components and $\mathcal{O}(h^{s-2})$ for the w-components. From the results in figure (4.3) it seems like the estimates of the theorem are too pessimistic. Apparently, we obtain global errors $\mathcal{O}(h^s)$ for the y and z components and one order lower for the w-components.

4.2 Local Error Estimation for DAEs

Verification of the error estimates from chapter 2 is more difficult for DAEs than for ODEs. The reason is that in every step (possibly except for the first) we start from inconsistent initial values, determined by the given tolerance. The difficulty is to find reasonable consistent initial values for the computation of a reference solution.

We therefore again use the pendulum as our test equation, because there is an intuitive and simple way to find consistent initial values. In the index 3

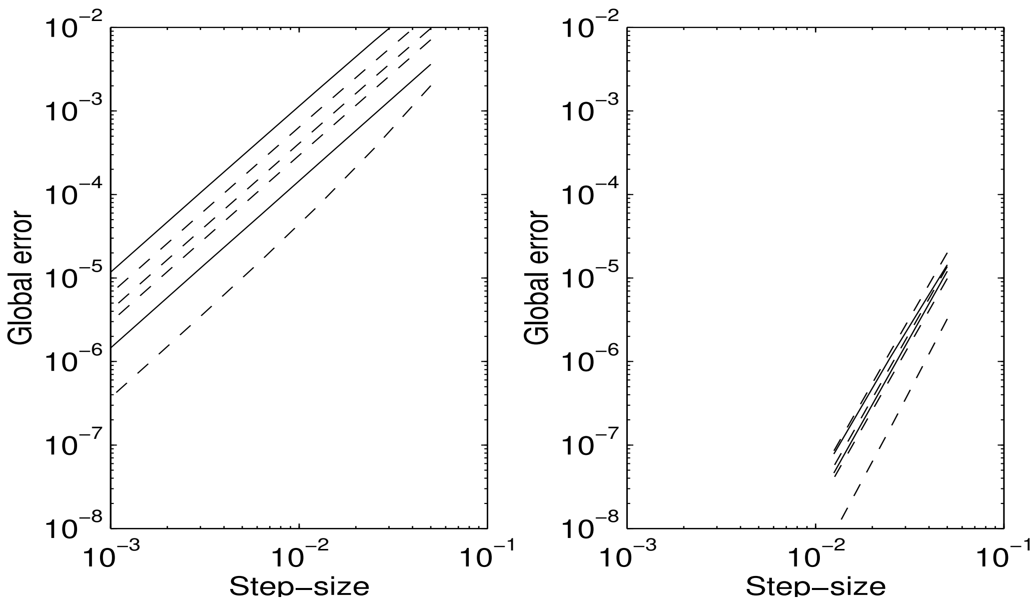

Figure 4.2: Estimates of the global error for 2nd and 4th order SIRK methods applied to the index 2 formulation (4.4) of the pendulum problem. Dashed lines represents index 0 and 1 components, solid lines represent index 2 components.

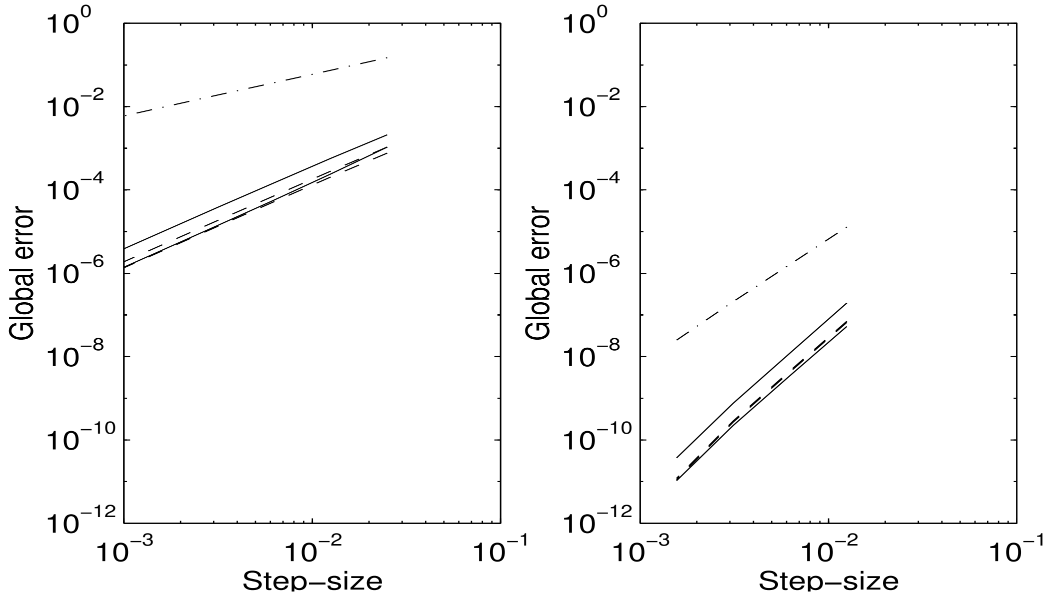

Figure 4.3: Estimates of the global error for 2nd and 4th order SIRK methods applied to the index 3 formulation (4.1) of the pendulum problem. The dashed lines represent index 0 and 1 components, solid lines represent index 2 components and the dash-dot line the index 3 component.

formulation (4.1), inconsistency in the constraint equation can be interpreted as a change in the length of the bar.

We compute consistent initial values by projecting the inconsistent initial values (x_0, y_0) onto the manifold given by $x^2 + y^2 - l^2 = 0$, parallel to the bar. Then we can use the ODE-formulation (4.5) to calculate a reliable reference solution.

4.2.1 An Index 2 Example

We have plotted the error estimates from chapter 2 applied to the index 2 formulation (4.4) of the pendulum problem in figure (4.4). The method used is the 4th order SIRK method. A reference solution is calculated from the consistent initial values described above, but the method is not restarted from these consistent values. That is why the local error does not always approach zero when the step-size decreases. The symbols are the same as in figure (2.2), but we have given three different plots for each step, corresponding to different norms. Motivated by the results of section 3.2.3 we will use the norm

$$\|\Delta u\| + \beta\|\Delta v\|, \tag{4.6}$$

where Δu is the error of the index 0 and 1 components, and Δv is the error of the index 2 components. The norm $\|\cdot\|$ is a weighted root mean square norm, given by

$$\|x\| = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{x_i}{w_i}\right)^2\right]^{\frac{1}{2}}, \tag{4.7}$$

where $x = [x_1, \dots, x_m]^T$ and $w = [w_1, \dots, w_m]^T$ is a weight vector given by user supplied tolerances. Similar results would be obtained with other choices of the norm.

In the first column of (4.4) we have used the norm (4.6) with $\beta = 0$, the second column corresponds to $\beta = h$, where h is the step-size. In the third column we do not distinguish between the components and use the norm

$$\left\|\begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix}\right\|. \tag{4.8}$$

The first step has a very different behaviour from the other steps, because we start from consistent initial values. In all cases the estimate (2.37) seems to be very sharp for the index 0 and 1 components, but also the cheaper estimate (2.40) seems to be useful. Apparently, the norm (4.8) can not be used for automatic step-size implementations. For the case $\beta = h$, the norm of the error approaches a constant as $h \to 0$, and therefore automatic step-size codes might fail. The choice $\beta = 0$ would probably give more robust codes for low tolerances.

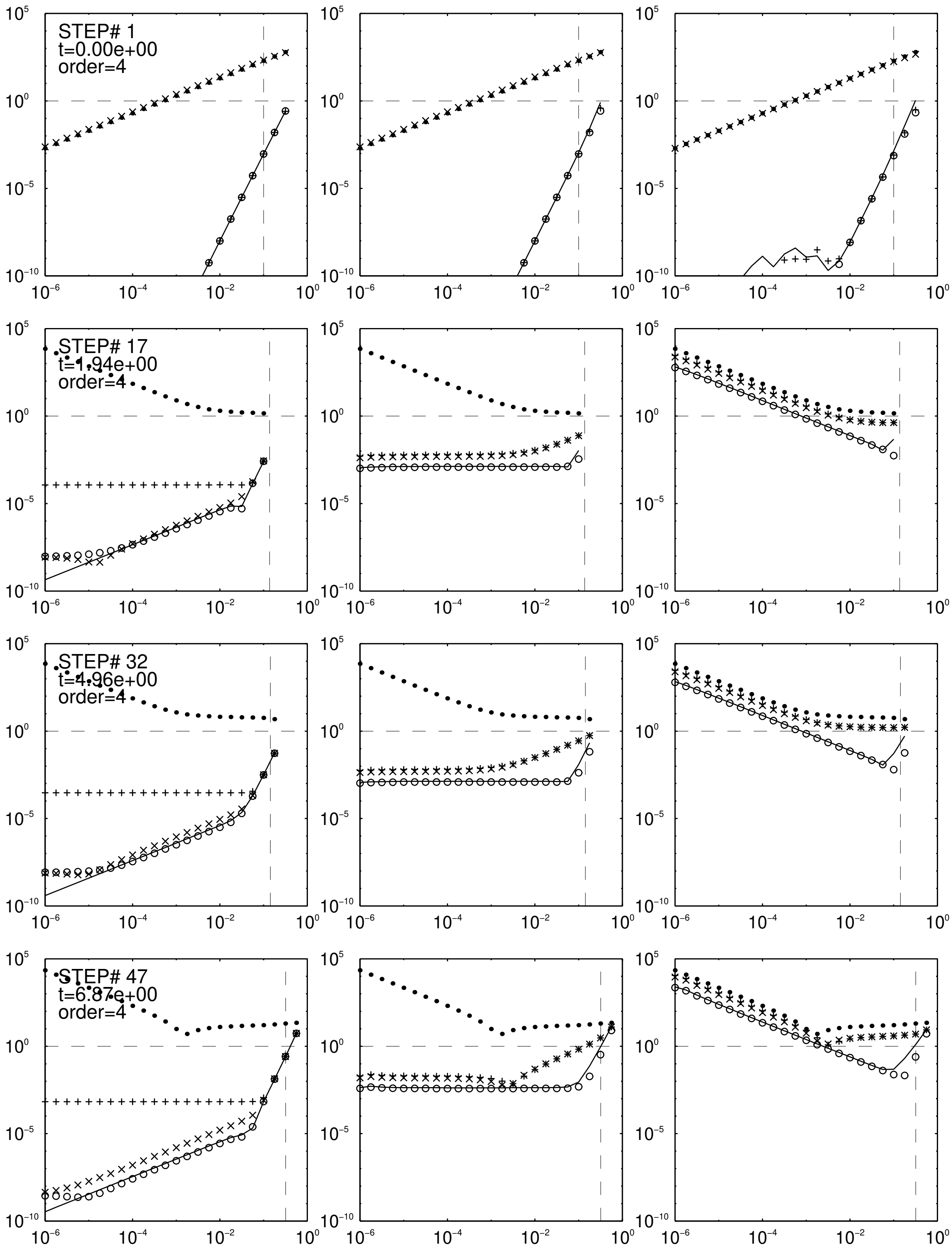

Figure 4.4: Local error estimates for the index 2 formulation (4.4) of the pendulum problem. Each row represent the same step, but with different norms.

