Implementation of Singly Diagonally Implicit Runge–Kutta Methods with Constant Step Sizes

Josefine Ståhl

Bachelor's thesis
2015:K12

Lund University

Faculty of Science
Centre for Mathematical Sciences
Numerical Analysis

Centrum Scientiarum Mathematicarum

Abstract

Runge–Kutta methods can be used for solving ordinary differential equations of the form $y' = f(t, y)$ with initial condition $y(t_0) = y_0$ and where $f : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$. The idea is to find a method that is efficient to implement. But it is also important for the method to be of high order and be stable. Diagonally Implicit RK-methods reduces an $sm \times sm$ matrix to s systems of $m \times m$ linear equations. Singly Diagonally Implicit RK-methods have only a single eigenvalue, which results in a reduction to only one LU-decomposition per time step. Combining the two methods, we get Singly Diagonally Implicit RK-methods.

Keywords: *Implicit Runge–Kutta methods*, *SDIRK*, *Implementation*

Acknowledgement

I am incredibly thankful for all the help and incouragement I have recieved from my supervisor Claus Führer at the University of Lund. I would also like to express my sincere gratitude to Gustaf Söderlind, who shared his expertise when I needed it and inspires me to continue working hard.

Table of Contents

Chapter 1

Introduction - Runge Kutta Methods

The first-order ordinary differential equation(ODE) with a given initial value, of the form

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0, \quad f : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m \tag{1.1}$$

can be solved with a one-step method. An ODE of this form is also called an initial value problem(IVP). The general form of a one-step method is

$$u_{n+1} = u_n + h\phi(f, h, t_n, u_n, u_{n+1}) \tag{1.2}$$

where h is the stepsize. If the function $\phi(f, h, t_n, u_n)$ does not depend on the variable u_{n+1} the method is explicit, and otherwise implicit.

Higher order differential equations can, by introducing extra variables, be written as a system of first-order ODEs. Without loss of generality we therefore only consider first-order ODEs.

Runge–Kutta methods are one-step methods, and have the form

$$\begin{aligned} y_{n+1} &= y_n + h\sum_{i=1}^{s} b_i Y_i \\ Y_i &= y_n + h\sum_{j=1}^{s} a_{ij} Y_j' \\ Y_i' &= f(t_n + c_i h, Y_i) \end{aligned} \tag{1.3}$$

where Y_i are the stage values, Y_i' the stage derivatives and s the number of stages. The method and its coefficents are chosen to satisfy the conditions that guarantee a certain order of accuracy. Furthermore the method sought to be stable and time efficient. The Butcher Tableau, which contains the coefficients b_i, c_i, a_{ij}, changes depending on the conditions. If A is strictly lower triangular, i.e. if $a_{ij} = 0$ for $i <= j$, then the method is explicit and implicit otherwise.

$$\begin{array}{c|c} c & A \\ \hline & b^{\mathrm{T}} \end{array} = \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

Table 1.1: Butcher Tableau

1.1 Properties

When solving an ordinary differential equation, and chosing an appropriate method for the problem, it is neccessary to consider a couple of properties. In applied mathematics, for instance for numerical methods, the notion of a well-posed problem is desired [1, p.9].

Definition 1.1.1 *A problem is well-posed if there exists a unique solution and if it depends continously on the parameters and the initial conditions.*

For the problem to depend continously on the parameters and the initial conditions, we assure that the approximated solution does not produce large errors if small changes are made in the initial conditions. This also means that local truncation errors at each time step do not affect the final solution too much.

Consider the one-step method (1.2). Let $u(t)$ denote the exact solution at time $t = nh$ and u_n the approximated solution at that time. Then the *global truncation error* at step n is defined as

$$\varepsilon(t) = u(t) - u_n$$

If we consider the last approximated solution to be the exact one, then the difference to the approximated solution at time t is *the local truncation error*. Furthermore, the method is said to be *consistent* if

$$\lim_{h \to 0} \frac{\varepsilon(t)}{h} = 0$$

and the *order* of the method is p if

$$\varepsilon(t) = O(h^{p+1}) \quad \text{as } h \to 0$$

When working with Runge–Kutta methods the consistency condition is definied as follows

Definition 1.1.2 *Let the vector b be the weights of the Butcher Tableau, and s the number of stages. Then the consistency condition for Runge–Kutta methods is defined as*

$$\sum_{j=1}^{s} b_j = 1$$

To make sure there exists a unique solution, we need to know about the Lipschitz condition. The defintion and theorem are found in [2, p. 22-23].

Definition 1.1.3 *A function $f : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$ is said to satisfy a Lipschitz condition, with respect to the second argument, if there exists a number L, known as the Lipschitz constant, s.t. $\forall u, v \in \mathbb{R}$*

$$||f(t, u) - f(t, v)|| \leq L||u - v||$$

Theorem 1 *If $f : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$ satisfies the Lipschitz condition, with respect to the second argument. Then there exists a unique solution to the initial value problem (1.1) which depends continuously on the initial value.*

Definition 1.1.4 *Let $t = nh$. The solution converges if $\lim_{h \to 0} y_n = y(t)$ where y_n is the approximated solution at t and $y(t)$ the exact one.*

Given that the IVP (1.1) is consistent and the solution is unique and convergent (i.e. well-posed), we consider a real homogeneous linear system of differential equations with constant coefficients.

