IMPLEMENTING RUNGE-KUTTA SOLVERS IN JAVA

by

Murray Dan Patterson

Thesis

submitted in partial fulfillment of

the requirements of the Degree of

Bachelor of Computer Science with Honours.

Acadia University

April 2003

This thesis by Murray D. Patterson

is accepted in its present form by the

Jodrey School of Computer Science

as satisfying the thesis requirements for the Degree of

Bachelor of Computer Science with Honours.

Approved by the Thesis Supervisor

Dr. J. Diamond   Date

Approved by the Director of the School

Dr. L. Oliver   Date

Approved by the Honours Committee

Date

I, Murray D. Patterson, hereby grant permission to the University Librarian at Acadia University to provide copies of the thesis, on request, on a non-profit basis.

Signature of Author

Signature of Supervisor

Date

Acknowledgements

The research for this thesis was supported financially by NSERC. I also wish to thank all those who assisted or encouraged me throughout this endeavor, particularly Dr. J. Diamond for his ideas and help, and for promptly taking the role of supervisor at Acadia University, given the situation. I wish to especially thank Dr. R. Spiteri, for his wisdom, assistance, and enthusiasm in supervising my work for two successful years.

Table of Contents

Abstract

The theory and implementation of numerical ordinary differential equation (ODE) solvers for initial-value problems (IVPs) are well established. There are many software packages designed for this purpose, the vast majority of which are implemented either in classical programming languages such as FORTRAN or C or in more modern languages such as C++ or Matlab. But, to our knowledge, there has never been such implementation in Java, a programming language that is very common and growing more popular every day.

This thesis describes the Java software package `odeToJava`, which solves ODEs using Runge-Kutta methods. It is available at http://www.netlib.org/ode. To our knowledge, no other such implementation exists in the public domain. `odeToJava` should appeal to the person who wants fast and accurate solutions to stiff and non-stiff IVPs but may be more comfortable using Java than FORTRAN or Matlab.

`odeToJava` includes an embedded Dormand-Prince Runge-Kutta solver with an interpolant similar to Matlab's `ode45`. Step-size control by a step-doubling method can also be used in the absence of embedded methods. It also allows the user to specify an arbitrary Runge-Kutta triple with the options of event location and stiffness detection. For stiff problems, there is a linearly implicit implicit-explicit Runge-Kutta solver.

These solvers are tested on some well-known IVPs, and the numerical solutions are visualized with `odeToJava`'s plotting tool. It turns out that Java is a more competitive medium for the numerical solution of ODEs than some may have originally thought.

Chapter 1

Introduction

In its simplest form, an ordinary differential equation (ODE) gives the rate of change of a quantity as a function of that quantity and possibly time. Because of the way that they implicitly define a function, many things in nature can be expressed in terms of solutions to ODEs. For example, Newton's second law, which states that force equals mass times acceleration, can be viewed as an ODE for position as a function of time. In this thesis, we consider only ODEs in explicit form:

$$\dot{y}(t) = f(t, y(t)). \tag{1.1}$$

That is, we know the derivative of an unknown function $y(t) \in \Re^m$ as a nonlinear function f of t (the independent variable) and $y(t)$ (the dependent variable). This is a very common form for ODEs in many branches of science and engineering.

One type of ODE problem that appears frequently in practice is the initial-value problem (IVP). An IVP is comprised of an ODE (1.1) and a prescribed y_0 value (the initial value) at a certain time t_0. In practice, a time span $[t_0, t_f]$ is also given; i.e., the interval of time over which the problem is to be solved. We consider IVPs of the form:

$$\dot{y}(t) = f(t, y(t)), \quad y(t_0) = y_0, \quad t \in [t_0, t_f]. \tag{1.2}$$

An IVP is solved over the interval $[t_0, t_f]$, starting with the initial value y_0 at time t_0, to produce the function $y(t)$ over this time interval. Solving an IVP using the ODE can be thought of as predicting the path that a quantity will take during a certain time interval, given an initial quantity. For example, chemical reactions can often be described by ODEs [9, test problem 1]. A typical experiment is run from known initial reactant concentrations. Given these initial concentrations and the time interval $[t_0, t_f]$, we can use the ODE that describes this reaction to trace the concentration changes during this time interval.

Sufficient conditions are known under which an IVP has a unique solution [3, p. 12], but obtaining this solution in an analytical form is often too cumbersome, and for many problems, an analytical solution in terms of elementary functions does not exist. However, there are ways to approximate the solutions of most IVPs, such as by solving them numerically. Obtaining the numerical solution to a problem involves the use of a numerical method. Numerical methods can often produce a solution to any degree of accuracy that the computer can represent. Of course, higher accuracy requirements generally mean more computational effort.

Of the several sets of methods for numerically solving IVPs, the set of explicit Runge-Kutta methods [7, §9.6.2] are among the most popular because of their speed and accuracy. The simplest and most basic explicit Runge-Kutta method for solving IVPs is Euler's method [7, §9.2.1], also known as forward Euler. The formula for this method is:

$$y_{n+1} = y_n + hf(t_n, y_n),$$

where y_n is an approximation of the point on the curve $y(t)$ at t_n and y_{n+1} is the approximation at $t_{n+1} = t_n + h$. In this formula, h is called the step-size. Figure 1.1 is a graphical example of what an approximation to the curve $y = e^t$ with forward Euler would look like, where the solid line is the curve $y = e^t$, and the "$\times$" marks are the approximations made at each step. Here the step-size h is 0.1. As the step-size h for this solution is reduced, more steps are needed to solve to t_f, but the approximation becomes more accurate. Forward Euler is very easy to understand and implement, but often it is not as efficient as some higher-order explicit Runge-Kutta methods. These higher-order explicit Runge-Kutta methods operate in a similar fashion to forward Euler in that they approximate the solution to $y(t)$ by stepping to t_f. The difference is that in each step, instead of using just $f(t_n, y_n)$, higher-order explicit Runge-Kutta methods take a weighted average of several function evaluations, typically within $[t_n, t_{n+1}]$. Although this means more computations per step, the accuracy of the solution is much better relative to the amount of work done. The more efficient explicit Runge-Kutta methods are also a bit harder to understand and implement than forward Euler, but the overall gain in efficiency makes implementing them worth this effort.

The above paragraph may suggest that the step-size is held constant throughout the solution of the ODE. This, however, need not be the case; the step-size can be controlled [7, §9.3.3] so that it varies throughout the interval, as needed. Controlling the step-size can make a solver even more efficient, as the step-size can be reduced in parts of the solution that require a small step-size to obtain the accuracy needed, and the step-size can be relaxed where a small step-size is not needed to meet the required accuracy. The step-size of an explicit Runge-Kutta method can be controlled in several different ways. One way of controlling the step-size is by using a so-called embedded method [8], where the explicit Runge-Kutta method has another method embedded within it. At each step of the solution, the approximation that each method gives can be compared to the other to give a measure of the error, and the step-size can be adjusted accordingly. Typically, in the absence of an embedded method, another way to control the step-size is to use step-doubling [6]. With step-doubling,