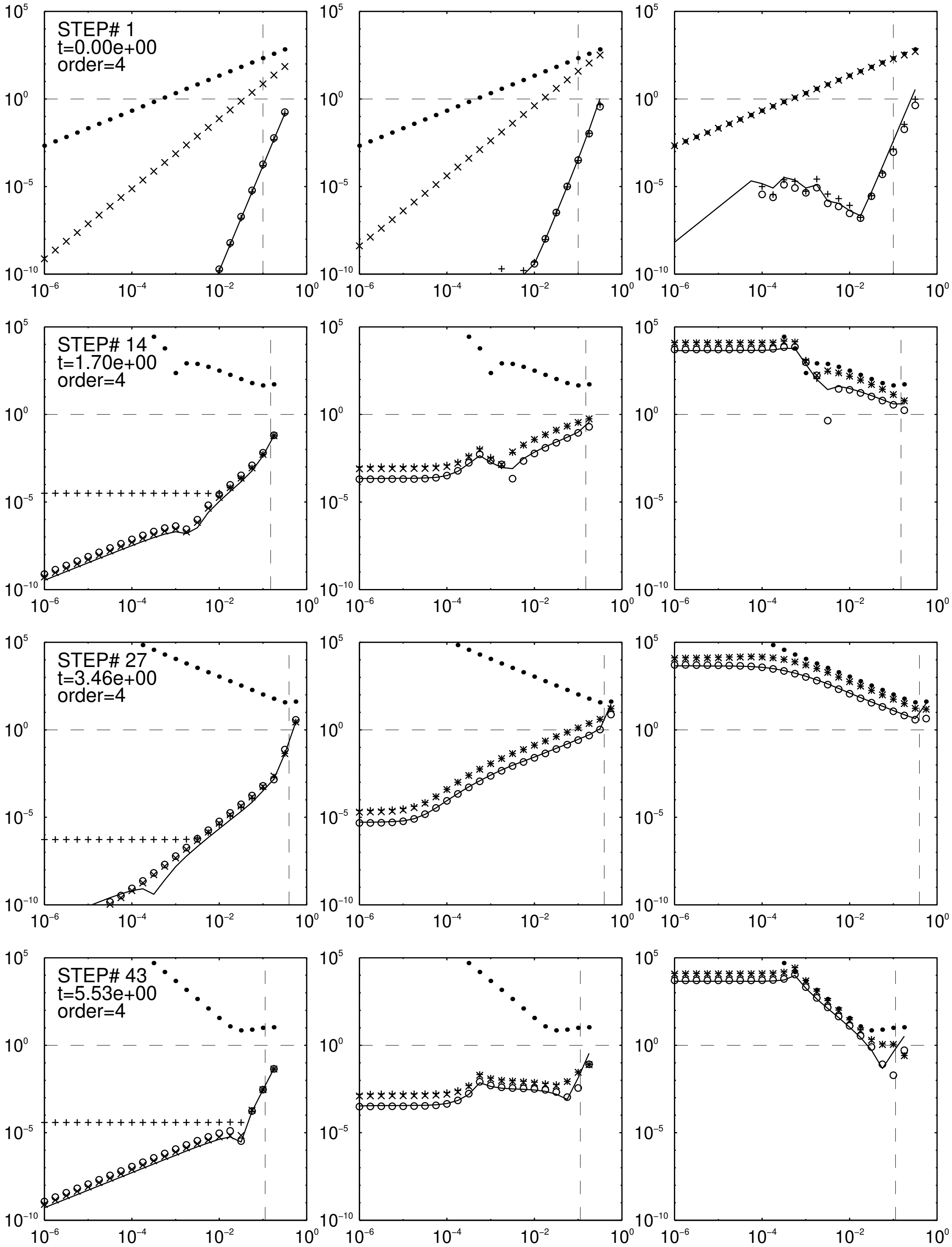

Figure 4.5: Local error estimates for the index 3 formulation (4.1) of the pendulum problem. Each row represent the same step, but with different norms.

4.2.2 An Index 3 Example

We have performed an experiment similar to the one in the last section, to the index 3 formulation (4.1) of the pendulum problem. Figure (4.5) shows the results, where the scaled norm (4.6) has been replaced by

$$\|\Delta u\| + \beta_1\|\Delta v\| + \beta_2\|\Delta w\|, \tag{4.9}$$

where Δu is the error of the index 0 and 1 components, Δv is the error of the index 2 components and Δw is the error of the index 3 components.

Now the plots in the first column of figure (4.5) corresponds to the above norm with $\beta_1 = \beta_2 = 0$, the second column corresponds to the choices $\beta_1 = h$ and $\beta_2 = h^2$, and the last column to the norm

$$\left\| \begin{bmatrix} \Delta u \\ \Delta v \\ \Delta w \end{bmatrix} \right\|.$$

Again, the first choice seem to be the safest for automatic step-size implementation, at least when high accuracy is needed.

4.3 STRIDE++

The code STRIDE++ is a SIRK solver for problems of the implicit form $\Phi(y', y, t) = 0$. We have used C++ for the implementation, and the development is based on the FORTRAN code STRIDE, written by K. Burrage, J. C. Butcher and F. H. Chipman. An overview of STRIDE++ can be found in appendix A and a detailed description of the code in appendix B.

4.3.1 The Automatic Step-size Control

The FORTRAN version of STRIDE is implemented with automatic selection of both step-size and order. The choice of order is based on estimates of the error corresponding to SIRK methods of one order higher and one order lower than the method used in the previous step . Numerical experiments show that these estimates do not work well for differential algebraic equations. Testing and improvement of the estimates was unfortunately beyond the scope of this report, and therefore an alternative constant order code has been implemented.

We have used the ideas of Gustafsson, described in [8] section IV.8 p. 124, to implement the step-size mechanism for STRIDE++.

We assume that the local error can be written as

$$||E_{n+1}|| = C_n h_n^{p+1}. \tag{4.10}$$

The strategy (2.26) is based on the assumption that $C_{n+1} \approx C_n$. Following Gustafsson, we now instead assume that $\log C_n$ is a linear function of n. In other words $\log C_{n+1} - \log C_n$ is constant or equivalently,

$$\frac{C_{n+1}}{C_n} \approx \frac{C_n}{C_{n-1}}. \tag{4.11}$$

If we now insert C_n and C_{n-1} from (4.10), and $C_{n+1} h_{\text{new}}^{p+1} = 1$ into (4.11) we get

$$h_{\text{new}} = \beta h_n \left(\frac{1}{||E_{n+1}||} \right)^{1/(p+1)} \cdot \frac{h_n}{h_{n-1}} \left(\frac{||E_n||}{||E_{n+1}||} \right)^{1/(p+1)}, \tag{4.12}$$

where β is a safety factor $0 < \beta \leq 1$.

Furthermore we will force the new step-size h_{new} to lie in the interval $[0.2h_n < h < 5h_n]$, and after divergent steps, we do not allow any increase at all in the step-size for three successive steps. We will use the local error estimate (2.37).

The estimate (2.22) can be used to stop Newton-Raphson iterations at an early stage, if the maximum number of iterations seems to be exceeded. We have tested the estimate, and it gives good results for some equations, but for other the estimate seems to be far to pessimistic, and leads to unnecessary rejection of steps. We have therefore chosen not to use the estimate.

4.3.2 Numerical Evaluation of the Jacobian

For some of the problems in the test set, an analytic Jacobian is not provided, and should be computed numerically by the solver. In STRIDE++ we compute an approximation of the k-th column of J by the finite difference formula

$$\frac{\Phi(t, y + \delta_k e_k, y') - \Phi(t, y, y')}{\delta_k}, \tag{4.13}$$

where e_j is a vector where the j-th component is 1 and all the other components are 0. We use the value of δ_k given in [1] p. 124,

$$\delta_k = \max(|y_k|, |hy'_k|, w_k)\sqrt{\tau},$$

where w_k is given by the user supplied tolerances ATOL and RTOL as

$$w_k = \mathrm{RTOL} \cdot |y_k| + \mathrm{ATOL}$$

and τ is the unit roundoff error in the computer. We use the same method to compute $M = \partial\Phi/\partial y'$.

Remark. In the BDF code DASSL the matrix $[\alpha M + J]$ is computed directly by finite differences, instead of first finding the approximations of M and J, and then add them together (see [1] p. 124). But in STRIDE++ the matrix M is needed to calculate the right hand side of the linear system (2.18), and we will have to approximate two matrices anyway.

4.3.3 Performance

We have tested STRIDE++ on some of the problems in the CWI Test Set [13]. This is a collection of stiff ODEs, DAEs of the form $My' = f(y, t)$ and IDEs of the form $\Phi(y', y, t) = 0$. Our favourite SIRK method is the 4-th order, and this order will be used in the tests below, unless something else is specified.

In addition we have run the same problems on one or more of the solvers:

DASSL is a BDF code for problems on the IDE form. The code is capable of solving some DAEs of index 1 and 2. More details about the code can be found in [1].

RADAU5 is based on the 3-stage, 5-th order RadauIIA method, and reads problems on the form $My' = f(y, t)$. The code can solve both index 1, 2 and 3 problems. Section IV.8 in [8] describes details about the code.

Following the standard from the CWI Test set we will use the following data to characterize a run:

- `solver`: The name of the numerical solver used.

- `rtol`: The supplied relative error tolerance. Uniform for all components.
- `atol`: The supplied absolute error tolerance. Uniform for all components.
- `h0`: The initial step-size (if required).
- `scd`: Denotes the minimum number of significant correct digits in the numerical solution at the endpoint, calculated by

$$scd := -\log_{10}(\text{max norm of the relative error in the endpoint })\ .$$

- `steps`: The total number of steps taken, including failed steps and divergent steps.
- `accept`: The number of accepted steps.
- `# f`: The number of derivative evaluations. For the RADAU5 and STRIDE++ code we have not counted the function evaluations used for local error estimation.
- `# Jac`: The number of Jacobian evaluations.
- `# LU`: The number of LU-factorizations (if different from # Jac). For STRIDE++ and DASSL the size of the matrix equals the number of variables m in the system. For RADAU5 each LU-factorization corresponds to factorization of one real and one complex system, each of size m.
- `# CPU`: The CPU time in seconds to perform a run.

All problems in this section have been taken from the CWI test set, and descriptions of the problems can be found in the www-page [13]. We have used the `F77` compiler for the FORTRAN code, and `g++` for the C++ code, both with the option `-O`. The runs have been performed on a Sun Ultra Enterprise 4000.

We want to emphasize that the STRIDE++ code should still be considered as a test version only. We have used most of our effort to make the code user friendly and reliable, and there is still much to be done to optimize the execution time. Therefore STRIDE++ will usually not compare very well to the other codes in terms of CPU-time.

The Transistor Amplifier

The Transistor Amplifier problem is an index 1 differential algebraic equation, describing a circuit. The system is of the form

$$My' = f(y).$$

The matrix

$$M = \begin{bmatrix} -C_1 & C_1 & & & & & & & \\ C_1 & -C_1 & & & & & & & \\ & & -C_2 & & & & & & \\ & & & -C_3 & C_3 & & & & \\ & & & C_3 & -C_3 & & & & \\ & & & & & -C_4 & & & \\ & & & & & & -C_5 & C_5 \\ & & & & & & C_5 & -C_5 \end{bmatrix}$$

is constant and singular, and $f(y)$ is nonlinear. A more detailed description of the problem can be found in [9] or at the CWI www-page [13].

We have solved the problem, using DASSL, RADAU5 and STRIDE++, for different tolerances. The run characteristics are:

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-4}	10^{-4}	10^{-4}	4.65	819	637	8756	702		0.99
	10^{-7}	10^{-7}	10^{-7}	6.20	1990	1850	19760	1292		2.42
DASSL	$10^{-4.0}$	$10^{-4.0}$		2.81	8056	5197	15006	5606		1.09
	$10^{-7.0}$	$10^{-7.0}$		5.02	39012	22690	74175	32460		5.43
RADAU5	$10^{-4.0}$	$10^{-4.0}$	$10^{-5.0}$	4.61	674	498	6295	495	674	0.22
	$10^{-7.0}$	$10^{-7.0}$	$10^{-9.0}$	6.81	1760	1547	17511	1537	1759	0.60

We can see that DASSL has difficulties solving this problem. STRIDE++ seems to perform well in terms of steps, function and Jacobian evaluations, but the CPU-time does not compare very well with RADAU5.