$$y' = Ay, \quad A \in \mathbb{R}^{n\times n} \tag{1.4}$$

Assume A is diagonalizable. Then there exists a similarity transformation

$$TAT^{-1} = \text{diag}(\lambda_1, \lambda_2, \cdots, \lambda_n) = \Lambda,$$

where the columns of T are linearly independent eigenvectors of A, with corresponding eigenvalues $\lambda_i \in \mathbb{C}$, $i = 1, \ldots, n$. After a change of variables $z(t) = T^{-1}y(t)$, the system (1.4) can be written as

$$z' = \Lambda z = \begin{cases} z_1' = \lambda_1 z \\ z_2' = \lambda_2 z \\ \vdots \\ z_n' = \lambda_n z \end{cases}$$

Consider one of these components, the so called *linear test equation*

$$z'(t) = \lambda z(t), \quad z(0) = 1, \quad \lambda \in \mathbb{C} \tag{1.5}$$

to check stability of the solution. Let $z(t)$ and $\hat{z}(t)$ be two solutions to the linear test equation (1.5). If $z(t)$ corresponds to the solution with the correct initial data, and $\hat{z}(t)$ has a small error in the initial condition. Then the stability can be studied by the behavior of the difference of both.

$$|z(t) - \hat{z}(t)| = |(z(0) - \hat{z}(0))e^{\lambda t}| = |z(0) - \hat{z}(0)|e^{Re(\lambda)t}$$

Thus, the solution $z(t) = e^{\lambda t}$, for $t \geq 0$, is said to be *stable* if $Re(\lambda) \leq 0$, *asymptotically stable* if $Re(\lambda) < 0$ and *unstable* if $Re(\lambda) > 0$ [1, p. 23-28]. The stability conditions for Runge–Kutta methods can be obtained by first considering the stability function

Definition 1.1.5 *The RK-method* (1.3) *applied to the linear test equation* (1.5) *yields* $y_{n+1} = R(z)y_n$, *where*

$$R(z) = 1 + zb^T(I - zA)^{-1}\mathbf{1} \tag{1.6}$$

is called the stability function, $z = h\lambda$, $\mathbf{1} = (1, 1, ..., 1) \in \mathbb{R}^s$ *and* I *the identity matrix.*

The stability function of implicit Runge–Kutta methods is a rational function

$$R(z) = \frac{P(z)}{Q(z)}$$

with degree less or equal to s. If the stability function (1.6) is applied to an explicit Runge-Kutta method we see that $R(z)$ is a polynomial of degree s, with $R(z) \to \infty$ as $z \to \infty$. Hence, the explicit RK-methods cannot be A-stable(see definition below).

Definition 1.1.6 *The stability region is defined as* $S = \{z \in \mathbb{R} : |R(z)| \leq 1\}$

Definition 1.1.7 *A Runge–Kutta method is said to be A-stable if its stability region contains* $\mathbb{C}^- = \{z \in \mathbb{R} : Re(z) \leq 0\}$*, i.e. the left half-plane.*

Applying the maximum principle on $\mathbb{C}^-$ [3, p.43], we get that a Runge–Kutta method is A-stable iff

$$|R(iy)| \leq 1, \quad y \in \mathbb{R} \tag{1.7a}$$

$$R(z) \text{ is analytic for } Re(z) < 0 \tag{1.7b}$$

If the solution of a stable ODE (1.1) is A-stable, then the solution remains bounded. L-stability on the other hand is more strict and the solution must also be damped at infinity.

Definition 1.1.8 *A Runge–Kutta method is L-stable if the method satisfies the following conditions:*

$$|R(z)| \leq 1, \quad \forall z \in \mathbb{C}^- \text{ (i.e. A-stable)}$$

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

where $R(z)$ *is the stability function.*

There are always errors when solving a system of differential equations with a numerical method. The question is how to balance the error and the execution time. If the error is proportional to h^n, where h is the stepsize, then we say the solution of the numerical method is n-th order accurate. This is a meassure of the rate of convergence to the exact solution. It expresses how fast the solution approaches its limit. This means that the higher order method, the larger stepsize can be used to reach a given tolerance.

Chapter 2

Efficiency Problems

Consider a general Runge Kutta method (1.3). To implement the method, Newton's method is used to solve the system of algebraic equations. Newton's method solves the equation $G(x) = 0$ by iterating over

$$x_{n+1} = x_n - (G'(x_n))^{-1} G(x_n) \Leftrightarrow G'(x_n)(x_{n+1} - x_n) = -G(x_n) \tag{2.1}$$

The solution is found when the difference $x_{n+1} - x_n$ is less than a given tolerance. If the Jacobian matrix $G'(x_n)$ is calculated at each time step, the convergence of the method is quadratic.

Consider the formula for stage values in the Runge–Kutta method (1.3).

$$Y_i = y_n + h \sum_{j=1}^{s} a_{ij} f(t_n + c_i h, Y_i)$$

The equation solved with Newton's method is

$$G(Y_i) = Y_i - y_n - h \sum_{j=1}^{s} a_{ij} f(t_n + c_i h, Y_i) = 0 \tag{2.2a}$$

$$G'(Y_i) = \mathrm{I} - h \sum_{j=1}^{s} a_{ij} f'(t_n + c_i h, Y_i) \tag{2.2b}$$

Thus the system to solve in the n:th Newton iteration, for Runge–Kutta methods, takes the following form

$$\begin{bmatrix} I - ha_{11}J_1 & -ha_{12}J_2 & \dots & -ha_{1s}J_s \\ -ha_{21}J_1 & I - ha_{22}J_2 & \dots & -ha_{2s}J_s \\ \vdots & \vdots & \ddots & \vdots \\ -ha_{s1}J_1 & -ha_{s2}J_2 & \dots & I - ha_{ss}J_s \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_s \end{bmatrix} = - \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_s \end{bmatrix} \tag{2.3}$$

where $d_i = Y_i^{n+1} - Y_i^n$, $F_i = Y_i^n - y_{n-1} - h\sum_{j=1}^{s} a_{ij} f(t_n + c_i h, Y_j^n)$ and the Jacobian matrix $J_i = \frac{\partial}{\partial y} f(Y_i^n)$ for $i = 1, \dots, s$. Note that each Jacobian is an $m \times m$ matrix, in a system of s differential equations. Hence in each time step an $sm \times sm$ system of equations has to be solved. With this known, it is important to see what options there are to make the implementation less expensive to execute.