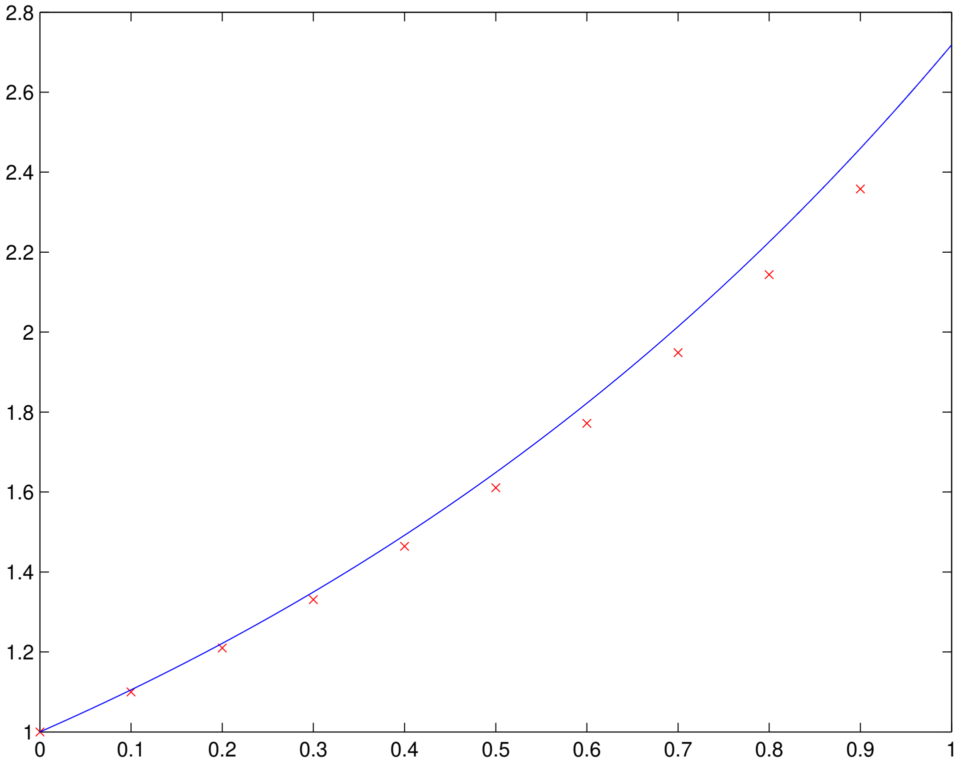

Figure 1.1: forward Euler

an approximation is made using the step-size h, and another approximation is made with two steps using a step-size of $h/2$. These two approximations are then compared to get an estimate of the error, and the step-size is adjusted accordingly. These are the two most common ways to allow for automatic step-size control.

While a numerical solution to an IVP is a set of points (t_n, y_n) that represents an approximation of the solution at a set of discrete points, interpolation is a way to provide a continuous approximation; i.e., a continuous curve that goes through any two points (t_n, y_n), (t_{n+1}, y_{n+1}) and provides a solution that is everywhere in $[t_n, t_{n+1}]$ as accurate as these endpoints [8]. An interpolant allows an ODE solver to perform event location [4]. For example, consider an IVP that represents a projectile traveling through space. An event for that IVP might be when the projectile collides with the ground. In this case, t_f is not specified a priori, but rather must be determined at run-time.

For a given problem and error tolerance, when the step-size used by an explicit Runge-Kutta method with automatic step-size control is being restricted on the grounds of stability and not accuracy, we say the problem is *stiff* [8]. In a Runge-Kutta method, errors in a step are always propagated to future steps. It is only when these propagated errors grow exponentially that the outcome is adversely affected and we say the method is unstable. As an example, consider the IVP $\dot{y} = \lambda y$, $y(t_0) = y_0$, $t \in [t_0, t_f]$, $Re(\lambda) \leq 0$. The exact solution to this problem is $y(t) = y_0 e^{\lambda t}$. A general step of forward Euler on this IVP is $y_{n+1} = y_n + hf(t_n, y_n) = y_n + h\lambda y_n$, which shows that $y_{n+1} = (1+h\lambda)\cdot y_n$. If $|1+h\lambda| \leq 1$ then the numerical solution does not increase just like the exact solution; but if $|1 + h\lambda| > 1$ the numerical solution will increase. This is called numerical instability. If $Re(\lambda) \ll -1$, then h must be chosen to be small enough so that $|1 + h\lambda| \leq 1$. In this case we say that the IVP is stiff; i.e., the step-size h is being chosen to maintain stability, not to provide the requested accuracy. Stiffness detection routines can be implemented so that explicit methods with some form of step-size control can determine if a problem is stiff [2]. If a stiffness detection routine diagnoses an IVP as stiff, a method that has better stability properties than an explicit method should be used to solve it. A set of methods that have the stability required for solving a stiff IVP is the set of implicit-explicit Runge-Kutta (IMEX-RK) methods [5]. Because an IMEX-RK method is implicit, it does more computations per step than an explicit Runge-Kutta method does. The payoff here is that an IMEX-RK method can take larger steps while still meeting the stability requirement, leading to a net gain in efficiency.

There are many fast and robust codes that implement numerical methods for IVPs and allow the user to take advantage of many of the features described. Traditionally, codes were implemented in FORTRAN or C, whereas more modern codes are often implemented in C++ or Matlab. However, there has been no publicly available implementation in the modern programming language Java. This is the topic of this thesis: a description of a software implementation of a set of Runge-Kutta solvers in Java called `odeToJava`. The software is designed to be easy to use, and includes all of the methods and features that are mentioned above. It is also quite efficient, where some of the routines have been shown to outperform similar routines in Matlab. `odeToJava` also has the advantage of being easy to modify or extend, making it both a great pedagogical tool as well as useful for research. It is publicly available at http://www.netlib.org/ode.

This package of Runge-Kutta solvers called `odeToJava` contains six solvers: four explicit solvers and two linearly implicit IMEX-RK solvers. `ErkTriple` is an explicit Runge-Kutta solver that allows the user to implement an explicit Runge-Kutta method, with the options of embedded step-size control and interpolation (an embedded Runge-Kutta method with interpolant is also known as a Runge-Kutta triple). There is also the `DormandPrince` solver, if the user wants to solve the ODE with the Dormand-Prince method [8], an optimal explicit Runge-Kutta triple. These first two solvers can also perform event location and stiffness detection. There are two simpler explicit solvers: `ErkSD`, an explicit Runge-Kutta solver that does step-size control with step-doubling; and `Erk`, an explicit Runge-Kutta solver that simply solves an IVP with a constant step size. Of the IMEX-RK solvers there is `Imex`, a linearly implicit IMEX-RK solver with constant step-size, and `ImexSD`, a linearly implicit IMEX-RK solver that does step-size control with step-doubling. Also included in the package are a set of ODE problems, an interpolant, and a plotter to graphically display numerical solution to an IVP. This package demonstrates that Java can be considered for the numerical solution of ODEs. Some well-known benchmark problems like a non-stiff three-body orbit problem [8], an IVP involving a projectile that collides with several surfaces, and the stiff van der Pol Problem [3, p. 225] have all been solved with good results. The results of these three problems will be explained in detail in Chapter 4.

In Chapter 2, the relevant theory of the different methods and features of the software are explained in detail, including forward Euler, explicit Runge-Kutta methods in general, step-size control, interpolation, event location, stiffness detection, and IMEX-RK methods. Chapter 3 extends the theory of Chapter 2 to explain how these methods and features are implemented in Java to make up the package `odeToJava`. Chapter 4 illustrates the numerical results from three well-known test problems. Finally, Chapter 5 concludes this thesis with a summary and ideas for possible future extension of `odeToJava`.

Chapter 2

Theory

In this chapter, we review the forward Euler method so that explicit Runge-Kutta methods can be explained in more detail. This then allows for an explanation of step-size control by embedded methods and step-doubling. Interpolation, event location, and stiffness detection are then reintroduced in more detail. Chapter 2 ends with a detailed explanation of IMEX-RK methods, methods that prove to be useful when the problem being solved is stiff.

2.1 Forward Euler

Forward Euler is a basic numerical method for solving IVPs. It is the method that many other numerical methods for solving IVPs are based upon, such as Taylor methods, linear multi-step methods, and explicit Runge-Kutta methods. Forward Euler is easy to understand and extremely easy to implement. As is typical of many numerical method for solving ODEs, forward Euler solves an IVP in a series of steps. The idea here is that the smaller the steps taken, the more accurate the solution, but the more computations required.

The solution by forward Euler of the IVP (1.2) over the interval $[t_0, t_f]$, with step-size h_n at the nth step proceeds in this fashion:

$$\begin{aligned} y_1 &= y_0 + h_1 f(t_0, y_0), \\ y_2 &= y_1 + h_2 f(t_1, y_1), \\ &\vdots \\ y_f &= y_{f-1} + h_f f(t_{f-1}, y_{f-1}), \end{aligned}$$

producing the set of points y_1, y_2, $\ldots$, y_f, corresponding to the times t_1, t_2, $\ldots$, t_f. This is a discrete approximate solution to the true solution $y(t)$ over the interval $[t_0, t_f]$.

Forward Euler does successive linear approximations to get a numerical solution to an IVP. We say that forward Euler has first-order convergence. This means that the error at time t_n produced with step-size h is reduced by a factor of 2 when it is approximated with step-size $h/2$. Order of accuracy is defined as follows: for a method of order p, given a step-size h, the error at time t satisfies:

$$e(t; h) \approx e_p(t) h^p \tag{2.1}$$

By this definition, forward Euler is first-order accurate or $O(h)$ accurate. This accuracy might suffice for trivial problems, but greater accuracy is needed when more challenging problems are presented. This can especially be seen for example for Runge-Kutta methods that are $O(h^4)$ accurate. This means that the error at time t_n produced with step-size h is reduced by a factor of 16 when it is approximated with step-size $h/2$.