For STRIDE++ we have also plotted some details from the run corresponding to the tolerance 10^{-4} in figure (4.6). The first plot is the 8-th component of the solution vector. The next two plots show the step-size and the estimated local error as functions of the time t. The 4th plot shows the number of Newton-Raphson iterations, where divergent iterations and iterations that exceed the maximum limit of 10 iterations, are marked with a $[\times]$. In the last plot the dots represent Jacobian evaluations, which can be compared to the total number of steps (the solid line). The local error was never exceeded in the run.

The step-size control seems to be stable, and the number of Newton-Raphson iterations is usually low. The main difficulty seems to be the divergent steps, and iterations that exceed the maximum limit. The convergence difficulties are lessened, when the we decrease the tolerance.

The Fekete Problem

The origin of this problem is the computation of the elliptic Fekete points. The problem is contributed to the CWI test set by the parallel-IVP-algorithm group of CWI.

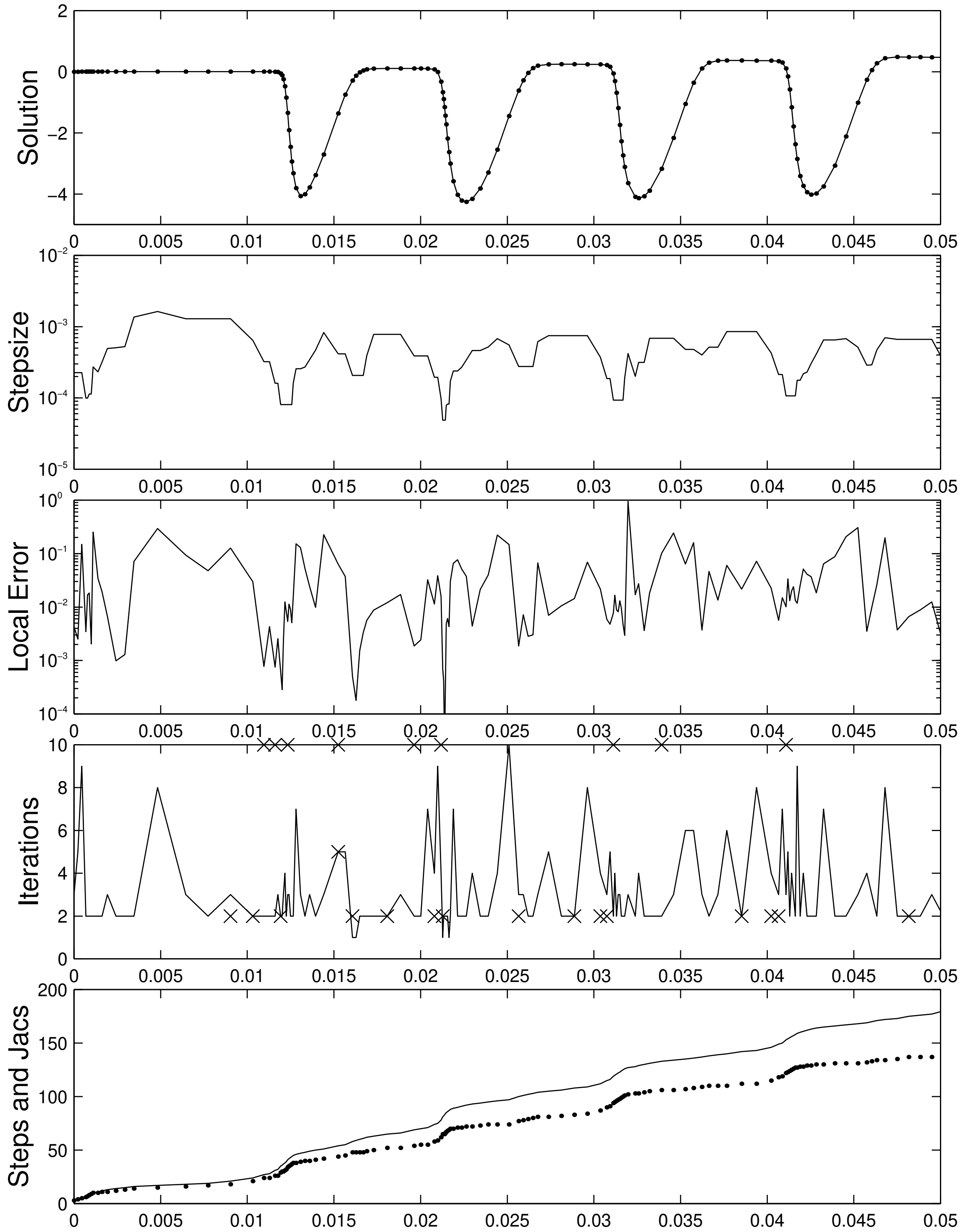

Figure 4.6: The 8-th solution component of the Transistor Amplifier problem, the step-size, local error, number of Newton-Raphson iterations and number of Jacobian evaluations for STRIDE++.

Let $\mathcal{S}^2$ denote the unit sphere in $\mathbb{R}^3$, and define the function V by

$$V(x) = \prod_{i<j} ||x_i - x_j||_2, \tag{4.14}$$

for any configuration of points $x = [x_1, \ldots, x_N]^T$, where $x_i \in \mathcal{S}^2$. The points $\widehat{x} = [\widehat{x}_1, \ldots, \widehat{x}_N]^T$, where V reaches its global maximum are called the elliptic Fekete points of order N.

This problem can be transformed to a differential algebraic system. Consider N particles on $\mathcal{S}^2$, and then invoke a repulsive force F_{ij} on the particles, where

$$F_{ij} = \frac{p_i - p_j}{||p_i - p_j||_2^{\gamma}},$$

and p_k is the Cartesian coordinates of particle number k. If we in addition impose an adhesion force on the particles, then the particles will reach a stationary configuration $\widehat{p} = [\widehat{p}_1, \ldots, \widehat{p}_N]^T$ after some time. We choose $\gamma = 2$, and we can then write the energy of the configuration $p(t) = [p_1, \ldots, p_N]^T$ as

$$E(p(t)) = \sum_{i<j} ||p_i(t) - p_j(t)||_2^{-1}.$$

Since the energy has a local minimum in the stationary configuration $\widehat{p}$ we have

$$\nabla_k E(p)|_{p=\widehat{p}} = \sum_{j \neq k} \frac{\widehat{p}_k - \widehat{p}_j}{||\widehat{p}_k - \widehat{p}_j||_2^2} = \xi_k \widehat{p}_k,$$

where ξ_k represent Lagrange multipliers, and ∇_k is the derivative with respect to p_k. If we differentiate the logarithm of (4.14), and apply the method of Lagrange, we obtain

$$\nabla_k \log(V(x))|_{x=\widehat{x}} = \sum_{j \neq k} \frac{\widehat{x}_k - \widehat{x}_j}{||\widehat{x}_k - \widehat{x}_j||_2^2} = \eta_k \widehat{x}_k,$$

which means that the computation of $\widehat{p}$ for $\gamma = 2$, gives the elliptic Fekete points.

More details on how to obtain an index 2 formulation of the mechanical system of the N particles are given in the description of problem 14 in the CWI test set [13].

We have solved the system from $t = 0$ to $t = 1000$ for N=20, which gives a system of 160 variables of the form

$$My' = f(y),$$

where the constant mass matrix M is given by

$$M = \begin{bmatrix} I_{120} & 0 \\ 0 & 0 \end{bmatrix}.$$

Solving the system numerically leads to a phenomenon, which in the CWI description is called numerical bifurcation. The paths of the individual particles may differ significantly for different choices of the error tolerance. Therefore we do not compare the configuration of particles against a reference solution. Instead the CWI test set provides an accurate value of V, given by (4.14), and we use this to compute the value of `scd`.

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-2}	10^{-2}	10^{-2}	7.16	170	137	1512	112		25.37
	10^{-4}	10^{-4}	10^{-4}	8.87	158	155	1556	84		21.42
	10^{-7}	10^{-7}	10^{-5}	14.73	589	588	5488	128		51.00
RADAU5	$10^{-2.0}$	$10^{-2.0}$	$10^{-2.0}$	5.57	64	58	611	54	62	10.11
	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$	6.96	118	114	940	110	118	14.95
	$10^{-7.0}$	$10^{-7.0}$	$10^{-5.0}$	14.34	371	371	2960	332	351	38.59

We have used a full matrix for the Jacobian, so the LU-factorizations are costly. With the tolerance 10^{-2}, STRIDE++ has difficulties with slow convergence of the Newton-Raphson iteration, but for lower tolerances the code seems to run efficiently. RADAU5 usually performs more LU-factorizations than STRIDE++, but is still faster in terms of CPU-time.

Andrew's Squeezing Mechanism

This problem is an index 3 DAE of dimension 27, and describes the motion of 7 rigid bodies. An analytic Jacobian is not provided, so we have to compute an approximation by finite differences. We refer to the CWI test set for details about the problem. The run characteristics for STRIDE++ and RADAU5 are:

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-4}	10^{-4}	10^{-3}	0.99	99	76	2904	96		1.17
	10^{-7}	10^{-7}	10^{-3}	2.05	163	143	4852	160		1.97
RADAU5	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$	0.13	112	67	883	63	109	0.27
	$10^{-7.0}$	$10^{-7.0}$	$10^{-7.0}$	0.48	116	94	1340	89	116	0.37

The high number of function evaluation shows that STRIDE++ has difficulties with slow convergence of the Newton-Raphson iteration for this problem. We have included plots of the step-size, local error esimate, number of Newton-Raphson iterations and the number of Jacobian evaluations for the tolerance 10^{-4} in figure 4.7.

The symbol $[\times]$ means that the Newton-Raphson iteration diverged or did not converge in $k_{\max} = 10$ steps. The tolerance is never exceeded in the run. We see that we usually need at least 5 Newton-Raphson iterations in every step. This example is one of the reasons for choosing a large value for $k_{\max}$. Otherwise the performance gets a lot worse, if we are able to get a solution at all.