By choosing when to recalculate the Jacobian, it is possible to reduce the amount of work needed in each time step. The ideal situation would be to use a single Jacobian for the entire problem. But that would probably result in slow convergence, or in worst case divergence, depending on the problem. As long as the speed of convergence is relatively high, to a given factor, we can use the same Jacobian.

A common method to use is the *simplified Newton's method*. Let the Jacobian be the same for each time step, i.e. $J = J_1 = J_2 = \cdots = J_s$. Now only one LU decomposition per time step is needed. But as a result of that we need to compensate with the speed of convergence, which now is linear and not quadratic. The system (2.3) can now be written with the Kronecker product

$$(I - hA \otimes J)d = -F \tag{2.4}$$

Definition 2.0.9 *The Kronecker product, or the direct product, of two matrices A and B is given by*

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1s}B \\ a_{21}B & a_{22}B & \dots & a_{2s}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{s1}B & a_{s2}B & \dots & a_{ss}B \end{bmatrix}$$

Even though the amount of work per time step is reduced, we still need to solve an $sm \times sm$ system, that means s^2m^2 operations per time step. To reduce the work even further, we consider the matrix A in the choosen Butcher Tableau. If the matrix A is full, or has few zeros, then the system to solve requires more work than if A has more zeros.

For instance there are the Radau methods which are all A-stable methods, with order $2s - 1$, but they have a coeffient matrix A with no zeros. This implies that the Radau methods need alot of work per time step, even though there are certain ways to reduce the execution time for these methods. Finding a method with similar qualities of stability and order, but with less work per time step would be desired. The order of the methods depend on the chosen Butcher Tableau and the stability on an ODE (1.1).

The convergence is influenced by the step size as well. By reducing the step size, the local truncation error gets smaller. But by reducing the step size, we get more time steps and hence more systems to solve. Therefore we want to minimize the work needed per time step before the step size is reduced.

2.1 Diagonally Implicit Runge–Kutta Methods

Consider the Diagonally Implicit Runge–Kutta (DIRK) methods, where $a_{ij} = 0$ for $i < j$ and at least one $a_{ii} \neq 0$ for $i = 0, 1, \ldots, s$. By combining the simplified Newton with DIRK methods we get the following system, with d, F and J defined as in (2.3)

$$\begin{bmatrix} I - ha_{11}J & 0 & \ldots & 0 \\ -ha_{21}J & I - ha_{22}J & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ -ha_{s1}J & -ha_{s2}J & \ldots & I - ha_{ss}J \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_s \end{bmatrix} = - \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_s \end{bmatrix} \tag{2.5}$$

For each Newton step, let us first consider the $m \times m$ system

$$(I - ha_{11}J)d_1 = -F_1$$

After solving this, we substitute the solution d_1 into the next $m \times m$ system

$$I - ha_{22}Jd_2 = -F_2 + ha_{21}Jd_1$$

This forward substitution continues in the same manner at step n

$$I - ha_{nn}Jd_n = -F_n + hJ\sum_{j=1}^{n-1} a_{nj}d_j$$

to construct the solution in each Newton step. This implies that we solve s systems of $m \times m$ linear equations instead of an $sm \times sm$ system at each time step. The number of operations is then reduced to sm^2 from s^2m^2 in the general implicit RK-methods. The amount of work that is needed for each time step is therefore less than RK-methods with a coefficient matrix with more elements.

2.2 Singly Implicit Runge–Kutta Methods

In the Singly Implicit Runge-Kutta methods (SIRK), the coefficient matrix A, only has a single s-fold eigenvalue.

To solve a system (2.2) with Newton's method, the coefficent matrix A must be LU-decomposed. To reduce the amount of work, we can consider matrices that can be transformed by a similarity transformation T such that

$$T^{-1}AT = S$$

where S has the same structure as the DIRK methods. If A only has a distinct eigenvalue, all diagonal elements of S are equal. Thus, only one LU-decomposition of an $m \times m$ system is needed in each time step.

The order and the stability depends on the Butcher Tableau and the ODE.

2.3 Singly Diagonally Implicit Runge–Kutta Methods

Taking advantage of the efficiency from DIRK methods, by reducing the $sm \times sm$ system to s systems of size $m \times m$ each, and combining the method with a SIRK method we get the Singly Diagonally Implicit Runge–Kutta method (SDIRK).

Consider again the stage values in (1.3)

$$Y_i = y_n + h \sum_{j=1}^{s} a_{ij} f(t_n + c_j h, Y_j)$$

with the single-fold eigenvalue of the system $a_{ii} = \alpha$, for all $i = 0, 1, \cdots, s$. After rewriting with the accurate coefficient matrix A, the stage values are now defined as

$$Y_i - h\alpha f(t_n + c_i h, Y_i) = y_n + h \sum_{j=1}^{i-1} a_{ij} Y_j'$$

Then we have $G(Y_i) = Y_i - h\alpha f(t_n + c_i h, Y_i) - y_n - h \sum_{j=1}^{i-1} a_{ij} Y_j'$ and $G'(Y_i) = \mathrm{I} - h\alpha f'(t_n + c_i h, Y_i)$. Thus, the Newton method can now be written as

$$(I - h\alpha J)d = y_n + h \sum_{j=1}^{i-1} a_{ij} Y_j' \tag{2.6}$$

For each time step one still solves s systems of size $m \times m$ each, but with the same matrix $I - h\alpha J$. That means that only one evaluation of the Jacobian and one LU decomposition is needed for each time step.