2.2 Explicit Runge-Kutta Methods

Recall that the major difference between higher-order explicit Runge-Kutta methods and forward Euler is that while forward Euler evaluates the f once at (t_n, y_n) to approximate the value of $y(t_{n+1})$, a higher-order explicit Runge-Kutta method evaluates f more than once (at various points "between" (t_n, y_n) and (t_{n+1}, y_{n+1})), then takes a weighted average of those evaluations to approximate the value of $y(t_{n+1})$. There are many such explicit Runge-Kutta methods, with various stages s and orders p (where s is number of function evaluations used for the approximation at each step).

An example of a higher-order explicit Runge-Kutta method is the classic four-stage fourth-order explicit Runge-Kutta method (ERK4). Given the IVP (1.2) and the point y_n (an approximation to $y(t_n)$), in stepping with a step-size of h_n to compute y_{n+1}, ERK4 evaluates f four times:

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{h_n}{2}, y_n + \frac{k_1}{2}\right), \\
k_3 &= f\left(t_n + \frac{h_n}{2}, y_n + \frac{k_2}{2}\right), \\
k_4 &= f(t_n + h_n, y_n + k_3),
\end{aligned}$$

then takes a weighted average of these four k_i values to compute y_{n+1} (an approximation to $y(t_{n+1})$):

$$y_{n+1} = y_n + h_n \left(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}\right).$$

Like forward Euler, stepping over the interval $[t_0, t_f]$ produces a set of points over the interval, except that they compose a solution that is generally more accurate than that produced by forward Euler.

This allows for extension to the idea of the general explicit Runge-Kutta formula with s stages. Given the IVP (1.2) and the point y_n, in stepping with a step-size of h_n to compute y_{n+1}, we evaluate the s stages:

$$
\begin{aligned}
k_1 &= f(t_n + h_n c_1, y_n), \\
k_2 &= f(t_n + h_n c_2, y_n + a_{2,1}k_1), \\
k_3 &= f(t_n + h_n c_3, y_n + a_{3,1}k_1 + a_{3,2}k_2), \\
&\vdots \\
k_s &= f\left(t_n + h_n c_s, y_n + \sum_{j=1}^{s-1} a_{s,j}k_j\right),
\end{aligned}
$$

then take a weighted average of the k_i values to compute y_{n+1}:

$$y_{n+1} = y_n + h_n \sum_{i=1}^{s} b_i k_i.$$

Note the sets of coefficients used here: the c coefficients $c_1, c_2, \ldots, c_s$, the A coefficients $a_{2,1}; a_{3,1}, a_{3,2}; a_{4,1}, a_{4,2}, a_{4,3}; \ldots; a_{s,1}, a_{s,2}, \ldots, a_{s,s-1}$, and the b coefficients $b_1, b_2, \ldots, b_s$. The values of these three sets of coefficients determine the explicit Runge-Kutta method.

For convenience, Runge-Kutta methods are usually presented in what is called a Butcher tableau [8]. In a Butcher tableau for an s-stage explicit method, the c coefficients are in an array of size s, the A coefficients are in an s-by-s matrix, and the b coefficients are in an array of size s. A Butcher tableau for an s-stage explicit

Runge-Kutta method is of this form:

$$
\begin{array}{c|cccc}
c_1 & & & & \\
c_2 & a_{2,1} & & & \\
\vdots & \vdots & \ddots & & \\
c_s & a_{s,1} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \dots & b_s
\end{array}
$$

As an example, the Butcher tableau for the classical four stage explicit Runge-Kutta method is:

$$
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{1}{2} & & \frac{1}{2} & & \\
1 & & & 1 & \\
\hline
 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array}
$$

Note that the matrix of A coefficients ERK4 is a strictly lower triangular matrix. In fact the s-by-s matrix of A coefficients is always a strictly lower triangular matrix when the Runge-Kutta method is explicit.

2.3 Step-Size Control

During the numerical solution to an ODE, there may be regions where a small step-size is required to attain the desired accuracy, and regions where the step-size can be relaxed and a larger step-size is sufficient to attain the needed accuracy. There are ways that the optimal step-size needed for any step can be estimated, so that the step-size can be adjusted as the solution of the ODE proceeds. The two methods of step-size control described here both involve taking a step with a certain order of accuracy, and then taking that same step with a method of a lower order of accuracy. Given that y_{n+1} denotes the next approximation in a solution, let $\hat{y}_{n+1}$ denote the less accurate approximation. Typically these two approximations differ in order by one; y_{n+1} is $O(h^p)$ and $\hat{y}_{n+1}$ is $O(h^{p-1})$. They can be then compared to give an approximation of the error as follows. Since $y_{n+1} = y(t_n) + O(h^p)$ and $\hat{y}_{n+1} = y(t_n) + O(h^{p-1})$, where $y(t_n)$ is the exact answer, then $y_{n+1} - \hat{y}_{n+1} = O(h^{p-1})$. The subtraction is $O(h^{p-1})$ because in theory h is assumed to approach 0. This error estimate is then used to compute the optimal step-size h_{opt} for this step, given a certain user-specified tolerance. If the error does not meet the tolerance, the current step h_n is rejected, replaced with h_{opt}, and attempted again. If, however, the error does meet the tolerance, the step is accepted and h_{opt} is used in the next step. For each step, adjusting the step-size to the calculated h_{opt} of the previous step will give a more efficient solution that meets the required accuracy.

We now describe the two methods for step-size control: embedded Runge-Kutta methods and step-doubling.

2.3.1 Embedded Runge-Kutta Methods

Consider two Runge-Kutta methods with the same number of stages s, the same set of c coefficients and the same set of A coefficients, represented by the following Butcher tableaux:

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array}$$

$$\begin{array}{c|c} c & A \\ \hline & \hat{b}^T \end{array}$$

Since these two Runge-Kutta methods have identical c and A coefficients, they can be combined into a single Runge-Kutta method containing two sets of b coefficients. In this sense, the combined Runge-Kutta method is one method with another one *embedded* in it, hence the name, *embedded method*. The embedded method can be represented by the following composite Butcher tableau:

$$\begin{array}{c|c} c & A \\ \hline & b^T \\ \hline & \hat{b}^T \end{array}$$

Given a point (t_n, y_n) within a solution, to take a step with a step-size of h_n with this embedded method, we first compute the s common k_i values:

$$k_i = f\left(t_n + h_n c_i, y_n + h_n \sum_{j=1}^{i-1} a_{ij} k_j\right).$$

Then, to get the two different approximations of the next step, y_{n+1} and $\hat{y}_{n+1}$, two different weighted averages of the k values are taken:

$$y_{n+1} = y_n + h_n \sum_{i=1}^{s} b_i k_i,$$
$$\hat{y}_{n+1} = y_n + h_n \sum_{i=1}^{s} \hat{b}_i k_i.$$

The error approximation then starts off with this subtraction, where the error of each component must meet the tolerance:

$$|y_{n,i} - \hat{y}_{n,i}| \leq \delta_i. \tag{2.2}$$

The value of each δ_i component is given by:

$$\delta_i = ATOL_i + \max(|y_{n,i}|, |\hat{y}_{n,i}|) \cdot RTOL_i, \tag{2.3}$$

where $ATOL_i$ and $RTOL_i$ are error tolerances defined by the user so that the accuracy desired can be customized. Since there is an error value for each component of the ODE, as a measure of the total error, the following norm is taken:

$$\varepsilon = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\frac{y_{n,i} - \hat{y}_{n,i}}{\delta_i}\right)^2}, \tag{2.4}$$

where m is the number of components of the ODE. If the step-size h_n taken was the optimal step-size h_{opt} for this step, then ε would be exactly 1. Since $y_{n+1} - \hat{y}_{n+1} = O(h^{p-1})$, $\varepsilon = \theta(h^{p-1}) = Ch^{p-1}$, where C is some constant. If this is so, then $Ch_{opt}^{p-1} = 1$, and it follows that $h_{opt} = h(\frac{1}{\varepsilon})^{\frac{1}{p}}$.