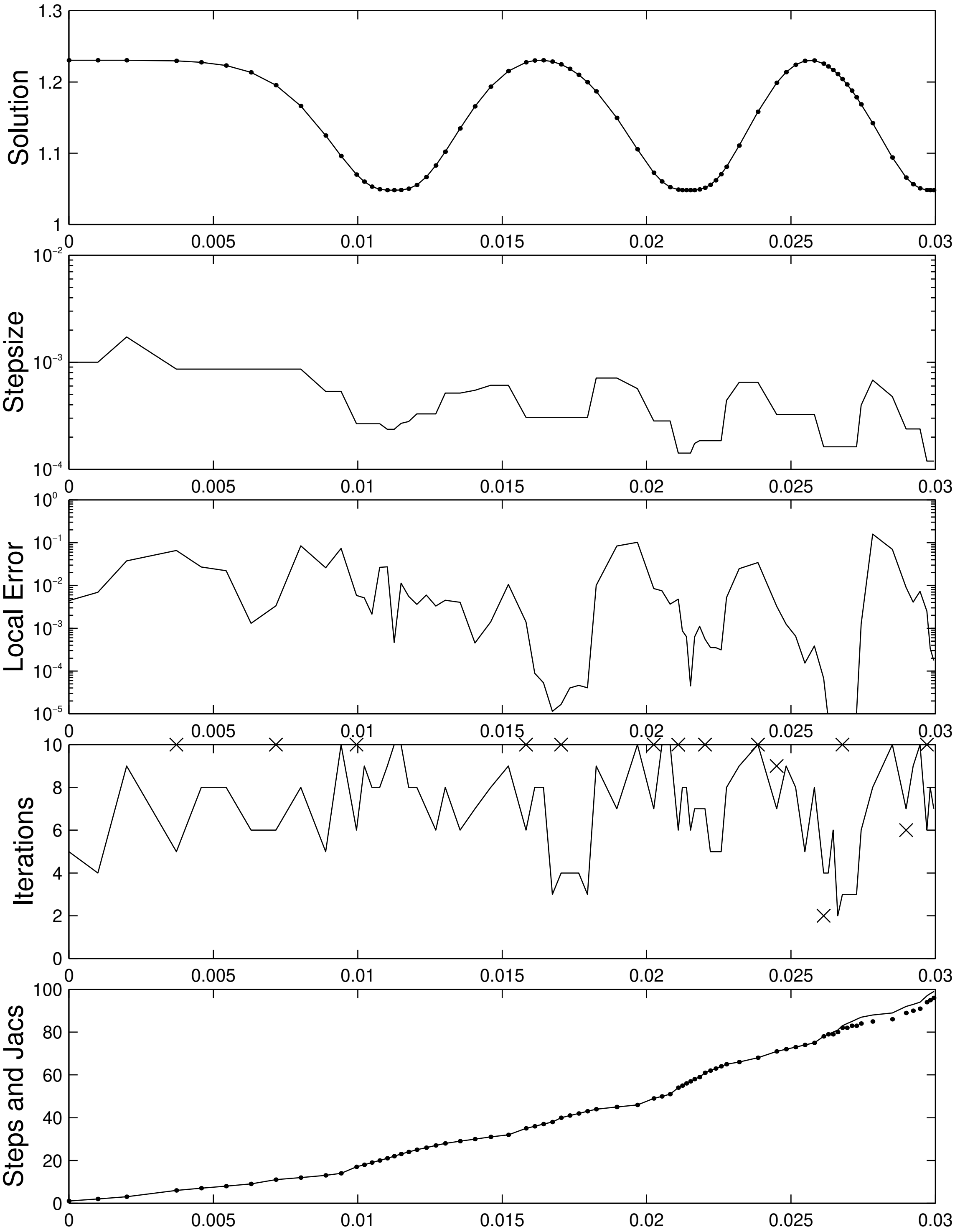

Figure 4.7: The 7-th solution component of Andrew's squeezing mechanism, the step-size, local error estimate, number of Newton-Raphson iterations and number of Jacobian evaluations for STRIDE++.

Other Differential Algebraic Equations

The following two differential algebraic systems are both modelling mechanical systems. We will here only include the run characteristic. See [13] for descriptions of the problems.

The Wheelset problem, index 2 IDE of dimension 17, Jacobian not provided.

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-4}	10^{-4}	10^{-4}	1.05	2763	2325	27384	1997		17.12
	10^{-6}	10^{-6}	10^{-4}	2.52	6905	5585	61488	4268		38.17
DASSL	$10^{-4.0}$	$10^{-4.0}$		0.13	5774	5054	9504	1057		4.02
	$10^{-6.0}$	$10^{-6.0}$		3.06	15329	13884	22704	1977		8.75

The Car Axis problem, index 3 DAE of dimension 10, Jacobian not provided.

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-4}	10^{-4}	10^{-4}	-0.25	123	116	2020	65		0.20
	10^{-7}	10^{-7}	10^{-5}	2.03	465	446	7568	175		0.74
	10^{-10}	10^{-10}	10^{-6}	3.99	1715	1693	25904	331		2.50
RADAU5	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$	-0.26	98	97	850	95	98	0.04
	$10^{-7.0}$	$10^{-7.0}$	$10^{-5.0}$	2.03	287	286	2551	283	287	0.12
	$10^{-10.0}$	$10^{-10.0}$	$10^{-6.0}$	2.59	879	879	8100	856	877	0.37

Ordinary Differential Equations

We have also included the results from applying STRIDE++ to some of the ordinary differential equations in the test set.

The Chemical AKZO problem models a chemical reaction, and the dimension of the system is 6. We have used the 6-th order SIRK method of STRIDE++.

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-4}	10^{-4}	10^{-4}	5.09	34	31	558	27		0.04
	10^{-7}	10^{-7}	10^{-7}	6.64	49	46	666	34		0.05
	10^{-10}	10^{-10}	10^{-10}	8.92	79	79	1116	39		0.08
DASSL	$10^{-4.0}$	$10^{-4.0}$		3.98	48	46	72	13		0.00
	$10^{-7.0}$	$10^{-7.0}$		5.76	165	160	225	24		0.01
	$10^{-10.0}$	$10^{-10.0}$		8.00	401	396	474	32		0.03
RADAU5	$10^{-7.0}$	$10^{-7.0}$	$10^{-7.0}$	6.22	37	34	292	28	37	0.01
	$10^{-10.0}$	$10^{-10.0}$	$10^{-10.0}$	8.06	85	85	649	54	65	0.01

RADAU5 fails for the tolerance 10^{-4}.

The Ring Modulator problem describes a circuit, and the dimension of the system is 15.

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-4}	10^{-4}	10^{-4}	2.39	2272	1965	26760	1260		4.11
	10^{-7}	10^{-7}	10^{-7}	4.55	7284	7045	61252	1841		10.10
	10^{-10}	10^{-10}	10^{-10}	6.87	27507	27323	222228	2805		35.89
DASSL	$10^{-4.0}$	$10^{-4.0}$		1.59	6288	6072	10380	440		1.28
	$10^{-7.0}$	$10^{-7.0}$		3.73	18350	18114	27525	504		3.47
	$10^{-10.0}$	$10^{-10.0}$		6.02	45570	45293	62202	742		7.99
RADAU5	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$	2.47	2245	1787	14754	998	2017	1.28
	$10^{-7.0}$	$10^{-7.0}$	$10^{-7.0}$	4.78	5682	5228	37531	1557	3459	2.85
	$10^{-10.0}$	$10^{-10.0}$	$10^{-10.0}$	6.94	16222	15806	111461	2939	9415	8.16

The EMEP problem is also a model of a chemical reaction. The system is of dimension 66, and the Jacobian is a full matrix.

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-2}	10^{0}	10^{-6}	2.76	459	366	5452	388		10.84
	10^{-4}	10^{0}	10^{-6}	3.83	586	506	7328	485		14.33
	10^{-6}	10^{0}	10^{-6}	4.94	1362	1169	12232	786		24.38
DASSL	$10^{-2.0}$	$10^{0.0}$		1.35	738	677	1388	165		1.71
	$10^{-4.0}$	$10^{0.0}$		2.99	1857	1789	3205	199		3.29
	$10^{-6.0}$	$10^{0.0}$		5.00	4213	4035	6713	442		7.09
RADAU5	$10^{-2.0}$	$10^{0.0}$	$10^{-7.0}$	2.39	328	239	3309	224	325	5.57
	$10^{-4.0}$	$10^{0.0}$	$10^{-7.0}$	2.67	493	407	4726	378	479	8.17
	$10^{-6.0}$	$10^{0.0}$	$10^{-7.0}$	4.38	942	821	8138	756	905	15.03

The HIRES problem originates from plant physiology, and is a system of 8 variables.

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-4}	10^{-4}	10^{-7}	1.48	56	51	544	44		0.04
	10^{-7}	10^{-7}	10^{-9}	4.59	104	104	916	48		0.08
	10^{-10}	10^{-10}	10^{-10}	6.93	364	364	2896	57		0.24
DASSL	$10^{-4.0}$	$10^{-4.0}$		1.07	108	99	176	32		0.01
	$10^{-7.0}$	$10^{-7.0}$		3.36	315	311	459	40		0.03
	$10^{-10.0}$	$10^{-10.0}$		7.02	1097	1081	1502	47		0.10
RADAU5	$10^{-4.0}$	$10^{-4.0}$	$10^{-7.0}$	1.15	43	35	314	22	43	0.01
	$10^{-7.0}$	$10^{-7.0}$	$10^{-9.0}$	4.31	79	72	684	31	61	0.02
	$10^{-10.0}$	$10^{-10.0}$	$10^{-10.0}$	7.15	199	199	1660	61	97	0.04

The Pollution problem is a system of dimension 20, and models a chemical reaction.

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-4}	10^{-4}	10^{-4}	4.56	22	21	184	21		0.05
	10^{-7}	10^{-7}	10^{-7}	5.42	42	42	332	27		0.08
	10^{-10}	10^{-10}	10^{-10}	7.50	131	131	1032	36		0.23
DASSL	$10^{-4.0}$	$10^{-4.0}$		1.97	37	36	57	14		0.01
	$10^{-7.0}$	$10^{-7.0}$		4.13	135	135	192	23		0.03
	$10^{-10.0}$	$10^{-10.0}$		5.55	368	365	497	40		0.09
RADAU5	$10^{-4.0}$	$10^{-4.0}$	$10^{-4.0}$	1.29	22	18	156	15	21	0.02
	$10^{-7.0}$	$10^{-7.0}$	$10^{-7.0}$	3.78	32	29	227	21	32	0.03
	$10^{-10.0}$	$10^{-10.0}$	$10^{-10.0}$	7.39	65	65	458	31	46	0.05

The NAND Gate is a model of a circuit. The system is of dimension 14, and is of the implicit form $C(y(t))y' = f(y(t), t)$. Therefore only the codes DASSL and STRIDE++ can be used to solve the problem. The Jacobian is not provided, and must therefore be evaluated numerically.

solver	rtol	atol	h0	scd	steps	accept	# f	# Jac	# LU	CPU
STRIDE++	10^{-4}	10^{-4}	10^{-4}	2.15	440	354	4916	333		1.78
	10^{-7}	10^{-7}	10^{-4}	5.68	2265	1633	18232	1596		7.78
DASSL	$10^{-4.0}$	$10^{-4.0}$		2.00	1058	899	1674	335		0.50
	$10^{-7.0}$	$10^{-7.0}$		5.90	4084	3724	5835	878		1.57

Chapter 5

Conclusion

There is still some work to be done before STRIDE++ can be an serious alternative to other solvers of differential algebraic equations. Below we have summarized some improvements that can be made to the code.

Order and Step-size Strategy

The FORTRAN version of STRIDE is implemented as a variable order and step-size code, thus exploiting one of the main advantages of SIRK methods. The same strategy is rewritten as a part of STRIDE++, but does not work well together with the new error estimates. Therefore a constant order strategy is used in the current version. We believe that it is possible to improve this strategy further. For some problems the number of steps rejected because of divergence or slow convergence of the Newton-Raphson iterations, is quite high. Probably, some of these rejected steps could have been avoided with a better step-size strategy.

Optimizing the Code

Compared to other solvers, like DASSL and RADAU5, STRIDE++ seems to spend more CPU-time per function evaluation and/or Jacobian evaluation and decomposition. In the current version all parts of the computation is implemented in C++, and not all parts of the code have been analyzed and optimized for maximum speed. The computation time would probably decrease significantly if optimized subroutines (f.ex. LAPACK) were used for some of the linear algebra parts. The construction of STRIDE++ allows users to use their own linear algebra for the evaluation, decomposition and backsubstitution of the matrix $[(1/H)M + J]$, and thus it is possible to test the code with optimized linear algebra, using the current version of STRIDE++. The time aspect of this report has not allowed this type of testing.

Final Words

We have in this report shown that SIRK methods are capable of solving many differential algebraic equations, and that sharp estimates of the local error can be computed in a reasonably cheap way. The overall performance we have achieved, measured in CPU time, is usually behind other solvers like DASSL and RADAU5. But in terms of function and Jacobian evaluations the difference is smaller, and sometimes STRIDE++ seems to perform equal to, or maybe better than other codes. (See for example the Transistor Amplifier and the Fekete problems.) With a more efficient and optimized implementation, STRIDE++ should be a reasonable alternative to other solvers of differential algebraic and stiff ordinary differential equations.

Appendix A

STRIDE++ Overview

STRIDE++ is an experimental version of the stiff ODE solver STRIDE by K. Burrage, J.C. Butcher and F.H. Chipman. The new version is capable of solving equations on implicit form and some DAEs of index 1, 2 and 3. Since the author is not familiar with software development in FORTRAN, C++ has been used for the implementation.