Regardless of the efficency of implementation, stability and order must be considered for a specific method.

Chapter 3

Implementation

As a part of the working progress of this Bachelor's thesis, the Runge–Kutta methods was implemented in Python. After the previous chapters with the presentation of Runge–Kutta methods, we can now see the problems and benefits when implementing the different methods.

To start with, the class `Onestepmethod()` was implemented [4]. Given the function f and y_0 from the IVP (1.1), with the interval $[t_0, t_e]$, the number of internal points N and a given tolerance, this code segment solves the IVP with a generell one–step method (1.2).

```
class Onestepmethod (object):
    def __init__(self,f,y0,t0,te,N, tol):
        self.f = f
        self.y0 = y0.astype(float)
        self.t0 = t0
        self.interval = [t0 , te]
        self.grid = linspace(t0,te,N+2) # N interior points
        self.h = (te-t0)/(N+1)
        self.N = N
        self.tol = tol
        self.m = len(y0)
        self.s = len(self.b)

    def step(self):
        ti , yi = self.grid[0], self.y0 # initial condition
        tim1 = ti
        yield ti , yi
```

```
        for ti in self.grid[1:]:
            yi = yi + self.h*self.phi(tim1, yi)
            tim1 = ti
            yield ti , yi
    def solve(self):
        self.solution = list(self.step())
```

The class consists of two functions `step()` and `solve()`. `solve()` calles the function `step()` that yields the solution y for all timesteps in the given interval.

When working with Runge–Kutta methods, the function `phi()` evaluates the sum of $b_j Y_j$ in each time step of a Runge–Kutta method (1.3) for $j = 0, 1, \ldots, s$, s the number of stages and Y_j the stage values.

Each time `phi()` is called, a Jacobian matrix $J = \frac{\mathrm{d}f}{\mathrm{d}y}(y_i)$ is evaluated. With this Jacobian, the function `phi_solve()` is called to return the stage derivative Y_i'. Then the sum can easily be computed.

```
class RungeKutta_implicit(Onestepmethod):
    def phi(self, t0, y0):
        """
        Calculates the summation of b_j*Y_j in one step of the
            RungeKutta
        method with

        y_{n+1} = y_{n} + h*sum_{j=1}^{s} b_{j}*Y

        where j=1,2,...,s, and s is the number of stages, b the
            nodes, and Y the
        stage values of the method.

        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the
            length
        of the initial condition y_0 of the IVP.
        """
        M = 10 # max number of newton iterations
        stageDer = array(self.s*[self.f(t0,y0)]) # initial value:
            Y'_0
```

```
        J = jacobian(self.f, t0, y0)
        stageVal = self.phi_solve(t0, y0, stageDer, J, M)
        return array([dot(self.b, stageVal.reshape(self.s,self.m)[:,
            j]) for j in range(self.m)])
```

The function `phi_solve()` is a method of the class `RungeKutta_implicit`. Here starts the Newton iteration, with maximum number of iterations M. If Newton's method does not converge within M iterations, a ValueError is raised. The error can be fixed by recalculate the Jacobian, increase the number of Newton iterations or as a final adjustment, decrease the stepsize h. In this implementation a constant stepsize $h = \frac{te-t0}{N+1}$ is used, which means that the first adjustment should be recalculating the Jacobian or increase the number of iterations.

```
    def phi_solve(self, t0, y0, initVal, J, M):
        """
        This function solves the sm x sm system
        F(Y_i)=0
        by Newton's method with an initial guess initVal.

        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the
            length
        of the initial condition y_0 of the IVP.
        initVal = initial guess for the Newton iteration
        J = m x m matrix, the Jacobian matrix of f() evaluated in y_i
        M = maximal number of Newton iterations

        Returns:
        -------------
        The stage derivative Y'_i
        """
        JJ = eye(self.s*self.m)-self.h*np.kron(self.A, J)
        luFactor = linalg.lu_factor(JJ)
        for i in range(M):
            initVal, norm_d = self.phi_newtonstep(t0, y0, initVal,
                luFactor)
            if norm_d < self.tol:
```

```
            print 'Newton converged in {} steps'.format(i)
            break
        elif i == M-1:
            raise ValueError('The Newton iteration did not
                converge.')
    return initVal
```

For each Newton step, `phi_newtonstep()` is called. The function is a method of the class `RungeKutta_Implicit` and solves the algebraic system (2.4), namely

$$(I - hA \otimes J)d = -F$$

where $d_i = Y_i^{(n+1)} - Y_i^{(n)}$, $F_i = Y_i^{(n)} - y_{n-1} - h\sum_{j=1}^{s} a_{ij} f(Y_j^{(n)})$. The matrix $(I - hA \otimes J)$ is evaluated in `phi_solve()` and its LU-factorization is then used as an input parameter for `phi_newtonstep()`. The matrix A is an $s \times s$ matrix, and the Jacobian an $m \times m$ matrix. Thus the system that is solved has the shape $sm \times sm$.

```
    def phi_newtonstep(self, t0, y0, initVal, luFactor):
        """
        Takes one Newton step by solvning
            G'(Y_i)(Y^(n+1)_i-Y^(n)_i)=-G(Y_i)
        where
        G(Y_i) = Y_i - y_n - h*sum(a_{ij}*Y'_j) for j=1,...,s

        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the
            length
        of the initial condition y_0 of the IVP.
        initVal = initial guess for the Newton iteration
        luFactor = (lu, piv) see documentation for linalg.lu_factor

        Returns:
        The difference Y^(n+1)_i-Y^(n)_i
        """
        d = linalg.lu_solve(luFactor, - self.F(initVal.flatten(),
            t0, y0))
```

```
        return initVal.flatten() + d, norm(d)
```