Given this theory on how the optimal step-size is selected, step-size control can be done; however there are a few practical concerns that need attention when doing this. In each step taken, error analysis is done, where ε is computed by (2.4) and h_{opt} is computed. As the method is stepping, if h_{opt} varies too rapidly, this is a sign that the error estimate is unreliable, so a few modifications are done, including multiplication by a safety factor $0 < \alpha < 1$ to increase the likelihood that the next step is accepted. The most common choice is $\alpha = 0.9$ [8]. So h_{opt} is calculated in this manner:

$$\begin{aligned} h_{temp} &= h\left(\frac{1}{\varepsilon}\right)^{\frac{1}{p}}, \\ h_{opt} &= \min\{\alpha_{max}h, \max\{\alpha_{min}h, \alpha h_{temp}\}\}, \end{aligned}$$

where h is the step-size for a given step, $\alpha_{max} = 5$, and $\alpha_{min} = \frac{1}{\alpha_{max}}$. This keeps h_{opt} from varying too rapidly. In any step, if $\varepsilon \leq 1$ to begin with, then the step is just accepted, because the required accuracy is already met, and the next step is tried with the h_{opt} computed in this step (this is as good as any guess for the step-size of the next step). If $\varepsilon > 1$, then the current step is rejected, and retried with the h_{opt} computed here. It has also been shown advisable to set $\alpha_{max} = 1$ after a step rejection [8].

2.3.2 Step-Doubling

Typically, in the absence of an embedded method, step-doubling is done to control the step-size. Given (2.1), and since the initial value $y(t_0) = y_0$ is exact, $e_p(t_0) = 0$, so $e_p(t) \approx e_p(t_0) + \dot{e_p}(t_0)(t - t_0) = (t - t_0)\dot{e_p}(t)$. This implies that $|e(t_0 + h; h)| \approx \varepsilon$, where $\varepsilon \approx |e_p(t_0 + h)h^p| \approx |\dot{e_p}(t_0)h^{p+1}|$. If $\dot{e_p}(t_0)$ can be determined, then an appropriate value of h can be computed. Let $\eta(t; h)$ be an error approximation for a certain step-size h and at a certain time t. If the following are computed:

$$\begin{aligned} \eta(t; h) - y(t) &\approx e_p(t) \cdot h^p, \\ \eta\left(t; \frac{h}{2}\right) - y(t) &\approx e_p(t) \cdot \left(\frac{h}{2}\right)^p, \end{aligned}$$

where $y(t)$ is the exact answer at some time t, then subtracting these gives: $\eta(t; h) - \eta(t; \frac{h}{2}) \approx e_p(t)(\frac{h}{2})^p(2^p - 1)$. Rearranging this gives: $e_p(t)(\frac{h}{2})^p \approx \frac{\eta(t;h) - \eta(t;\frac{h}{2})}{2^p - 1}$, which means that:

$$\eta\left(t; \frac{h}{2}\right) - y(t) \approx \frac{\eta(t; h) - \eta(t; \frac{h}{2})}{2^p - 1}.$$

Suppose the a step-size of H is used to compute $\eta(t+H; H)$ and $\eta(t+H; \frac{H}{2})$. Using the above formula, $e(t+H; \frac{H}{2}) \approx \frac{\eta(t+H;H) - \eta(t+H;\frac{H}{2})}{2^p - 1}$. Since $e(t+H; \frac{H}{2}) \approx e_p(t+H)(\frac{H}{2})^p \approx \dot{e_p}(h)H(\frac{H}{2})^p$, it follows that $\dot{e_p}(t) \approx \frac{1}{H^{p+1}} \frac{2^p}{2^p - 1}[\eta(t+H; H) - \eta(t+H; \frac{H}{2})]$. This implies that:

$$\frac{H}{h} \approx \sqrt[p+1]{\frac{2^p}{2^p - 1} \left\| \frac{\eta(t + H; H) - \eta(t + H; \frac{H}{2})}{\delta} \right\|},$$

which implies that $\frac{H}{h} \approx \sqrt[p+1]{2^p \left\| \frac{e(t+H;\frac{H}{2})}{\delta} \right\|}$, where H is the step-size taken, h is the step-size that should have been taken (h_{opt}), and:

$$\delta_i = ATOL + \max\{|y_{n,i}|, |y_{n+1,i}|\} \cdot RTOL, \tag{2.5}$$

where $y_{n,i}$ is the i^{th} component of y_n, and $y_{n+1,i}$ is the i^{th} component of y_{n+1}, and $ATOL$ and $RTOL$ are user-defined tolerances. This is very similar to the δ_i used with embedded methods, the major difference being that $ATOL$ and $RTOL$ are just scalar values instead of a vector; i.e., a general tolerance for the ODE is defined instead of tolerances for each component of the ODE.

There are also a few practical measures that need to be taken when using this method of step-size control. In each step taken, an error analysis is done, where ε and h_{opt} are computed. In this method some calculations are done including multiplication by a safety factor $0 < \alpha < 1$ as well, to increase the likelihood that the next step is accepted ($\alpha = 0.9$ is a common choice). So h_{opt} is calculated in this manner:

$$\begin{aligned} h_{temp} &= \frac{H}{\sqrt[p+1]{2^p \left\| \frac{e(t+H;\frac{H}{2})}{\delta} \right\|}}, \\ h_{opt} &= \min\left\{ \alpha_{max} H, \max\left\{ \frac{1}{\alpha_{max}} H, \alpha \cdot 2 \cdot h_{temp} \right\} \right\}, \end{aligned}$$

where H is the step-size for a given step and it has been shown useful to set $\alpha_{max} = 5$. This keeps h_{opt} from varying too rapidly. In any step, if $\frac{H}{h_{opt}} \leq 3$ to begin with, then the step is accepted because the required accuracy is already met, and the next step is tried with the h_{opt} computed in this step (this is as good as any guess for the step-size of the next step). If $\frac{H}{h_{opt}} > 3$, then the current step is rejected, and retried with the h_{opt} computed here.

2.4 Interpolation

The numerical solution to an IVP with a Runge-Kutta method is a set of discrete points (t_n, y_n) that represent an approximation to points of the true solution $y(t_n)$. However, if a user wishes to have the y-value at a certain t^* on the approximated curve where $t^* \in [t_n, t_{n+1}]$ for some n, this can be obtained during the computation of the solution. Once an *interpolation* formula is known, a corresponding $y^* \approx y(t^*)$ can be *interpolated*, where y^* has the same order of accuracy as the endpoints y_n and y_{n+1}. Given a step in the solution computation with a Runge-Kutta method from t_n to t_{n+1} with a step-size of h, the array of k_i values is computed. A weighted average of the k_i values is then taken to compute y_{n+1}, using the array of b values. While each of these weights in the b array are used to compute a point that is at $t_n + h$, the b array can be expressed as a function of $\theta \in [0, 1]$, and hence used to compute an approximation at the point $t^* = t_n + h(\theta)$. This way, a corresponding y^* can be computed for any t^*, where $t^* \in [t_n, t_{n+1}]$.

An example of this is an interpolant defined for the Runge-Kutta scheme of Dormand and Prince [1]. The b array for this seven-stage embedded method is:

$$
\begin{aligned}
b_1 &= 35/384, \\
b_2 &= 0, \\
b_3 &= 500/1113, \\
b_4 &= 125/192, \\
b_5 &= -2187/6784, \\
b_6 &= 11/84, \\
b_7 &= 0.
\end{aligned}
$$

For the purposes of interpolation, this b array is expressed as a function:

$$
\begin{aligned}
b_1(\theta) &= \theta^2(3-2\theta)\cdot b_1\theta(\theta-1)^2-\theta^2(\theta-1)^2 5\cdot(2558722523 \\
&\qquad -314030016\theta)/11282082432, \\
b_2(\theta) &= 0, \\
b_3(\theta) &= \theta^2(3-2\theta)\cdot b_3+\theta^2(\theta-1)^2 100\cdot(882725551 \\
&\qquad -157015508\theta)/32700410799, \\
b_4(\theta) &= \theta^2(3-2\theta)\cdot b_4-\theta^2(\theta-1)^2 25\cdot(443332067 \\
&\qquad -314030016\theta)/1880347072, \\
b_5(\theta) &= \theta^2(3-2\theta)\cdot b_5+\theta^2(\theta-1)^2 32805\cdot(23143187 \\
&\qquad -34892224\theta)/199316789632, \\
b_6(\theta) &= \theta^2(3-2\theta)\cdot b_6-\theta^2(\theta-1)^2 55\cdot(29972135 \\
&\qquad -70767360\theta)/822651844, \\
b_7(\theta) &= \theta^2(\theta-1)+\theta^2(\theta-1)^2 10\cdot(7414447-829305\theta)/29380423.
\end{aligned}
$$

It can be easily verified for example that $b_i(0)=0$ and $b_i(1)=b_i, i=1,2,\ldots,7$, as expected. See [8] for more details.