We have used parts of the software package `Diffpack` as a basis for the development. `Diffpack` is a C++ library for solution of PDE's, and contains the following packages:

Basic Tools provides basic arrays, error handling, memory management and generic I/O.

Linear Algebra Tools provides matrices and vectors, and solvers for linear and nonlinear algebraic equations.

DpKernel, DpUtil and DpAppl contains basic classes for finite element and finite difference methods.

We have used *Basic Tools* and *Linear Algebra Tools* for the implementation. More information, and a public release version of `Diffpack` can be found at [14]. [15] gives an overview of the `Diffpack` functionality we have used for STRIDE++, and [16] discusses the file organization and `make` features.

A.1 Construction

In this section we will assume that the reader is familiar with the basics of C++ and object oriented design. Information about this can be found in any C++ textbook.

Figure (A.1) shows the relationship between the classes in STRIDE. Solid lines represent *inheritance*, for example class `Stride` inherits the proportied of class

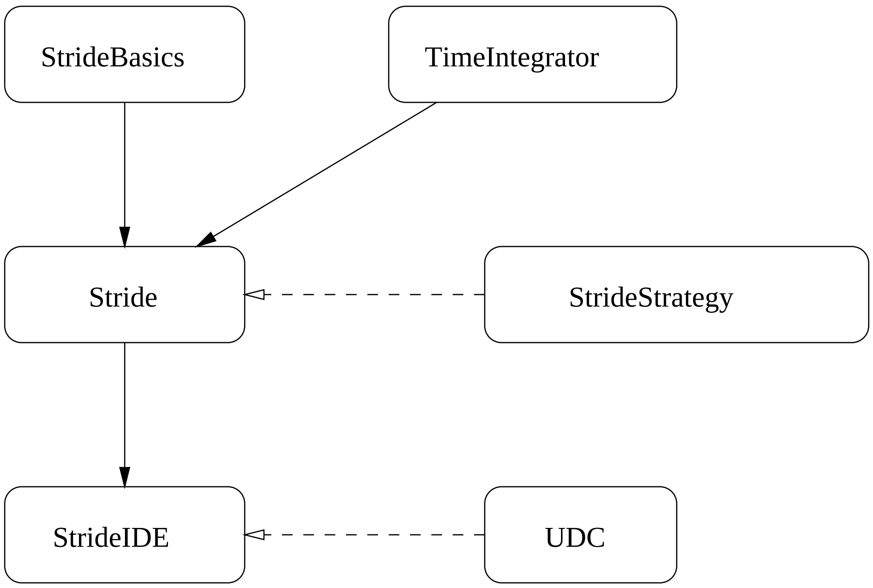

Figure A.1: The class hierarchie in STRIDE++

`StrideBasics` and class `TimeIntegrator`. The dotted lines means "contained in". For example class `StrideStrategy` is contained in class `Stride`.

The hierarchy is designed to be a flexible toolbox for implementation of IVP solvers (initial value problems), and especially SIRK methods. The class `TimeIntegrator` and also the problem classes (represented by `UDC` in the figure (A.1)) are general classes that may be used for other purposes than STRIDE. `StrideBasics` provides matrices and other data needed for implentation of SIRK methods. `Stride` contains functions for performing a SIRK step, getting starting values for the Newton-Raphson iteration and interpolating output values. It also connects the stepsize and order selection class `StrideStrategy` to the hierarchy. Class `Stride` does not contain the code for performing Newton-Raphson iterations, and not the problem definition. This is left to the class `StrideIDE`, which performs iterations on problems on the form $\Phi(y', y, t) = 0$. Because class `Stride` is independent of the problem representation, it is possible to derive several classes from `Stride`, each optimized for different kind of problem representations.

A.1.1 The Problem Hierarchy

We have provided interfaces for problems on the following forms:

ODEs (ordinary differential equations) are differential equations on the form $y' = f(y, t)$, and the interface is provided by class `ODESolverUDC`.

DAEs (differential algebraic equations) are systems on the form $My' = f(y, t)$, where M is a constant matrix, possibly singular. Class `DAESolverUDC` is

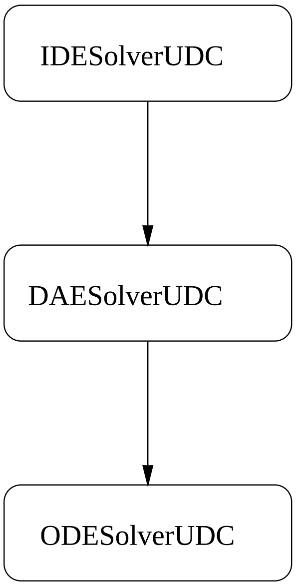

Figure A.2: The hierarchy of problems

the interface class for these problems.

IDEs (implicit differential equations) are systems on the general form $\Phi(y', y, t) = 0$. The corresponding problem class is `IDESolverUDC`.

Note that this classification is independent of the index of the equation. We are only interested in how the equations are expressed.

Since it is possible to convert problems on the ODE form to DAE form (setting $M = I$) and DAEs to IDEs ($\Phi(y', y, t) = My' - f(y, t)$), we arrange the problems in the hierarchy (A.2). We can easily implement automatic conversion from ODEs to DAEs and from DAEs to IDEs.

A.2 Example

A.2.1 The van der Pol equation

The problem is a nonlinear equation describing a circuit. It can be written in the form

$$\begin{aligned} y_1' &= y_2, \\ \epsilon y_2' &= (1 - y_1^2) y_2 - y_1. \end{aligned} \tag{A.1}$$

We multiply the last equation by $1/\epsilon$ to get the system on standard ODE form, and we can implement a class `ODEVanderPol` derived from `ODESolverUDC`. The declaration of `ODEVanderPol` should be written in a file `ODEVanderPol.h`:

```
#ifndef ODEVanderPol_H
#define ODEVanderPol_H

#include <StrideUDC.h>

/*<ODEVanderPol:*/
class ODEVanderPol : public ODESolverUDC {
private:
  real epsilon_;
public:
  ODEVanderPol() : ODESolverUDC(2, dpTRUE) { epsilon_ = 1e-6; }

  virtual void evalRHS(Vec(real)& f, Vec(real)& y, real t);
  virtual void initValues(Vec(real)& y, real &t);

  virtual void evalJacobian(Vec(real)& y, real t, Mat(real)& jac);
};
/*>ODEVanderPol:*/

/*Class: ODEVanderPol

NAME: ODEVanderPol

SYNTAX: @ODEVanderPol

KEYWORDS:

DESCRIPTION:

  This problem is a simple ODE describing a circuit.  The problem is stiff,
  and the dimension is 2.

AUTHOR:

  Erik Hamran Nilsen, NTNU Trondheim, Norway

End: */

#endif
```

The constructor `ODEVanderPol()` is implemented *inline* and the two arguments inform `ODESolverUDC` that this is a system of 2 equations, and an analytic jacobian is provided. The constant ϵ is set to 10^{-6}. The other member functions are implemented in a file `ODEVanderPol.C`:

```
#include <ODEVanderPol.h>

//-----------------------------------------------------------------------------
void
ODEVanderPol :: evalRHS(Vec(real)& f, Vec(real)& y, real t)
//-----------------------------------------------------------------------------
{
  f(1) = y(2);
  f(2) = ((1.0 - pow2(y(1)))*y(2) - y(1))/epsilon_;
}
```

```
//-----------------------------------------------------------------------------
void
ODEVanderPol :: initValues(Vec(real)& y, real &t)
//-----------------------------------------------------------------------------
{
  y(1) = 2.0;
  y(2) = -0.6;
  t = 0.0;
}

//-----------------------------------------------------------------------------
void
ODEVanderPol :: evalJacobian(Vec(real)& y, real t, Mat(real)& jac)
//-----------------------------------------------------------------------------
{
  jac(1,1) = 0.0;
  jac(1,2) = 1.0;
  jac(2,1) = ( -2.0 * y(1) * y(2) - 1.0)/epsilon_;
  jac(2,2) = (1.0 - pow2(y(1)))/epsilon_;
}
```

Now all we have to do is to implement a `main` routine that creates a solver and a problem, reads user defined tolerances from the command line, and prints the solution:

```
#include <StrideIDE.h>
#include <ODEVanderPol.h>

int main(int nargs, const char** args) {

  initDIFFPACK (nargs, args);

  real atol;
  initFromCommandLineArg("-a", atol, 1e-4);
  real rtol;
  initFromCommandLineArg("-r", rtol, 1e-4);
  real init_stepsize;
  initFromCommandLineArg("-i", init_stepsize, 1e-4);


  ODEVanderPol problem;

  StrideIDE solver(problem, init_stepsize);
  solver.setTolerance(rtol, atol);

  int n = problem.size();
  Vec(real) y(n);
  real t;

  s_o << "time       y1        y2\n";
  int i;
  for(i=1; i<20; i++) {
    t = 0.1 * i;
    solver.advance(t, y);
    s_o << t << "  ";
    y.print(s_o); s_o << "\n";
```

```
  }
  return 0;
}
```

The program is started by

```
app -a atol -r rtol -i init_stepsize
```

where the arguments are absolute tolerance, relative tolerance and an initial stepsize guess.

Appendix B

STRIDE++ Class Descriptions

B.1 Abstract Solver of Initial Value Problems

NAME TimeIntegrator

INCLUDE `include "TimeIntegrator.h"`

SYNTAX

```
//-----------------------------------------------------------------------------
class TimeIntegrator : public virtual HandleId
//-----------------------------------------------------------------------------
{
private:

  int size_;   // number of variables in system

  // ----- Storage for the tolerances -----

  Vec(real) rtol_;
  Vec(real) atol_;
  Vec(real) temp_tol_;
  Norm_type tol_norm_type_;  // one of l2 (discrete), L2 (cont), Linf

  // ----- Storing values at beginning and end of step -----

  BooLean valid_initial_values_;
  Vec(real) y0_;
  real      t0_;
  Vec(real) y_;
  real      t_;

  // ----- Performance data -----

  int n_steps_;
  int n_fail_;
  int n_diverged_;
  int n_jac_eval_;
  int n_func_eval_;

  CPUclock clock_;
  real CPUtime_;

  // ----- parameters for plotting the solution -----

  Os plot_file_;
  BooLean do_plot_;
  VecSimple(int) plot_components_;

  // ----- Member functions -----

protected:
  void        plotValues         ();
  void        updateToleranceNorm();
```

```
  void         stepAccepted       (real& t, Vec(real)& y);
  virtual void interpolate        (real t_end, Vec(real)& y) = 0;
public:
  // ----- Initialization -----

  TimeIntegrator(int n);
  void    init            (TimeIntegrator& solver);
  BooLean validInitValues () const { return valid_initial_values_; }
  void    setInitValues   (real t0, Vec(real)& y0);

  // ----- The Solution -----

  int               size         () const { return size_; }
  inline Vec(real)& getValues    () const;
  inline Vec(real)& getLastValues() const;
  inline real&      getTime      () const;
  inline real&      getLastTime  () const;
  real              getStepsize  () const { return t_ - t0_; }

  void setPlot    (Os& ofile, const VecSimple(int)& comp);
  void removePlot ();

  // ----- Performance -----

  void addAcceptedStep()      { n_steps_++; }
  void addRejectedStep()      { n_fail_++; }
  void addDivergedStep()      { n_diverged_++; }
  void addFunctionEval()      { n_func_eval_++; }
  void addFunctionEval(int n) { n_func_eval_ += n; }
  void addJacobianEval()      { n_jac_eval_++; }

  int  getAcceptedStep() const { return n_steps_; }
  int  getTotalSteps  () const { return n_steps_ + n_fail_ + n_diverged_; }
  int  getFunctionEval() const { return n_func_eval_; }
  int  getJacobianEval() const { return n_jac_eval_; }
  real getCPUtime     () const { return CPUtime_; }

  void printStats(Os stream);

  // ----- Local Error Tolerances and Norms -----

  void         setTolerance (real rtol, real atol);
  void         setTolerance (const Vec(real)& rtol, const Vec(real)& atol);
  Vec(real)&   getTolerance () { return temp_tol_; }

  virtual real toleranceNorm (Vec(real)& y);
  virtual real newtonNorm (Vec(real)& y);

  Norm_type    getToleranceNormType () { return tol_norm_type_; }
  void         setToleranceNormType (Norm_type norm) { tol_norm_type_ = norm; }

  // ----- Advancing the Solution -----

  virtual BooLean advance()=0;                        // single step
  virtual void    advance (real t_end, Vec(real)& y); // until t = t_end

};
```

KEYWORDS solver, initial value problem

DESCRIPTION

TimeIntegrator is a base class for IVP-solvers (Initial Value Problems). The class stores solution vectors, calculates a scaled norm based on user supplied tolerances, plots solutions and keeps track of the number of function evaluations, jacobian decompositions et.c.