The right-hand-side F of the algebraic system above, is calculated by the function `F()` in a for-loop.

```
    def F(self, stageDer, t0, y0):
        """
        Returns the subtraction Y'_{i}-f(t_{n}+c_{i}*h, Y_{i}),
            where Y are
        the stage values, Y' the stage derivatives and f the
            function of
        the IVP y'=f(t,y) that should be solved by the RK-method.

        Parameters:
        -------------
        stageDer = initial guess of the stage derivatives Y'
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the
            length
        of the initial condition y_0 of the IVP.
        """
        stageDer_new = empty((self.s,self.m)) # the i:th stageDer is
            on the i:th row
        for i in range(self.s): #iterate over all stageDer
            stageVal = y0 + array([self.h*dot(self.A[i,:],
                stageDer.reshape(self.s,self.m)[:, j]) for j in
                range(self.m)])
            stageDer_new[i, :] = self.f(t0 + self.c[i] * self.h,
                stageVal) #the ith stageDer is set on the ith row
        return stageDer - stageDer_new.reshape(-1)
```

When implementing the SDIRK method the functions `phi_solve()` and `phi_newtonstep()` differs from the general implicit Runge–Kutta methods.

The difference when working with SDIRK methods is the algebraic system to solve. We do no longer work with one $sm \times sm$ system, we solve s systems of size $m \times m$ instead. Consider the system (2.6). The matrix in each newton step is the same, $JJ = (I - h\alpha J)$, and is calculated once in `phi_solve()` before the Newton iterations.

```
class SDIRK(RungeKutta_implicit):
    def phi_solve(self, t0, y0, initVal, J, M):
        """
        This function solves F(Y_i)=0 by solving s systems of size m
            x m each.

        Newton's method is used with an initial guess initVal.

        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the
            length
        of the initial condition y_0 of the IVP.
        initVal = initial guess for the Newton iteration
        J = m x m matrix, the Jacobian matrix of f() evaluated in y_i
        M = maximal number of Newton iterations

        Returns:
        -------------
        The stage derivative Y'_i
        """
        JJ = np.eye(self.m) - self.h*self.A[0,0]*J
        luFactor = linalg.lu_factor(JJ)
        for i in range(M):
            initVal, norm_d = self.phi_newtonstep(t0, y0, initVal, J,
                luFactor)
            if norm_d < self.tol:
            print 'Newton converged in {} steps'.format(i)
                break
            elif i == M-1:
                raise ValueError('The Newton iteration did not
                    converge.')
        return initVal
```

`phi_newtonstep()` for SDIRK methods is constructed with a for-loop that solves an $m \times m$ system in each iteration. This time only one LU-factorisation is needed each time step, which makes the implementation less time consuming.

```
    def phi_newtonstep(self, t0, y0, initVal, J, luFactor):
        """
        Takes one Newton step by solvning
            G'(Y_i)(Y^(n+1)_i-Y^(n)_i)=-G(Y_i)
        where
        G(Y_i) = Y_i - haY'_i - y_n - h*sum(a_{ij}*Y'_j) for
            j=1,...,i-1

        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the
            length
        of the initial condition y_0 of the IVP.
        initVal = initial guess for the Newton iteration
        luFactor = (lu, piv) see documentation for linalg.lu_factor

        Returns:
        The difference Y^(n+1)_i-Y^(n)_i
        """
        x = []
        for i in range(self.s): # solving the s mxm systems
            rhs = - self.F(initVal.flatten(), t0,
                y0)[i*self.m:(i+1)*self.m] +
                sum([self.h*self.A[i,j]*dot(J,x[j]) for j in
                range(i)], axis = 0)
            d = linalg.lu_solve(luFactor, rhs)
            x.append(d)
        return initVal + x, norm(x)
```

The specific Runge–Kutta methods that are used for testing, are created as classes that inherits from the class `RungeKutta_implicit`, either directly or through the class `SDIRK`. There are three different methods that are tested. The first one is a 3-stage Gauss Implicit Runge–Kutta method of order 6[3, p.76].

$$\begin{array}{c|c} c & A \\ \hline & b^{\mathrm{T}} \end{array} = \begin{array}{c|ccc} \frac{1}{2}-\frac{\sqrt{15}}{10} & \frac{5}{36} & \frac{2}{9}-\frac{\sqrt{15}}{15} & \frac{5}{36}-\frac{\sqrt{15}}{30} \\ \frac{1}{2} & \frac{5}{36}+\frac{\sqrt{15}}{24} & \frac{2}{9} & \frac{5}{36}-\frac{\sqrt{15}}{24} \\ \frac{1}{2}+\frac{\sqrt{15}}{10} & \frac{5}{36}+\frac{\sqrt{15}}{30} & \frac{2}{9}+\frac{\sqrt{15}}{15} & \frac{5}{36} \\ \hline & \frac{5}{18} & \frac{4}{9} & \frac{5}{18} \end{array}$$

Table 3.1: Gauss implicit RK-method, $s = 3$

```
class Gauss(RungeKutta_implicit): #order 6
    A=array([[5/36, 2/9 - sqrt(15)/15, 5/36 - sqrt(15)/30],[ 5/36 +
        sqrt(15)/24, 2/9, 5/36 - sqrt(15)/24],[ 5/36 + sqrt(15)/30,
        2/9 + sqrt(15)/15, 5/36]])
    b=[5/18,4/9,5/18]
    c=[1/2-sqrt(15)/10,1/2,1/2+sqrt(15)/10]
```