2.5 Event Location

While the right-hand side of an ODE, $f(t,y)$, is often a continuous function, there are applications where $f(t,y)$ can be discontinuous. For example, there can be discontinuities defined for the function where special things happen that cause the ODE to change after the discontinuity. An example of this is a bouncing ball, where the bounces introduce discontinuities in the right-hand side of the ODE. A ball in flight will follow a certain trajectory, but after it collides with something it will follow a different trajectory. The discontinuities in the right-hand sides of these ODEs are usually called *events*, and if they are not located accurately during the solution of the

ODE, then the numerical solution may be very unreliable. A good way to represent the events in an ODE is to define them with a separate function called an *event function*. This event function is any function of t and $y(t)$ (like the ODE) that has a zero whenever an event occurs in this ODE. This event function can have many components, each one defining an event.

How the event locator of `odeToJava` works is that at every step from t_n to t_{n+1}, it checks to see if any component of the event function has changed sign, implying that there is at least one zero within $[t_n, t_{n+1}]$. If there is a zero in this interval, a root-finding method like the secant method [7, §5.2.4] can be used to find the location of this zero more accurately. Since the event function could be defined in terms of $y(t)$, e.g., an event occurs whenever $y_3 = 0$, it needs continuous information from y_3 to locate this zero. The root finder uses interpolation to aid it in isolating the zero (t^*, y^*), where $y_3(t^*) = 0$, $t^* \in [t_n, t_{n+1}]$, using a hybrid method of the secant method and the bisection method [7, §5.2.1].

If there are several events within the interval $[t_n, t_{n+1}]$, the event locator of `odeToJava` detects the first event (the t^* closest to t_n). If an event is detected, the solver halts and outputs the solution (t^*, y^*). It has been shown useful to advance past the event with a very small forward Euler step of size 10^{-14} after an event has occurred before continuing the solver on the interval $[t^*, t_f]$. This is a very basic event locator, but it provides the basic functionality to properly solve IVPs with events.

2.6 Stiffness Detection

During the solution of an IVP with an explicit method, if the step size is being chosen on the grounds of stability and not accuracy, then the IVP is said to be stiff. When an IVP is stiff, a method more stable than an explicit method is desirable because it can take a larger step. We now consider a methodology of quickly diagnosing stiffness in an IVP while it is being solved with an explicit method. Potentially we can then switch to a more stable method for the solution. The stiffness detection routine we have implemented works in two phases. The first phase determines if a stiffness check should be done. In the second phase, a stiffness check is actually performed.

The first phase determines when to check for stiffness. It is too expensive to do this check in every step. `odeToJava` has two ways of determining when to check for stiffness. The first way is to have a set value $MAXFCN$, where, if the number of IVP function evaluations during the whole solution goes above this value, the IVP is checked for stiffness. The second way is that if there is ever a series of steps in the solution where more than 10 steps have been rejected, while less than 50 have been accepted, then the IVP is suspected to be stiff if in addition:

$$h_n \in \left[AMAX \cdot h_{avg}, \frac{h_{avg}}{AMAX} \right], \tag{2.6}$$

$$fevals > MAXFCN \cdot \frac{t_n - t_0}{t_f - t_0}, \tag{2.7}$$

where t_n and h_n are the current time and step-size when this condition is noticed. It has been shown useful to set $AMAX = 5$; h_{avg} is the running average of the step-size since either the solution started, or the last time stiffness was checked due to the ratio of rejected to accepted steps being too large. Also, $fevals$ is the number of IVP function evaluations since the last time stiffness was checked due to the ratio of rejected to accepted steps being too large. The reason for equation (2.7) is that if the number of function evaluations so far is such that it is predicted that the total amount will not exceed $MAXFCN$ anyway, then stiffness is not checked.

The second phase of the stiffness detection routine is the actual stiffness test. There is more than one way to test for stiffness, and an example of one is a method used with the Dormand-Prince method. First ρ is calculated:

$$\rho = \frac{||k_7 - k_6||}{||g_7 - g_6||}, \tag{2.8}$$

where k_6 and k_7 are the last two values in the array of k values. Recall that:

$$k_6 = f\left(t_n + c_6 h_n, y_n + h\sum_{j=1}^{5} a_{6,j} k_j\right), \tag{2.9}$$

$$k_7 = f\left(t_n + c_7 h_n, y_n + h\sum_{j=1}^{6} a_{7,j} k_j\right). \tag{2.10}$$

The values g_6 and g_7 in equation (2.8) are $y_n + h\sum_{j=1}^{5} a_{6,j}k_j$ from (2.9) and $y_n + h\sum_{j=1}^{6} a_{7,j}k_j$ from equation (2.10) respectively. The value ρ calculated in equation (2.8) is what indicates the stiffness of the problem at the given step. After ρ is calculated, it is multiplied by the current step-size h; then $h \cdot \rho$ is compared to the stability region boundary of the method. This is an empirically found measure of the stability of a method; for example, the stability region boundary of the Dormand-Prince method is approximately 3.25. If $h \cdot \rho$ is close to this stability region boundary (e.g., $\rho > 3.25$), then stiffness has been diagnosed with this method, and a more stable method should be used to more efficiently solve this IVP.

2.7 IMEX-RK Methods

IMEX-RK methods [5] are methods that often have the stability needed to solve problems that are too stiff for explicit methods to solve. That is, when solving a problem with an IMEX-RK method, the step-size is not chosen based on the grounds of stability, but rather only based on accuracy.

IMEX-RK methods assume that an ODE can be expressed as a sum of two parts:

$$\dot{y} = f(y) + g(y), \tag{2.11}$$

where $f(y)$ is assumed to be non-stiff and non-linear, and $g(y)$ is assumed to be stiff and linear. Strictly speaking, this means we are considering only linearly implicit IMEX-RK methods. We note that this equation (2.11) is also in autonomous form, without loss of generality. ODEs in autonomous form are such that time is a component of the ODE, instead of being stated separately. If $\dot{y}$ does not naturally divide into stiff and non-stiff parts, we take $g(y) = Jy$ where J is the Jacobian matrix of the ODE right-hand side. This gives:

$$\begin{aligned} g(y) &= Jy, \\ f(y) &= \dot{y} - Jy. \end{aligned}$$

In solving an ODE with an $(s,\hat{s})$-stage IMEX-RK method, where s is the number of stages used to advance the stiff part, and $\hat{s}$ is the number of stages used to advance the non-stiff part in each step, an array of s k-values, and an array of $\hat{s}$ $\hat{k}$-values are computed. For reasons of efficiency, we choose the implicit method to be a singly diagonally implicit Runge-Kutta (SDIRK) method (see, e.g., [8]). In a step, we solve for k_i in $k_i = g(y_i(k_i))$ where

$$y_i(k_i) = y_n + h_n \sum_{j=1}^{i} a_{i,j} k_j + h_n \sum_{j=1}^{i} \hat{a}_{i+1,j} \hat{k}_j,$$

and $y_n \approx y(t_n)$, where h_n is the step-size at the n^{th} step. $a_{i,j}$ is a coefficient of an s-by-s matrix of A coefficients for computing the k array for the stiff part, and $\hat{a}_{i+1,j}$ is a coefficient of an $\hat{s}$-by-$\hat{s}$ matrix of $\hat{A}$ coefficients for computing the $\hat{k}$ array for the non-stiff part. Since $g(y) = Jy$, $k_i = g(y_i(k_i))$ becomes $k_i = Jy_i(k_i)$, thus one can solve for k_i explicitly:

$$k_i = (I - h \cdot a_{i,i} J_n)^{-1} \cdot J_n \left(y_n + h \sum_{j=1}^{i-1} a_{i,j} k_j + h \sum_{j=1}^{i} \hat{a}_{i+1,j} \hat{k}_j \right).$$

The $\hat{k}$ values are computed as they would be in an explicit method, which is normally acceptable because these values are for the non-stiff part of the ODE. After the arrays of k and $\hat{k}$ values are computed, a weighted average of these arrays is taken to compute

y_{n+1}:

$$y_{n+1} = y_n + h_n \sum_{i=1}^{s} b_i k_i + h_n \sum_{i=1}^{\hat{s}} \hat{b}_i \hat{k}_i,$$

where $b_1, b_2, \ldots, b_s$ and $\hat{b}_1, \hat{b}_2, \ldots, \hat{b}_{\hat{s}}$ are arrays of b and $\hat{b}$ coefficients used as weights for computing the weighted averages of the stiff and non-stiff k and $\hat{k}$ values, respectively. The sum of these weighted averages is the value of the next step y_{n+1}.