CONSTRUCTORS AND INITIALIZATION

The constructor takes one argument - the number of variables in the system to solve.

MEMBER FUNCTIONS

`size` - returns the number of variables in the problem.

`getValues` - returns the solution vector at the end of the current step.

`getTime` - returns the time at the end of the current step.

`setPlot` - starts saving of solution values at the end of each step. A vector specifying the wanted components should be supplied.

`removePlot` - turns off the recording of solution values.

`printStats` - prints a table of statistics from the last `advance` call, including number of accepted and rejected steps, number of Jacobian evaluations and CPU-time.

`setTolerance` - is used to set the relative and absolute error tolerances for the numerical solution. If the arguments are scalars, uniform tolerances are used for all components. By supplying vectors, the user can control the tolerance of each component individually.

`getTolerance` - returns the local tolerance $t_i = a_i + r_i \cdot |y_i| \quad , \; i = 1, \ldots, m$, where $y = (y_i)$ is the solution from the last step and $a = (a_i)$ and $r = (r_i)$ are absolute and relative tolerances.

`toleranceNorm` - takes an error vector e as input and calculates the norm $\max_{1 \leq i \leq m} \frac{|e_i|}{t_i}$, where $t = (t_i)$ is the local tolerance.

`newtonNorm` - calls `TimeIntegrator::toleranceNorm`, but may be overloaded if the Newton-Raphson iterations requires a different norm.

`advance` - calculates the solution at a point. The instance without parameters will automaticly choose a stepsize and perform one step. The other instance will calculate the solution at a specifikk time, by repeated calls to `advance`, and a call to `interpolate` in the end.

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.2 STRIDE - Stiff Runge-Kutta Integrator for Differential Equations

B.2.1 Utilities for Implementation of SIRK methods

NAME StrideBasics

INCLUDE `include "StrideBasics.h"`

SYNTAX

```
//-----------------------------------------------------------------------------
class StrideBasics : public virtual HandleId
//-----------------------------------------------------------------------------
{
private:
  real accuracy_;
  int max_stage_;

  VecSimplest(Vec(real)) lagrre_zeros_;

protected:

  // ----- Storage of RK-matrices and transformation matrices -----

  VecSimplest(Mat(real)) A_;
  VecSimplest(Mat(real)) Ainv_;
  VecSimplest(Mat(real)) T_;
  VecSimplest(Mat(real)) Tinv_;

  VecSimplest(Vec(real)) rinv_;

  VecSimplest(int) lz_choice_;
  ArrayGen(real) error_factors_;

  real initMachineAccuracy();
  void initLaguerreZeros();
  void initErrorFactors();
  void initMatrices();
  void initErrorCoeff();

  void build();

public:
  StrideBasics(int max_s);

  int  getMaxS     () { return max_stage_; }
  real getAccuracy () { return accuracy_; }

  real getErrorCoeff  (int s, int i)           { return rinv_(s)(i); }
  void laguerreEval   (real x, Vec(real)& p, BooLean diff = dpFALSE);
  real laguerreOutput (int S);
  real laguerreZero   (int degree, int n) { return lagrre_zeros_(degree)(n); }
```

```
};
```

KEYWORDS stride, transformation, laguerre

DESCRIPTION

The purpose of this class is to provide basic tools for implementation of SIRK methods (Singly Implicit Runge-Kutta methods). Among these tools are evaluation of Laguerre polynomials and their zeros, calculation of transformation matrices and the A matrix of the Butcher tablau, for SIRK methods of different orders.

CONSTRUCTORS AND INITIALIZATION

The constructor needs one parameter - the maximum number of stages for the SIRK methods. Current implementation does not support more than 10 stages.

MEMBER DATA

`A_` - contains the matrices of the Butcher tablau, for methods of 1 to `getMaxS` stages.

`Ainv_` - contains the inverses of the matrices in `A_`.

`T_` - contains the transformation matrices associatet with the `A_` matrices.

`Tinv_` - contains the inverse of the matrices in `T_`.

`lz_choice_` - is a list of integers that defines the λ parameter of the methods. If `lz_choice_(s) = i`, we will use $\lambda = 1/c_i$ as parameter for the s-stage SIRK method, where c_i is the i-th zero of L_s. The values are selected to make the methods almost A-stable ($A(\alpha)$-stable) and to minimize the error constant.

`error_factors_` - contains scalars used for error estimation

MEMBER FUNCTIONS

We will use the notation $L_i(x)$ for the Laguerre polynomial of degree i evaluated at x.

`getMaxS` - returns maximum number of stages allowed.

`getAccuracy` - returns the calculated machine accuracy.

`laguerreEval` - computes the values of the laguerre polynomials of degree 1 to s at a point x. s is the size of the argument vector p where the values are stored. If the flag is set to *dpFALSE*, the difference $p_i = L_i - L_{i-1}$ is also computed.

`laguerreOutput` - chooses the value of λ for the method of s stages.

`laguerreZero` - returns the n-th zero of a Laguerre polynomial of degree $1, \ldots,$ `getMaxS`.

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.2.2 General SIRK Solver

NAME Stride

INCLUDE `include "Stride.h"`

SYNTAX

```
class Stride : public TimeIntegrator, public StrideBasics {
private:

 // ----- Storage of debug information -----
  BooLean do_debug_;
  int db_max_steps_;
  int db_steps_;
  VecSimple(int)  db_status_;
  Vec(real)       db_times_;
  Vec(real)       db_stepsize_;
  VecSimple(int)  db_order_;

  VecSimple(int)  db_newton_it_;
  VecSimple(int)  db_jacs_;
  VecSimple(real) db_roc_;

  VecSimple(real) db_error_;

protected:
  BooLean startup_;

  // ----- Storage of Error Estimates -----

  BooLean error_estimate_is_valid_;
  real error_estimate_;

  Vec(real) error_;
  Vec(real) error_higher_;
  Vec(real) error_lower_;

  // ----- Jacobian evaluation info -----

  BooLean evaluate_jacobian_;
  int jac_eval_;  // the step when the Jacobian was last evaluated
  real Hjac_;     // and the value of H when it happened
```

```
// ----- Workspace -----

BooLean dy0_is_valid_;
Vec(real) dy0_;

Vec(real) scratch_;
Vec(real) y_temp_;

VecSimplest(Vec(real)) F_;
VecSimplest(Vec(real)) Z_;
VecSimplest(Vec(real)) Y_;

// ----- Stepsize and order selection -----

Handle(StrideStrategy) strategy_;

// ----- Newton iteration parameters -----

int max_number_of_iterations_;
real kappa_;        // safety factor for convergence test
real minimum_roc_;  // re-evaluate jac next time if
                    // rate of convergence is lower
int extra_iterations_; // do a few extra iteration after conv.

// ----- Statistics for last step -----

int iteration_status_;
int number_of_iterations_;
int age_of_jacobian_;
real last_norm_;
real last_eta_;
real last_roc_;
real last_error_;

real  max_conv_H_;

// ----- Protected Function Members -----

void reset(real init_h);
void initStageValues(VecSimplest(Vec(real))& HF,
                     VecSimplest(Vec(real))& Z);

virtual void interpolate(real t, Vec(real)& y);

BooLean stepWithErrorCheck();
BooLean step(int& n_it);
void initDerivative(Vec(real) dy);

BooLean validErrorEstimate() { return error_estimate_is_valid_; }

virtual void calcErrorEstimates();

// ---- Purely virtual functions  - must be defined in derived classes -----

virtual void    iteration      (VecSimplest(Vec(real))& Z,
                                 real& y_norm, real& diff_norm) = 0;
virtual void    endIteration   (VecSimplest(Vec(real))& Z) = 0;
```

```
  virtual BooLean                 updateJacobian()=0;
  virtual VecSimplest(Vec(real))& getTransStageDer() = 0;

public:
  Stride(int n, real init_h);
  ~Stride();

  inline void reStart(Vec(real)& dy, Vec(real)& y, real t, real init_h);
  void        reStart(Stride& solver);

  // ----- Provide order and stepsize of the current and the last
           accepted step -----

  int  getS     () { return strategy_.getRef().getS(); }
  int  getLastS () { return strategy_.getRef().getLastS(); }
  real getH     () { return strategy_.getRef().getH(); }
  real getLastH () { return strategy_.getRef().getLastH(); }

  StrideStrategy& getStrategy        ();
  void            setStrategy        (StrideStrategy& strat)
                                     { strategy_.rebind(strat); }

  Vec(real)&              getDerivative()  { return dy0_; }
  VecSimplest(Vec(real))& getStageValues() { return Z_; }

  // ----- Debugging of the step-size mechanism -----

  void setDebug(int max_steps);
  void doDebug();
  void printDebug(Os& ofile, BooLean acc, BooLean divg, BooLean fail);

  // ----- Functions for advancing the solution -----

  virtual BooLean advance   ();
  virtual void    advance   (real t, Vec(real)& y)
                                       { TimeIntegrator::advance(t,y); }
  virtual BooLean forceStep (int order, real stepsize);

  // ----- Newton-Raphson iteration -----

  real getRateOfConvergence() { return last_roc_; }
  int  getNumberOfIterations() { return number_of_iterations_; }
  real getMaxConvH() { return  max_conv_H_; }

  void setExtraIterations (int n) { extra_iterations_ = n; }
  void setEvalJacobian    ()      { evaluate_jacobian_ = dpTRUE; }
  int  getAgeOfJacobian   ()      { return getAcceptedStep() - jac_eval_; }

  // ----- Local error estimation -----

  real getErrorEstimateLowerOrder();
  real getErrorEstimateHigherOrder();

  virtual void getErrorEstimate(Vec(real)& error);
  virtual real getErrorEstimate();
```

```
};
```

KEYWORDS solver, initial value problem, SIRK

DESCRIPTION

The class implements a general SIRK solver, that should be able to solve ODEs on different forms, f.ex. IDE, DAE and the usual explicit form. The functions that depend on the representation of the equation are abstract, and should be implemented in derived classes.

CONSTRUCTORS AND INITIALIZATION

Stride needs the number of unknown variables and an initial stepsize for initialization.

MEMBER FUNCTIONS

The `getS` and `getLastS` methods returns the current order and the order of the last accepted step. `getH` and `getLastH` returns the value of $H = \lambda h$, where h is the stepsize.

`reStart` - makes it possible to start integration from different initial values.

`getStrategy` - returns a reference to the current `StrideStrategy`.

`setStrategy` - swaps to the stepsize/order strategy given as argument.

`getDerivative` - returns a reference to a vector containing the derivative at the start of the current step.

`setEvalJacobian` - will force the jacobian to be evaluated at the next step.

`getAgeOfJacobian` - returns the number of accepted steps since the jacobian was last evaluated.