The second RK-method is a 2-stage SDIRK method of order 3, with $p = \frac{3-\sqrt{3}}{6}$ [1, p.106].

$$\begin{array}{c|c} c & A \\ \hline & b^{\mathrm{T}} \end{array} = \begin{array}{c|cc} p & p & 0 \\ 1-p & 1-2p & p \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Table 3.2: SDIRK method, $s = 2$

```
class SDIRK_tableau2s(SDIRK): # order 3
    p = (3 - sqrt(3))/6
    A = array([[p, 0], [1 - 2*p, p]])
    b = array([1/2, 1/2])
    c = array([p, 1 - p])
```

Finally, the last method is a 5-stage SDIRK method of order 4 [3, p.107].

$$
\begin{array}{c|c} c & A \\ \hline & b^{\mathrm{T}} \end{array} =
\begin{array}{c|ccccc}
\frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
\frac{3}{4} & \frac{1}{2} & \frac{4}{4} & 0 & 0 & 0 \\
\frac{11}{20} & \frac{17}{50} & -\frac{1}{25} & \frac{1}{4} & 0 & 0 \\
\frac{1}{2} & \frac{371}{1630} & -\frac{137}{2720} & \frac{15}{544} & \frac{1}{4} & 0 \\
1 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \\
\hline
 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4}
\end{array}
$$

Table 3.3: SDIRK method, $s = 5$

```
class SDIRK_tableau5s(SDIRK): #order 4
    A = array([[1/4, 0, 0, 0, 0], [1/2, 1/4, 0, 0, 0], [17/50,
        -1/25, 1/4, 0, 0],[371/1360, -137/2720, 15/544, 1/4,
        0],[25/24, -49/48, 125/16, -85/12, 1/4]])
    b = array([25/24, -49/48, 125/16, -85/12, 1/4])
    c = array([1/4, 3/4, 11/20, 1/2, 1])
```

Chapter 4

Testing

Both scalar IVPs and systems of differential equations was tested and each problem is described further in separate subsections. The order of the three methods, described in the previous chapter, was observed by a loglog-plot of the error and the step size in two of the tests. Also the execution time was observed.

4.1 Scalar Differential Equation

The scalar differential equation that was solved is

$$\frac{dy}{dt} = -5y, \quad t \in [0, 1] \quad y(0) = 1$$

with the exact solution $y(t) = e^{-5t}$. This problem was solved with Gauss implicit Runge–Kutta method and a loglog plot of the mean error vs the stepsize was constructed. The plot shows the slope for each method and all slopes are correct compared to the order of each method. 100 different step sizes was used for the plot and the execution time was 19.98 seconds. For each time step the number of iterations for which Newton converged in was printed. As expected when solving a scalar differential equation, Newton converged in a single iteration. This implies that the code solves the problem as expected.

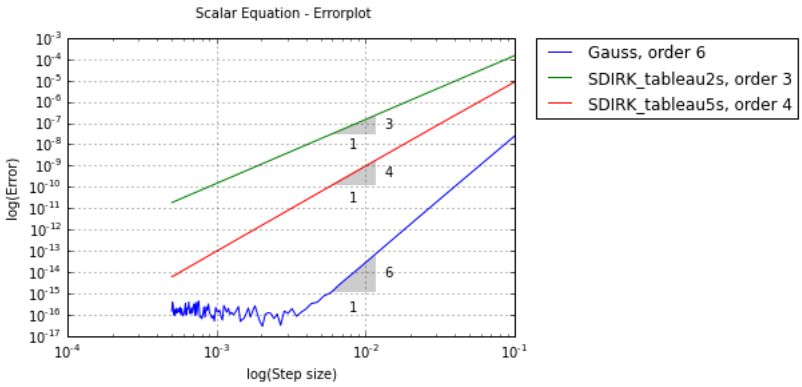

Figure 4.1: Loglog-plot of the mean error vs the step size for 100 distinct step sizes.

As can be seen in the plot there are some irregularities in the error for Gauss method. When the error is that small, around 10^{-15}, it is normal for it to fluctuate when the step size decreases further. The code script can be seen below.

```
def test_scalar():
        t0, te = 0,0.1
        tol_newton = 1e-9
        tol_sol = 1e-5
        N = [2*n for n in range(100)]
        for method in [(Gauss, 6), (SDIRK_tableau2s, 3),
            (SDIRK_tableau5s, 4)]:
            stepsize = []
            mean_error = []
            for n in N:
                stepsize.append((te-t0)/(n+1))
                timeGrid = linspace(t0,te,n+2) #N interior points
                expected = [(t,exp(-5*t)) for t in timeGrid]
                scalar = method[0](lambda t,y: -5*y,
                    array([1]),t0,te,n, tol_newton)
                scalar.solve()
                result = scalar.solution
                error = [abs(expected[i][1] - result[i][1]) for i in
                    range(len(timeGrid))]
                mean = np.mean(error)
```

```
            mean_error.append(mean)
        loglog(stepsize, mean_error, label = '{}, order
            {}'.format(method[0].__name__, method[1]))
        mp.slope_marker((stepsize[8], mean_error[8]),
            (method[1],1))
        suptitle('Scalar Equation - Errorplot')
        xlabel('log(step size)')
        ylabel('log(error)')
        legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    grid()
    assert allclose(result, expected, atol = tol_sol)
```

4.2 System of Differential Equations

4.2.1 Autonomous System

Autonomous systems of differential equations have the form

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y(t)), \quad y(y_0) = t_0, \quad f : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$$

where $f(y(t))$ does not depend directly on the independent variable t.