IMEX-RK methods can also be presented in the convenient form of a Butcher tableau. Because an IMEX-RK scheme is essentially a pair of methods (to handle the stiff and non-stiff parts of an ODE), they are also presented in a pair of Butcher tableaux as:

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array}$$

$$\begin{array}{c|c} \hat{c} & \hat{A} \\ \hline & \hat{b}^T \end{array}$$

where the first tableau contains the coefficients involved in the advancing of the stiff part of the ODE, and the second tableau contains the coefficients involved in advancing of the non-stiff part. $\hat{A}$ is strictly lower triangular, as was mentioned earlier. $\hat{A}$ is generally lower triangular with equal elements on the diagonal. With IMEX-RK methods, $c = \hat{c}$, and often $b = \hat{b}$ as well.

Chapter 3

Implementation

The purpose of this chapter is to demonstrate the functionality of `odeToJava`, relating it to the theory upon which it is based. There are a number of details in learning how to use `odeToJava`. The main directory, `ODE_solvers`, in which the contents of `odeToJava` exist, contains a readme file, `readme.txt`, as well as the directories `testers`, `plotting_files`, `templates`, `jar_files`, and `source_code`. The readme explains all the contents of the `ODE_solvers` directory, so it is a good place to start. The `testers` directory contains a sample program for each solver. The `plotting_files` directory contains files of solutions to certain IVPs that can be plotted so that the user can learn more about the plotter. The `templates` directory contains a template Java file for each solver that can be filled in so that it will run that solver on the ODE specified. The `jar_files` directory is for the jar files needed to run `odeToJava` routines. The `source_code` directory contains the source code of the solvers and modules that are explained in this chapter.

A point of notation: in Chapters 1 and 2, an ODE is defined as $\dot{y} = f(t, y(t))$, and an initial value is defined as $y(t_0) = y_0$. In `odeToJava`, $y(t)$ is represented by `double[] x`, so $y(t)$ is referred to as $x(t)$, and an ODE is referred to as $\dot{x} = f(t, x(t))$

in this chapter. In `odeToJava`, y_0 is represented by `double[] x0` or `double[] u0`, so y_0 is referred to as x_0 or u_0 in this chapter.

3.1 Background on Java

Java is a fairly recent programming language. It became very popular with networking, web design, and graphical user interfaces (GUIs). Java is an interpreted language which is run on a virtual machine. It is a very object-oriented language; only a few languages, such as Smalltalk, are more object-oriented than Java. Java is an interpreted language, and it involves run-time type identification (RTTI) and polymorphism. These properties give the programmer many advantages; however, this can make program execution slow for some applications.

Java has become increasingly popular for many applications, and it is one of the primary languages taught in undergraduate computer science programs. However, it has not been used for many complex applications in scientific computing. Although there exist a few packages in Java that use basic numerical methods, there have not yet been any major packages in Java for the numerical solution of ODEs. To our knowledge, `odeToJava` is the first package written purely in Java that computes numerical solutions to IVPs for ODEs.

3.2 OdeToJava

The major components of `odeToJava` include six different numerical IVP solvers as well as a plotting tool to plot the solutions obtained. `odeToJava` contains four explicit solvers: `Erk`, `ErkTriple`, `DormandPrince`, and `ErkSD`, as well as two IMEX-RK solvers: `Imex` and `ImexSD`. Each of these solvers contains one or more static methods. These methods are such that whenever a class is built to solve an IVP, a static method of the given solver must be called in that class to run that solver. The solvers are best explained by going through the parameter list of the static methods of each solver.

There are some parameters that every static method has in common. These parameters are: `ODE function`, `Span tspan`, `double[] x0` or `double[] u0`, `String fileName`, and `String stats`.

`ODE function` represents the right-hand side of the ODE that is to be solved. `ODE` is an interface class defined in the `modules` directory of the package. This interface class contains two template functions:

```
public double[] f(double t, double[] x);
public double[] g(double t, double[] x);
```

These template functions can be filled in by the user to implement a class of type `ODE` to solve. These two template functions represent the ODE $\dot{x} = f(t, x(t))$, and the event function: $event = g(t, x(t))$. They each take in a `double` parameter `t` and a `double` array parameter `x`, and they return `double` arrays representing $\dot{x}$ and $event$ respectively.

`Span tspan` represents the time span $[t_0, t_f]$ over which the IVP is to be solved. Objects of type `Span` can be defined by one of the three constructors of the class:

```
public Span(double a, double b);
public Span(double a, double b, double inc);
public Span(double[] times);
```

The first constructor has two `double` parameters `a` and `b`, representing t_0 and t_f respectively. The parameters `a` and `b` in the second constructor also represent t_0 and t_f respectively, and its `inc` parameter represents an increment inc where a solution point is reported at every $t_0 + k \cdot inc$, where $k = 0, 1, 2, \ldots, \lfloor \frac{t_f}{inc} \rfloor$. t_f is also reported if it is greater than $\lfloor \frac{t_f}{inc} \rfloor$. The third constructor simply has a `double` array as a parameter, where at each time value in this array, a solution point is reported, and where the first and last values in that array represent t_0 and t_f respectively. Any solver accepts a `Span` object from any constructor, but only solvers with interpolants do interpolation when called with `Span` objects constructed with either the second or the third constructor of the `Span` class. Solvers without an interpolant simply extract the t_0 and t_f information from the `Span` object and solve the IVP without interpolation.

All solvers have a parameter that represents the initial value of the IVP. In the explicit solvers, this is `double[] x0`, representing the initial value x_0 of the IVP. In the IMEX-RK solvers, this is `double[] u0` representing the initial value u_0 of the IVP.

`String fileName` is the name of the file to which the user wishes to write the solution of the IVP. If the file name has a path name with it, then the file is written to the directory specified by the path name. If there is no path name, then the file is written in the directory where the program calling the solver resides. If the file name is simply the empty string `''`, then the solver does not output to file.

`String stats` specifies the level of statistics output the solver uses when solving the ODE. Every solver accepts the three different string values that `stats` can have. If `stats` has the value `''Stats_On''`, then the solver outputs many statistics at every step of the solution, as well as some at the end of the solution. If `stats` has the value `''Stats_Intermediate''`, then the solver outputs a few statistics at every step of the solution, as well as some at the end of the solution. If `stats` has the value `''Stats_Off''`, then the solver only outputs statistics at the end of the solution. The type and quantity of the statistics at each of the three levels also varies according to the solver and the features being used with this solver.

3.2.1 Erk

The solver `Erk` is a constant-step solver that solves an IVP with any s-stage explicit Runge-Kutta method. The signature of a static method of `Erk` is:

```
public static void erk(ODE function, Span tspan, double[] x0,
double h, Btableau butcher, String fileName, String stats);
```

`double h` represents the constant step-size used at every step.

`Btableau butcher` represents the Butcher tableau that determines which explicit Runge-Kutta method is used by `Erk` to solve this IVP. The `Btableau` constructor that is to be used with `Erk` has the following signature:

```
public Btableau(double[][] a, double[] b, double[] c, String FSAL);
```

where `a` represents the matrix of A coefficients of the Runge-Kutta method and `b` and `c` represent the arrays of b and c coefficients of the method. The `FSAL` parameter is one that only concerns explicit Runge-Kutta methods that have the "first same as last" property. In these methods, the final stage k_s of any given step is equal to the first stage k_1 of the next step. So if a method has this property, the `FSAL` parameter can have the value `''FSALenabled''`, and the solver sets k_1 of the current step to the k_s of the previous step (of course, except for the first step of the solution), thus making the algorithm more efficient. The `FSAL` parameter must have the value `''FSALdisabled''` otherwise. These four parameters represent the same properties of the Butcher tableau regardless of the constructor used to construct the `Btableau` object. The `Btableau` class has the classic four-stage fourth-order explicit Runge-Kutta method pre-defined as well, and can be called with the constructor:

```
public Btableau(String special);
```

where the `special` parameter should have the value ``‘‘erk4’’``. This is convenient because this method is quite popular.