`forceStep` - performs one step with a given stepsize and order, and prepares for a new step (even if the error is greater than the tolerance). Returns `dpFALSE` in case of divergence (or too slow convergence).

`getErrorEstimate` - calculates an estimate of the error in the last convergent step. There is two overloaded instances, where one of them calculates the error vector, while the other returns the norm if the error.

`getErrorEstimateLowerOrder` - calculates an estimate of the error if a method of one order less than the current method was used.

`getErrorEstimateHigherOrder` - calculates an estimate of the error if a method of one order higher than the current was used.

`reset` - will set the Stride class into initial state. Requires an initial stepsize.

`initStageValues` - estimates starting values for the Newton-Raphson iteration based on the previous accepted step.

`interpolate` - interpolates a solution value.

`advance` - automaticly finds a new stepsize and tries to advance the solution. Returns `dpTRUE` if success. In case of failure other stepsizes will be tried, but if nothing helps the function will return `dpFALSE`.

`step` - will try to find the Runge Kutta solution for the next step in a given number of iterations, using the stepsize and order provided by the `StrideStrategy` object. Returns `dpFALSE` at failure.

`stepWithErrorCheck` - calls `step` and compares the local error estimate with the tolerance. Return value `dpFALSE` means that the tolerance was exceeded or the Newton-Raphson iteration has diverged.

`setDebug` - starts recording of data from each step. See the `printDebug` function for details. The data is stored in internal lists, and the maximum number of steps should be provided as argument.

`printDebug` - prints data from accepted/diverged/failed steps to file. The format is

```
step#  NR-status  t  h  S  NR-iterations  Jac-eval  roc  error
```

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.2.3 Stride for IDEs

NAME StrideIDE

INCLUDE `include "StrideIDE.h"`

SYNTAX

```
class StrideIDE : public Stride {

private:
  IDESolverUDC& eqdef_;
  BooLean numerical_jacobian_;

  VecSimplest(Vec(real)) D_;
  VecSimplest(Vec(real)) R_;
  VecSimplest(Vec(real)) Dtrans_;
```

```
  VecSimplest(Vec(real)) Rtrans_;
  VecSimplest(Vec(real)) bigscratch_;
  Vec(real) rhs_;

  BooLean HFtrans_is_valid_;
  VecSimplest(Vec(real)) HFtrans_;

  void calcHFtrans(VecSimplest(Vec(real))& Z);

  BooLean sharp_error_is_valid_;
  BooLean sharp_error_damped_is_valid_;
  BooLean fast_error_damped_is_valid_;

  Vec(real) sharp_error_;
  Vec(real) sharp_error_damped_;
  Vec(real) fast_error_damped_;

protected:
  // ----- parameters -----

  BooLean norm_all_components_;

  BooLean scale_index2_;
  BooLean scale_index3_;

  // ----- virtual functions -----
  virtual void iteration(VecSimplest(Vec(real))& Z,
                         real& y_norm, real& diff_norm);
  virtual void    endIteration(VecSimplest(Vec(real))& Z);
  virtual BooLean updateJacobian();
  virtual VecSimplest(Vec(real))& getTransStageDer();

  void calcSharpError();
  void calcDampedError(Vec(real)& err);
  virtual void calcErrorEstimates();
public:
  StrideIDE(IDESolverUDC& eq, real init_h);

  virtual real toleranceNorm(Vec(real)& y);
  void setAllComponentsNorm(BooLean val) { norm_all_components_ = val; }

  virtual void getErrorEstimate(Vec(real)& error);
  virtual real getErrorEstimate();
  real getErrorEstimate(int opt);

  void setScaleHigherIndex(BooLean index2, BooLean index3)
    { scale_index2_ = index2; scale_index3_ = index3; }

  IDESolverUDC& getProblem() { return eqdef_; }
  setNumericalJacobian(BooLean flag) { numerical_jacobian_ = flag; }
};
```

KEYWORDS IDE, solver

DESCRIPTION

Use this class to solve implicit differential systems on the form $F(\frac{dy}{dt}, y, t) = 0$. The system is solved by *Singly Implicit Runge Kutta methods* using varable stepsize and order in the range $2, \dots, 10$.

CONSTRUCTORS AND INITIALIZATION

The only constructor of this class takes an IDESolverUDC and an initial stepsize as parameter.

MEMBER FUNCTIONS

`getProblem` - returns a reference to the differential system, represented by an IDESolverUDC object.

`setNumericalJacobian` - forces the jacobian to be evaluated numerically if the argument is `dpTRUE`, otherwise analytical jacobian will be used if it exists.

`iteration` - performs a modified Newton-Raphson iteration and calculate new estimations of the stage values.

`updateJacobian` - evaluates and factorizes the jacobian (numerically or analytically). Returns `dpFALSE` in case of failure.

`getTransStageDer` - returns a reference to the transformed stage derivatives multiplied by H, corresponding to the last iteration.

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.3 Stepsize and Order Strategies

B.3.1 Base Class

NAME StrideStrategy

INCLUDE `include "Strategy.h"`

SYNTAX

```
class StrideStrategy : public virtual HandleId {
protected:
  Stride& solver_;
  // real max_stepsize_;
  // stepsize
  real H_old_;
  real H_last_;
  real H_;
  // and order
  int S_old_;
  int S_last_;
  int S_;

  BooLean step_failed_;
  BooLean step_diverged_;
  BooLean step_LUfailed_;

  real newH_divg_;

  int nfailed_;

  void rotate();
public:
  StrideStrategy(Stride& solver) : solver_(solver) { init(); }
  Stride& getSolver() { return solver_; }

  real getH() { return H_; }
  int getS() { return S_; }
  real getLastH() { return H_last_; }
  int getLastS() { return S_last_; }

  int getFailCount() { return nfailed_; }

  void setInitH(real H);
  void setOrder(int S);

  virtual void init();
  virtual void init(StrideStrategy &strategy);

  // one of the newstep methods _must_ be called at the start of
  // each step
  virtual void newstep() = 0;

  // use this to force a special order/stepsize for next step
```

```
  void newstep(int order, real stepsize);

  void reportFailure();
  void reportDivergence(real newH = 0.0);
  void reportLUFailure(real newH = 0.0);
};
```

KEYWORDS strategy, order, stepsize

DESCRIPTION

StrideStrategy is a base class for stepsize and order selection in Stride.

CONSTRUCTORS AND INITIALIZATION

The class has to consult Stride for error estimates et.c., and it is necessary to provide a reference to a Stride object at initialization.

MEMBER FUNCTIONS

The `getS` and `getLastS` functions returns the order of the current and last accepted step respectivly. `getH` and `getLastH` returns the value of $H = \lambda h$.

`newstep` - should be called at the start of each step. There is two overloaded instances, where one of them lets the user manually specify a certain stepsize and order for the next step.

`reportFailure` - informs the class that the current step caused the local error estimate to exceed the tolerance. The class will assume that the step is rejected.

`reportDivergence` - should be called when the Newton-Raphson iteration diverges (or convergence is too slow). The user can propose a new stepsize, or leave the choice to `StrideStrategy`.

`reportLUFailure` - reports a failure in the evaluation or factorization of the jacobian.

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.3.2 A Variable Stepsize and Order Strategy

NAME StrategyBasic

INCLUDE `include "Strategy.h"`

SYNTAX

```
class StrategyBasic : public StrideStrategy {
protected:
  StrategyState state_;

  int nsstep_;
  int nopt_;
  real errmo_;
  real errso_;

  // ----- Parameters -----
  int max_order_;
  int min_order_;

  Vec(real) cost_;
  VecSimple(real) init_factors_;
  VecSimple(real) magic_factors_;
  VecSimple(int) counts_;

  virtual void initial();
  virtual void pessimistic();
  virtual void normal();
  virtual void optimistic();
  virtual void prefer(real& Hnew, real H, real Hmax, int s, real cost,
                      real err, real xiout, real& score, BooLean& change);

public:
  StrategyBasic(Stride& solver) : StrideStrategy(solver) { init(); }

  virtual void init();
  virtual void init(StrideStrategy &strategy);

  StrategyState getState() { return state_; }
  real getCost(int S) { return cost_(S); }
  void setState(StrategyState state) { state_ = state; nsstep_ = 0;
  nopt_ = 0;}

  virtual void newstep();

};
```

KEYWORDS strategy, order, stepsize

DESCRIPTION

StrategyBasic is a simplified version of the order and stepsize control mechanism from the FORTRAN version of Stride. The class operates in three different states; *optimistic*, *normal* and *pessimistic*.

CONSTRUCTORS AND INITIALIZATION

The class has to consult Stride for error estimates et.c., and it is necessary to provide a reference to a Stride object at initialization.

MEMBER FUNCTIONS

`newstep` - calculates new values for the stepsize and order, based on error estimates from the last step or reported failures.

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.3.3 The Gustafsson Stepsize Control

NAME Gustafsson

INCLUDE `include "Gustafsson.h"`

SYNTAX

```
//---------------------------------------------------------------------------
class Gustafsson : public StrideStrategy
//---------------------------------------------------------------------------
{

  real err0_;
  real err1_;
  real err_H_;

  BooLean err1_is_valid_;

  BooLean startup_;

public:
  Gustafsson(Stride& solver) : StrideStrategy(solver) { init(); }

  virtual void init();
  virtual void newstep();
};
```

KEYWORDS stepsize, strategy

DESCRIPTION

The class Gustafsson is a variable stepsize, constant order strategy for the Stride integrator. The implementation is based on the idea of Gustafsson, described in [8] section IV.2.

CONSTRUCTORS AND INITIALIZATION

The constructor needs a reference to the Stride integrator.

MEMBER FUNCTIONS

`init` - initializes the member data, and thereby restarts the strategy. An initial stepsize should be given to the parent class `StrideStrategy`.

`newstep` - must be called by the integrator at the start of each step. This function calculates the size of the next step.

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.4 Abstract Problem Classes

B.4.1 Implicit Ordinary Differential Equations (IDE)

NAME IDESolverUDC

INCLUDE `include "StrideUDC.h"`

SYNTAX

```
//-----------------------------------------------------------------------------
class IDESolverUDC : public virtual HandleId {
//-----------------------------------------------------------------------------
protected:
  int size_;

  VecSimple(int) index1_;
  VecSimple(int) index2_;
  VecSimple(int) index3_;

  //  FactStrategy fact_strategy_;
  Handle(Mat(real)) jac_;
  Handle(Mat(real)) mass_;
  Handle(Mat(real)) iteration_matrix_;

  BooLean analytic_jac_;
  BooLean analytic_mass_;

  int jac_bw_;
  int mass_bw_;
  int itmat_bw_;

  Vec(real) scratch_;
  VecSimple(int) perm_;

public:
  IDESolverUDC(int n, BooLean analytic_jac = dpTRUE, int jac_bw = 0,
                      BooLean analytic_mass = dpTRUE, int mass_bw_ = 0,
               int ind1 = 0, int ind2 = 0, int ind3 = 0);

  void redim(int n, int jac_bw, int mass_bw);
  Mat(real)& getJacobian() { return jac_.getRef(); }
  Mat(real)& getMass() { return mass_.getRef(); }
  Mat(real)& getMatrixIDE() { return iteration_matrix_.getRef(); }
  virtual int size() { return size_; }

  void addMatrices(real alpha);

  BooLean analyticJacobian() { return analytic_jac_; }
  BooLean analyticMass() { return analytic_mass_; }

  void setIndex(int ind1, int ind2, int ind3);
//  void setIndex(const VecSimple(int)& ind) { index_ = ind; }
```

```
  VecSimple(int)& getIndex1() { return index1_; }
  VecSimple(int)& getIndex2() { return index2_; }
  VecSimple(int)& getIndex3() { return index3_; }


// ------- User defined functions ------

  virtual void initValues(Vec(real)& dy, Vec(real)& y, real& t)=0;
  virtual void residual(Vec(real)& F, Vec(real)& dy, Vec(real)& y, real t)=0;

// ------- Optional user defined functions -----

  virtual void evalJacobian(Vec(real)& dy, Vec(real)& y, real t,
                            Mat(real)& jac);
  virtual void evalMass(Vec(real)& dy, Vec(real)& y, real t,
                        Mat(real)& mass);

  virtual void massProd(const Vec(real)& D, Vec(real)& R);
  virtual void evalMatrixIDE(Vec(real)& dy, Vec(real)& y,
                                  real t, real alpha);

  virtual BooLean factorize();
  virtual void solve(Vec(real)& b, Vec(real)& x);
};

#define  ClassType IDESolverUDC
#include <Handle.h>
#undef   ClassType
```

KEYWORDS IDE, UDC

DESCRIPTION

This class contains an abstract interface for ODE*s on* the form $F(\frac{dy}{dt}, y, t) = 0$. The actual system of equations must be provided by the user in a derived class (UDC = User Dependent Code).