The problem that is tested from this group of systems is the following

$$y'' + y' = 0, \quad y(0) = 2, \quad y'(0) = 3, \quad t \in [0, 1]$$

By introducing extra variables $y = [y_1, y_2]$ we can construct a first-order ODE

$$\begin{cases} y_1' = y_2 & y_1(0) = 2 \\ y_2' = -y_1 & y_2(0) = 3 \end{cases}$$

$$\Leftrightarrow y' = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

First, Gauss Implicit RK-method was used to solve the problem. The mean of the global error was printed as output. The execution time was 0.049 seconds.

```
The mean error: 1.23551613301e-15
```

When using the two stage SDIRK method, the execution time was 0.053 seconds. The output was

```
The mean error: 1.12786251576e-08
```

Finally the 5-stage SDIRK method was used. The problem was solved in 0.164 seconds and the mean error was

```
The mean error: 1.46622048612e-11
```

Usually SDIRK methods are executed faster than Gauss methods, due to less operations. But these tests showed the opposite. The problem that was solved was only a two dimensional differential equation. Therefore the LU-factorization followed by solving a system of equation(where the solve method is written in C) is faster than the backward substitution written in Python. The difference in time efficency is more clear when working with differential equations with higher dimension. The code script can be seen below.

```
def test_autonomous():
    N = 100
    t0, te = 0, 1
    tol_newton = 1e-9
    tol_sol = 1e-5
    timeGrid = linspace(t0,te,N+2) #N interior points
    system = SDIRK_tableau5s(lambda t,y:
        dot(array([[0,1],[-1,0]]),y),array([2.,3]),t0,te, N,
        tol_newton)
    system.solve()
    result = system.solution
    y1 = np.array([2*cos(t)+3*sin(t) for t in timeGrid])
    y2 = np.array([-2*sin(t) + 3*cos(t) for t in timeGrid])
    expected = [np.array(i) for i in zip(y1, y2)]
    error = [norm(expected[i] - result[i][1]) for i in
        range(len(timeGrid))]
    print 'The mean error: ', np.mean(error)
    assert allclose(zip(*result)[1], expected, atol = tol_sol)
```

4.2.2 Nonautonomous ODE

The nonautonomous ordinary differential equation

$$y'' = (1 - 2t)y, \quad y(0) = 1, \quad t \in [0, 2]$$

has the solution $y(t) = e^{t-t^2}$. This problem was solved with $N = 50$ internal points and the Gauss method. The execution time was 0.081 seconds and the output was the mean error

```
The mean error: 1.14141602153e-12
```

See the code below.

```
def test_nonAutonomous():
    N = 50
    t0, te = 0, 2
    tol_newton = 1e-9
    tol_sol = 1e-4
    timeGrid = linspace(0,2,N+2)
    nonautonomous = Gauss(lambda t,y: y*(1-2*t), array([1.]), t0,
        te, N, tol_newton)
    nonautonomous.solve()
    result = nonautonomous.solution
    expected = [(t,exp(t-t**2)) for t in timeGrid]
    error = [abs(expected[i][1] - result[i][1]) for i in
        range(len(timeGrid))]
    print 'The mean error: ', np.mean(error)
    assert allclose(result, expected, atol=tol_sol)
```

4.2.3 Nonlinear Differential Equation

A differential equation representing the motion of a single pendulum is defined as

$$\theta'' = -\frac{g}{L}\sin(\theta), \quad \theta(t_0) = \theta_0, \quad t \in [t_0, t_e] \tag{4.1}$$

The acceleration constant was chosen to be $g = 13.7503671636040745$, the length $L = 1$ and the interval $[t_0, t_e] = [0, 2]$. The exact solution was calculated by Gauss method and a higher accuracy with $N = 10000$.

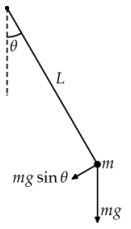

Figure 4.2: Simple pendulum, where θ is the angle, L the length of the pendulum, m the mass and g the acceleration due to gravity.

The pendulum equation (4.1) we want to solve can be written as a system of two first-order differential equations.

$$\begin{cases} \theta_p' = \theta_v & \theta_p(0) = \frac{\pi}{2} \\ \theta_v' = -\frac{g}{L}\sin(\theta_p) & \theta_v(0) = 0 \end{cases}$$

The function plots a loglog error plot to show the order of the different methods. As expected we have the slopes 6, 3 and 4 for Gauss, 2- and 5-stage SDIRK methods respectively. The errorplot that was recieved was the following

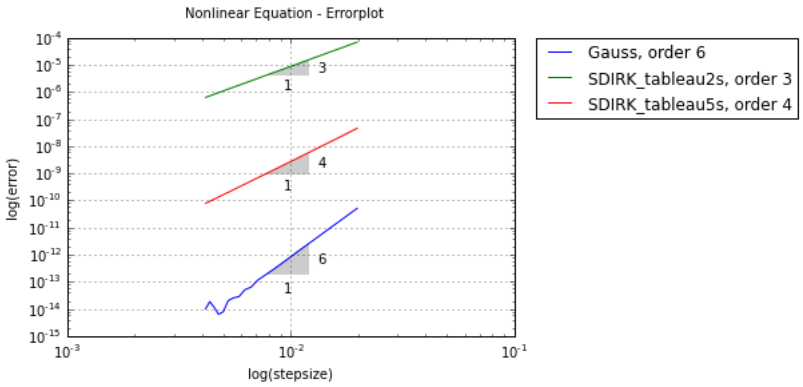

Figure 4.3: Loglog plot of step size vs error for the simple pendulum equation with 20 different step sizes.