The alternate static method for the `Erk` solver is:

```
public static void erk(ODE function, Span tspan, double[] x0, double
h, Btableau butcher, String fileName, String stats, int nPoints);
```

where the `nPoints` parameter determines the amount of points written to the output file. In using the other static method, the number of points defaults to 1000, so that an inexperienced user does not request output from a simulation using a small step-size to solve an ODE over a large range, thus causing so many points to be output to file that it is not tractable.

3.2.2 ErkTriple

The solver `ErkTriple` is a solver that solves an IVP with any s-stage explicit Runge-Kutta method. It solves this IVP with a constant step or with step-size control by an embedded method. It can also do interpolation if an interpolant is specified by the user. Stiffness detection can be done if the user specifies a method of detecting stiffness. Event location is also an option with `ErkTriple`, if the user specifies an interpolant. The signature of a static method of `ErkTriple` is:

```
public static double[] erk_triple(ODE function, Span tspan, double[]
x0, double h0, Btableau butcher, double highOrder, double[] atol,
double[] rtol, String fileName, String stiffnessDet, String eventLoc,
String stats);
```

`double h0` represents the initial step-size h_0 that is used when the solver solves with step-size control; otherwise it is the constant step-size used over the whole interval when the solver solves with a constant step. If the solver is using step-size control, and `h0` is `-1`, then an initial step-size is chosen by the initial step-size selection routine. This routine has the constructor:

```
public Initsss(ODE function, Span tspan, double[] x0, double[] atol,
double[] rtol);
```

It uses the five parameters (of which the last two are explained later) to select an appropriate initial step-size. `Initsss` also has a constructor:

```
public Initsss(ODE function, Span tspan, double[] x0, double[] atol,
double[] rtol, double maxStep);
```

where `maxStep` is maximum size that the initial step-size can be.

A `Btableau` constructor designed to be used with `ErkTriple` to solve the IVP is:

```
public Btableau(double[][] a, double[] b, double[] bEmb, Btheta
btheta, double[] c, String FSAL);
```

where `bEmb` represents the embedded $\hat{b}$ array that allows for step-size control. `btheta` is the parameter that defines the interpolant used for interpolation. The `Btheta` class is an interface class with one method signature:

```
public double[] f(double theta);
```

where this function can be defined by the user so that it represents the `b` array as a function of `theta`. There is one pre-defined interpolant that comes with `odeToJava`, and that is `BthDopr`, an interpolant (see §2.4) for the Dormand-Prince Runge-Kutta method [1]. Since `ErkTriple` can also solve with step-size control without interpolation, or just with a constant step-size, it also accepts `Btableau` objects constructed with the following constructors:

```
public Btableau(double[][] a, double[] b, double[] c, String FSAL);
public Btableau(double[][] a, double[] b, double[] bEmb, double[] c,
String FSAL);
public Btableau(String special);
```

where the first constructor is only for constant-step methods, the second one is for embedded methods with no interpolant, and the third one is this one for pre-defined schemes. In this case, `special` can have the value `''erk4''` so that the classical four-stage fourth-order method is used, but it can also have the value `''dopr54''` so that the Dormand-Prince method is used.

`double highOrder` is the order of the higher-order method of the embedded method. For example, since the Dormand-Prince method is an order-5 method with an order-4 method embedded in it, the `highOrder` argument when using Dormand-Prince should be `5`. If the `highOrder` argument is less than or equal to zero, then the solver solves the ODE without step-size control. This is the only argument that determines whether `ErkTriple` solves with a constant step or with step-size control.

`double[] atol` represents the array of absolute tolerances $ATOL$, where each $ATOL_i$ is used to calculate each δ_i for error estimation with embedded methods (2.3).

`double[] rtol` represents the array of relative tolerances $RTOL$, where each $RTOL_i$ is used to calculate each δ_i for error estimation with embedded methods (2.3).

`String stiffnessDet` determines whether stiffness detection is done. If the string `stiffnessDet` has the value `''StiffDetect_Halt''`, then `ErkTriple` does stiffness detection, and if it has the value `''StiffDetect_Off''` it does not. If stiffness detection is on, and stiffness is detected in the problem being solved, `ErkTriple` simply stops at the time that stiffness was detected and outputs a few statistics. The only available stiffness detection module that `odeToJava` currently has built-in is for Dormand-Prince stiffness detection, but others can be added.

`String eventLoc` determines whether event location is done. If `eventLoc` has the value `''EventLoc_Halt''`, then `ErkTriple` does event location. It does this by checking at every step if any component of the `g` method of the `function` parameter changes sign. If at least one component changes sign, it finds the time of the first sign-change within the interval of the step and returns it. If the `eventLoc` parameter has the value ``EventLoc_Off'', then event location is not done. If event location is on, and an event is located in the problem being solved, `ErkTriple` simply stops at the time that the event was located and outputs a few statistics.

`ErkTriple` also has four alternate constructors that amount to it having two optional parameters that it can have at the end of its parameter list. These two optional parameters are:

```
String append
int nPoints
```

`append` is a file-writing option where if the file being written to already exists, then the data points from the current solution are appended to the file instead of overwriting. The default behavior is for files to be overwritten. `nPoints` is as it was for `Erk`: when `ErkTriple` is solving an IVP with a constant step, it defaults to outputting only 1000 solution points to file, but if `nPoints` is specified, it outputs `nPoints` solution points to file.

Note that `ErkTriple` has a return of type `double[]`, an array representing the solution points, $y_1, y_2, \ldots, y_f$, that is the discrete solution to the ODE corresponding to the times $t_1, t_2, \ldots, t_f$.

3.2.3 DormandPrince

The solver `DormandPrince` is a specialized solver that solves an IVP with the Dormand-Prince Runge-Kutta method [1]. Dormand-Prince is an explicit Runge-Kutta triple, so `DormandPrince` solves an IVP with embedded step size control and has interpolation. Stiffness detection and event location are also options with `DormandPrince`. The signature of a static method of `DormandPrince` is:

```
public static double[] dormand_prince(ODE function, Span tspan,
double[] x0, double h0, double[] atol, double[] rtol, String fileName,
String stiffnessDet, String eventLoc, String stats);
```

`double h0` represents the initial step size h_0 that is used. If `h0` is `-1`, the initial step size is chosen by the initial step size selection routine as it is with `ErkTriple`.

The parameters `double[] atol`, `double[] rtol`, `String stiffnessDet` and `String eventLoc` are handled exactly the same as they are with `ErkTriple`.

The alternate static method for `DormandPrince` is:

```
public static double[] dormand_prince(ODE function, Span tspan,
double[] x0, double h0, double[] atol, double[] rtol, String fileName,
String stiffnessDet, String eventLoc, String stats, String append);
```

This optional parameter `append` is handled the same as it is in `ErkTriple`. Note that `DormandPrince` returns a `double` array, which contains the solution points of the IVP over the specified time interval.

3.2.4 ErkSD

The solver `ErkSD` solves an IVP with any user-specified s-stage explicit Runge-Kutta method. It solves this IVP with step-size control by step-doubling. The signature of the only static method of `ErkSD` is:

```
public static void erk_sd(ODE function, Span tspan, double[] x0,
double h0, Btableau butcher, double atol, double rtol, String
fileName, String stats);
```

`double h0` represents the initial step-size h_0 for the solution. Since the initial step-size selection routine was programmed only for embedded methods, it cannot be used with this solver.

The `Btableau` constructors that are to be used with `ErkSD` are the same ones that are to be used with `Erk`.

`double atol` represents the scalar $ATOL$ that is used for error estimation with step-doubling (2.5).

`double rtol` represents the scalar $RTOL$ that is used for error estimation with step-doubling (2.5).