The user has to supply `evalRHS` for the evaluation of F, and initial values for the problem in `initValues`. In addition it is usually smart to overload the functions `evalMass` and `evalJacobian` (otherwise numerical approximations will be calculated in Stride).

The current implementation supports full and banded matrices, but it is possible for the user to make optimized linear algebra by providing new instances of `massProd`, `evalMatrixIDE`, `factorize` and `solve`.

MEMBER FUNCTIONS

`getJacobian` - returns a reference to a matrix used to store the derivative $J = \frac{\partial F}{\partial y}$.

`getMass` - returns a reference to the matrix $M = \frac{\partial F}{\partial x}$, where $F = F(x, y, t)$.

`getMatrixIDE` - returns a reference to the matrix containing $J + \alpha K$, where the scalar α is provided by STRIDE.

`massProd` - calculates the product Mx.

`evalMatrixIDE` - evaluates $J + \alpha K$ and stores the matrix internally.

`factorize` - performes a partially pivotet LU factorization of the matrix.

`solve` - uses forward and backward substitution to solve the linear system.

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.4.2 Differential Algebraic Equations (DAE)

NAME DAESolverUDC

INCLUDE `include "StrideUDC.h"`

SYNTAX

```
//---------------------------------------------------------------------------
class DAESolverUDC : public IDESolverUDC
//---------------------------------------------------------------------------
{
protected:
  BooLean mass_is_evaluated_;

public:
  DAESolverUDC(int n, BooLean analytic_jac = dpTRUE,
               int jac_bw = 0, int mass_bw = 0,
               int ind1 = 0, int ind2 = 0, int ind3 = 0);

  virtual void evalRHS(Vec(real)& f, Vec(real)& y, real t) = 0;
  virtual void evalMass(Mat(real)& jac) = 0;
//  virtual void initValues(Vec(real)& dy, Vec(real)& y, real& t) = 0;

  virtual void evalJacobian(Vec(real)& y, real t, Mat(real)& jac);

  virtual void evalMatrixDAE(Vec(real)& dy, Vec(real)& y,
                                real t, real alpha);

// -------- Conversion to IDESolverUDC --------
  virtual void residual(Vec(real)& F, Vec(real)& dy, Vec(real)& y, real t);
//   virtual void evalMatrixIDE(Vec(real)& dy, Vec(real)& y,
//                                real t, real alpha);
  virtual void evalMass(Vec(real)& dy, Vec(real)& y, real t,
```

```
                      Mat(real)& mass);
  virtual void evalJacobian(Vec(real)& dy, Vec(real)& y, real t,
                          Mat(real)& jac);

};


#define  ClassType DAESolverUDC
#include <Handle.h>
#undef   ClassType
```

KEYWORDS DAE, UDC

DESCRIPTION

This class contains an abstract interface for differential equations on the DAE form $M\frac{dy}{dt} = f(y,t)$, where M is a constant matrix, possibly singular. The actual system must be provided by the user in a derived class (UDC = User Dependent Code). `evalRHS` should evaluate the function f, `evalMass` should evaluate M and `initValues` should provide initial values for the system.

To prevent numerical evaluation of the Jacobian, `evalJacobian` should be implemented if possible.

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.4.3 Explicit Ordinary Differential Equations (ODE)

NAME ODESolverUDC

INCLUDE `include "StrideUDC.h"`

SYNTAX

```
//-----------------------------------------------------------------------------
class ODESolverUDC : public DAESolverUDC
//-----------------------------------------------------------------------------
{
public:
  ODESolverUDC(int n, BooLean analytic_jac = dpTRUE, int bandwidth=0)
    : DAESolverUDC(n, analytic_jac, bandwidth, 1) {}

//  virtual void evalRHS(Vec(real)& f, Vec(real)& y, real t) = 0;
  virtual void initValues(Vec(real)& y, real& t)=0;
//  virtual void evalJacobian(Vec(real)& y, real t, Mat(real)& jac);
```

```
  // The following functions should not be overloaded - they provide
  // automatic conversion to DAE's and IDE's

  virtual void residual(Vec(real)& F, Vec(real)& dy, Vec(real)& y, real t);
  virtual void initValues(Vec(real)& dy, Vec(real)& y, real& t);
  virtual void evalMass(Mat(real)& mass);
  virtual void massProd(const Vec(real)& D, Vec(real)& R);

//   virtual void evalMatrixIDE(Vec(real)& dy, Vec(real)& y,
//                               real t, real alpha);
  virtual void evalMatrixDAE(Vec(real)& dy, Vec(real)& y,
                             real t, real alpha);
  virtual void evalMatrixODE(Vec(real)& dy, Vec(real)& y,
                             real t, real alpha);
};

#define  ClassType ODESolverUDC
#include <Handle.h>
#undef   ClassType
```

KEYWORDS ODE, UDC

DESCRIPTION

This class contains an abstract interface for differential systems on the form $\frac{dy}{dt} = f(y,t)$. The actual equation must be provided by the user in a derived class (UDC = User Dependent Code) in the functions `evalRHS`, `initValues` and `evalJacobian` (optional).

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.5 Problem Conversion

B.5.1 IDEs from the CWI Test Set

NAME IDE_CWI

INCLUDE `include "IDE_CWI.h"`

SYNTAX

```
//-----------------------------------------------------------------------------
class IDE_CWI : public IDESolverUDC
//-----------------------------------------------------------------------------
{
protected:
  int mljac_, mujac_;

  int analytic_jac_;

  real tbegin_, tend_;
  double* scratch_;

public:
  IDE_CWI();
  ~IDE_CWI();

  virtual void initValues(Vec(real)& dy, Vec(real)& y, real& t0);
  virtual void residual(Vec(real)& F, Vec(real)& dy, Vec(real)& y, real t);

  // jacobian evaluation not implemented
  //
  //  virtual BooLean evalJacobian(Vec(real)& dy, Vec(real)& y, real t,
  //                               Mat(real)& jac);
  //  virtual BooLean evalMass(Vec(real)& dy, Vec(real)& y, real t,
  //                           Mat(real)& mass);

  void exact(Vec(real)& y, real& t);
};
```

KEYWORDS IDE, CWI

DESCRIPTION

The class provides a C++ interface for problems on IDE form in the CWI testset (FORTRAN 77). Since all IDEs in the testset use the same function names, no more than one problem can be compiled at the same time in an application.

CONSTRUCTORS AND INITIALIZATION

No parameters needed.

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.5.2 DAEs from the CWI Test Set

NAME DAE_CWI

INCLUDE `include "DAE_CWI.h"`

SYNTAX

```
//-----------------------------------------------------------------------------
class DAE_CWI : public DAESolverUDC
//-----------------------------------------------------------------------------
{
protected:
  int mljac_, mujac_;
  int mlmas_, mumas_;
//  int bw_;
  int analytic_jac_;

  real tbegin_, tend_;
  double* scratch_;

public:
  DAE_CWI();
  ~DAE_CWI();

  virtual void initValues(Vec(real)& dy, Vec(real)& y, real& t0);
  virtual void evalRHS(Vec(real)& f, Vec(real)& y, real t);
  virtual void evalMass(Mat(real)& jac);
  virtual void evalJacobian(Vec(real)& y, real t, Mat(real)& jac);

  void exact(Vec(real)& y, real& t);
};
```

DESCRIPTION

The class provides a C++ interface for problems on DAE form in the CWI testset.

SEE ALSO IDE_CWI

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

B.5.3 ODEs from the CWI Test Set

NAME ODE_CWI

INCLUDE `include "ODE_CWI.h"`

SYNTAX

```
//-----------------------------------------------------------------------------
class ODE_CWI : public ODESolverUDC
//-----------------------------------------------------------------------------
{
protected:
  int mljac_, mujac_;
  int analytic_jac_;

  real tbegin_, tend_;
  double *scratch_;
public:
  ODE_CWI();
  ~ODE_CWI();

  virtual void initValues(Vec(real)& y, real& t0);
  virtual void evalRHS(Vec(real)& f, Vec(real)& y, real t);
  virtual void evalJacobian(Vec(real)& y, real t, Mat(real)& jac);

  void exact(Vec(real)& y, real& t);
};
```

DESCRIPTION

The class provides a C++ interface for problems on ODE form in the CWI testset.

SEE ALSO DAE_CWI, IDE_CWI

DEVELOPED BY

Erik Hamran Nilsen, the University of Auckland, New Zealand

Bibliography

[1] K. E. Brenan, S. L. Campbell, L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* North-Holland, Amsterdam 1989.

[2] K. Burrage, *A special family of Runge-Kutta methods for solving stiff differential equations.* BIT 18 (1978), 22-41.

[3] K. Burrage, J. C. Butcher, F. H. Chipman *An implementation of singly-implicit Runge-Kutta methods*, BIT, 20 (1980), 326-340.

[4] J. D. Lambert, *Numerical Methods for Ordinary Differential Systems; The Initial Value Problem.* Wiley, Chichester 1991.

[5] J. C. Butcher, *The Numerical Analysis of Ordinary Differential Equations.* Wiley, New York 1987.

[6] J. C. Butcher, *A Transformed Implicit Runge-Kutta Method.* Journal of the Association for Computing Machinery, Vol. 26, No. 4. (1979), pp. 731-738.

[7] E. Hairer, S. P. Nørsett, G. Wanner, *Solving Ordinary Differential Equations; Nonstiff Problems.* Springer-Verlag, Berlin 1987.

[8] E. Hairer, G. Wanner *Solving Ordinary Differential Equations II; Stiff and differential-algebraic problems.* Springer-Verlag, Berlin 1996.

[9] E. Hairer, C. Lubich, M. Roche, *The Numerical Solution of Differential-Algebraic Systems by Runge-Kutta Methods.* Lecture Notes in Math. 1409, Springer-Verlag (1989).

[10] R. März *Numerical methods for differential algebraic equations.* Acta Numerica (1991), pp. 141-198.

[11] S. P. Nørsett, *Semi Explicit Runge-Kutta Methods.* Mathematics and Computation, Mathematics Dept., University of Trondheim, Norway, Report No. 6/74.

[12] R. Alexander, *Diagonally implicit Runge-Kutta methods for stiff ODEs.* SIAM J. Numer. Anal., 14 (1977), pp. 1006-1021.

[13] `http://www.cwi.nl/cwi/projects/IVPtestset.shtml` *Test set for IVP solvers.*

[14] `http://www.oslo.sintef.no/avd/33/3340/diffpack/index.html` *The Diffpack Home Page*

[15] H. P. Langtangen, *Basic Concepts in Diffpack*. The `Diffpack` Report Series, SINTEF Applied Mathematics, 1994.

[16] H. P. Langtangen, T. V. Stensby, *Tools for Developing Software Packages Under Unix*. The `Diffpack` Report Series, SINTEF Applied Mathematics, 1994.