The execution time was 61.276 seconds. The script for this test is inserted below.

```
def test_nonlinear():
        g = 13.7503671636040745
        l = 1
        N = [100+20*n for n in range(20)]
        t0, te = 0, 2.
        tol_newton = 1e-9
        tol_sol = 1e-4

        M = 10000 # Number of internal points for exact solution
        nonlinear = Gauss(lambda t,y: array([y[1], -g/l*sin(y[0])]),
            array([pi/2, 0]), t0, te, M, tol_newton)
        nonlinear.solve()
        exact = nonlinear.solution
        pickle.dump(exact, open('Exact_10000', 'wb'))

        for method in [(Gauss, 6), (SDIRK_tableau2s, 3),
            (SDIRK_tableau5s, 4)]:
            error = []
            stepsize = []
            expected = exact[-1][1] # do not want to include the
                time, therefore the [1]
            for n in N:
                nonlinear = method[0](lambda t,y: array([y[1],
                    -g/l*sin(y[0])]), array([pi/2, 0]), t0, te, n,
                    tol_newton)
                nonlinear.solve()
                result = nonlinear.solution
                stepsize.append((te-t0)/(n+1))
                error.append(norm(expected - result[-1][1]))
            loglog(stepsize, error, label = '{}, order
                {}'.format(method[0].__name__, method[1]))
            mp.slope_marker((stepsize[8], error[8]), (method[1],1))
            suptitle('Nonlinear Equation - Errorplot')
            xlabel('log(stepsize)')
            ylabel('log(error)')
            legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
        grid()
```

```
assert allclose(result[-1][1], expected, atol=tol_sol)
```

4.2.4 Multibody System - 2D Truck

Consider a two dimensional multibody system of a truck where the road, with eventual irregularities, is pulled under the truck [5, p.12-15]. Let

$$Mp'' = f_a(t, p, p')$$

where M is the mass matrix and p_i for $i = 1, \ldots, 9$ are the position coordinates of the individual bodies of the truck. p' is the velocity, p'' the acceleration and $f_a(t, p, p')$ the applied force.

The second-order differential equation can be written as a system of two first-order differential equations. With a change of variables $x_1 = p'$, we have

$$\begin{cases} x_1' = p' \\ x_2' = M^{-1} f_a(x, t) \end{cases}$$

Consider the problem at the initial state, when the truck is standing still and the road has not begun to move. Then the velocity of the truck should be zero. For simplicity, consider a single body part of the truck, for instance the motion of the front wheel p_4.

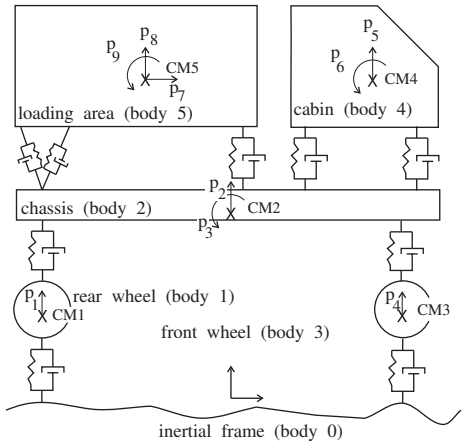

Figure 4.4: Two dimensional multibody system of a truck.

The "x"s in 4.2.4 denotes the center of mass for each individual bodies of the truck.

When testing this problem, there were som stability problems both with and without irregularities on the road. Therefore the road without any roughness was considered, until a solution is found.

Both of the SDIRK methods were tested and it was approximately the same results for both of them. The problem might lie in the convergence of Newton's method. It seems that the initial value for each time step gets worse. Thus, more and more iterations are needed to met the given tolerance. With an increased number of maximum iterations $M = 50$, Newton's method only converges until approximately $t = 0.5$ for $N = 700$, 7000 and 100000 internal point. The endpoint of the interval is $t_e = 2$ and the method was the 5-stage SDIRK method. For the other 2-stage SDIRK method the results differed more when increasing the number of internal points N. When using $N = 700$, Newton's method converges until $t = 0.8$. But when we increase N to $N = 500000$, we get worse results. At the time $t = 0.56$ Newton's method stopped to converge.

One possible reason to this problem is the stability of the function, that the coefficients of the implementation is not set correctly.

Chapter 5

Recomendation for Future Work

The implementation of the implicit Runge–Kutta methods seems to work fine with the correct order of each method. And the fact that Gauss was executed faster than the SDIRK methods were expected due to the small dimension of the problems and how the methods were implemented. It would be interesting to test a larger problem to see the time efficiency benefits with SDIRK methods.

The last test-equation regarding the truck had some stability problems that need to be fixed. For instance it is a good idea to check that the function defining the truck problem is correctly written.

The code can be improved even further by for instance introducing step sizes depending on the speed of convergence in each time step. Before the step sizes are changed we can try to calculate a new Jacobian for the particular Newton iterate. To reduce the amount of work for each time step we can also save the Jacobian for several time steps as long as the speed of convergence is high enough.

References

[1] Uri M. Ascher and Linda R. Petzold. *Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics in Philadelphia, 1998.

[2] John C. Butcher. *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. John Wiley & Sons Ltd, 1987.

[3] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*. Springer-Verlag Berlin Heidelberg, 1991.

[4] Claus Führer. Computional mathematics with python - unit 7: Object oriented programming with classes. http://www.maths.lth.se/na/courses/NUMA22/media/material/unit07.pdf, 2015. Latest used on the 29th June 2015.

[5] Claus Führer and Edda E. Soellner. *Numerical Methods in Multibody Dynamics*. Teubner Stuttgart 1998, 2008.

Bachelor's Theses in Mathematical Sciences 2015:K12
ISSN 1654-6229

LUNFNA-4006-2015

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/