3.2.5 Imex

The solver `Imex` is a constant-step solver that solves an IVP with any linearly implicit IMEX Runge-Kutta method as described earlier. The signature of a static method of `Imex` is:

```
public static void imex(ODE function, Span tspan, double[] u0, double
h, Btableau butcher, String fileName, String stats);
```

`double h` represents the constant step-size used throughout the whole interval.

The `Btableau` constructor designed to be used with `Imex` is:

```
public Btableau(double[][] a, double[][] ahat, double[] b, double[]
bhat);
```

where `ahat` represents the matrix of $\hat{A}$ coefficients, and `bhat` represents the array of $\hat{b}$ coefficients and `a` and `b` are as described before. The constructor:

```
public Btableau(String special);
```

can also be used with `Imex`, where `special` should have the value `‘‘imex443’’` or `‘‘imex343’’`.

The alternate static method for `Imex` has one extra parameter:

`int nPoints`

This parameter explicitly specifies the number of solution points output to a file. If this parameter is not used, then the number of points output defaults to 1000 as it does for the `Erk` solver.

3.2.6 ImexSD

The solver `ImexSD` solves an IVP with a linearly implicit IMEX Runge-Kutta method as described earlier. It solves this IVP with step-size control by step-doubling. The signature of the only static method of `ImexSD` is:

```
public static void imex_sd(ODE function, Span tspan, double[] u0,
double h0, Btableau butcher, double atol, double rtol, String
fileName, String stats);
```

`double h0` represents the initial step-size h_0 for the solution. The initial step-size selection routine cannot be used with this solver either.

The `Btableau` constructors that are to be used with `ImexSD` are the same ones that are to be used with `Imex`.

The `double atol` and `double rtol` parameters are the tolerances for step-doubling, and they are handled as they are with `ErkSD`.

3.3 Other Features

3.3.1 Plotter

The only other major feature of `odeToJava` that is somewhat independent of the six solvers is the plotting tool, `Plotter`. `Plotter` is easy to call, and one of two static methods can be used to start it. The first method is:

```
public static void showPlotter();
```

Calling this method will bring up a GUI where data can be entered so that a solution can be plotted from a file. The second constructor is:

```
public static void showPlotter(String fileName);
```

where `String fileName` is the name of an existing file that a solution to an IVP is contained. This method saves the user from entering the file name when the GUI comes up.

Chapter 4

Numerical Results

In this chapter we present some numerical results from `odeToJava` applied to solve some well-known benchmark IVPs. The results not only show the performance of the solvers, but they also illustrate the value of many of the features explained in Chapter 2. The following benchmark problems and their results will be explained, analyzed, and presented: the 3-body Orbit Problem [8], the Bouncing Ball Problem, and the van der Pol Problem [3, p. 225].

4.1 The 3-body Orbit Problem

The 3-body Orbit Problem is an IVP that represents the system of a satellite orbiting a planet that orbits a third body in two dimensions, much like the system of our Moon, Earth, and Sun. The initial conditions of this system are the initial positions and velocities of the satellite. The time span is the period of time that it takes for the planet to orbit the third body once. This non-stiff IVP is a good problem for testing the accuracy and efficiency of explicit Runge-Kutta routines because of the periodic nature of its solution, i.e., the solution should be a closed orbit.

The Orbit Problem can be mathematically described as follows. The ODEs describing the orbit of these bodies is the system:

$$
\begin{aligned}
\dot{y}_1 &= y_3 \\
\dot{y}_2 &= y_4 \\
\dot{y}_3 &= y_1 + 2y_4 - \frac{\mu_2(y_1 + \mu_1)}{d_1} - \frac{\mu_1(y_1 - \mu_2)}{d_2} \\
\dot{y}_4 &= y_2 - 2y_3 - \frac{\mu_2 y_2}{d_1} - \frac{\mu_1 y_2}{d_2}
\end{aligned}
$$

$$
\begin{aligned}
d_1 &= ((y_1 + \mu_1)^2 + y_2^2)^{3/2} \\
d_2 &= ((y_1 - \mu_2)^2 + y_2^2)^{3/2}
\end{aligned}
$$

$$
\begin{aligned}
\mu_1 &= 0.012277471 \\
\mu_2 &= 1 - \mu_1,
\end{aligned}
$$

where y_1 and y_2 make up the position of the satellite in two dimensions, and y_3 and y_4 are the velocities of this satellite. d_1 and d_2 are distance-related calculations, and μ_1 and μ_2 are the masses of moon and the planet respectively. The initial values of this IVP are taken to be:

$$
\begin{aligned}
y_1(t_0) &= 0.994 \\
y_2(t_0) &= 0 \\
y_3(t_0) &= 0 \\
y_4(t_0) &= -2.00158510637908252240537862224.
\end{aligned}
$$

The final component of the IVP is the time span:

$$
\begin{aligned}
t_0 &= 0 \\
t_f &= 17.0652165015796255889172062 49.
\end{aligned}
$$

Note the precision of t_f and $y_4(t_0)$; these are the amount of time it takes for the planet to orbit the third body *exactly* once, and the corresponding initial velocity of the moon, respectively. Because these values are so precise, the accuracy of a solver can be measured to a very fine degree by how closely the final values of the solution obtained match the initial values.

The implementation of this IVP can be found in the `functions` directory of the `odeToJava` package, where it is called `Orbit.java`. The IVP is solved with the `Erk` solver, with parameters that can be described by the following piece of Java code:

```
Span span = new Span(0.0, 17.065216501579625588917206249);

double[] x = new double[4];

x[0] = 0.994;
x[1] = 0.0;
x[2] = 0.0;
x[3] = -2.00158510637908252240537862224;

Erk.erk(new Orbit(), span, x, 0.0025, new Btableau(''erk4''),
''Orbit_erk.txt'', ''Stats_Off'');
```

The `Erk` solver solves the Orbit Problem to give the results shown by this output:

```
final t = 17.064999999999504
final solution =
0.851780349627702
-0.13616705377863367
0.5120169567373406
0.03300162153915713
```

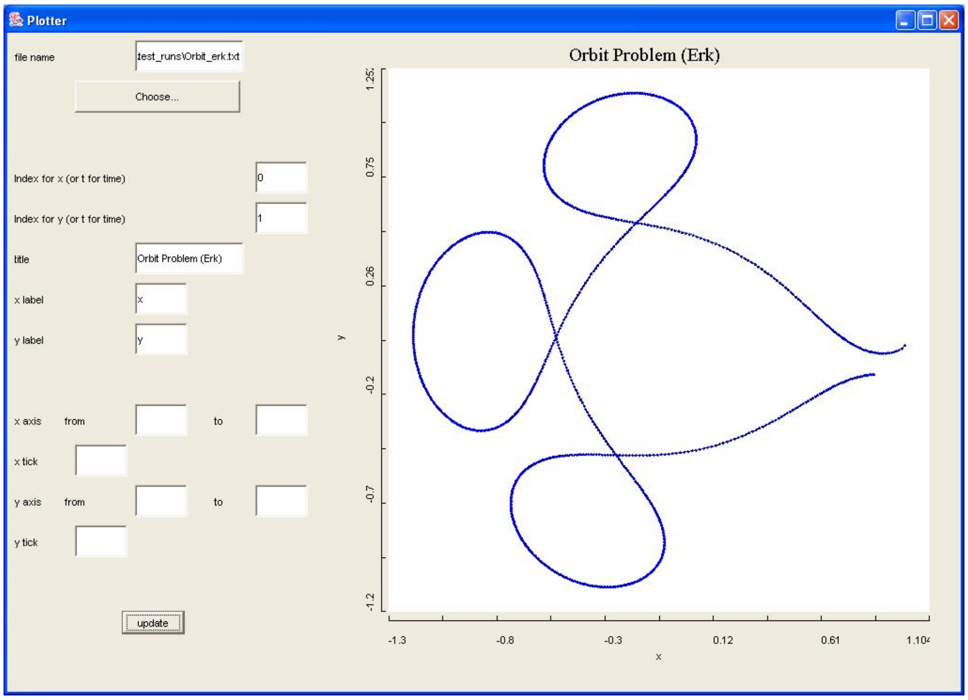

Figure 4.1: Orbit Problem with `Erk`

Note that this final solution deviates from the initial value by a relatively large amount. If the plotter is opened by the call:

```
Plotter.showPlotter();
```

and the file `Orbit_erk.txt` is opened, the two position components of the solution can be plotted against each other to give a visual idea of what the orbit of the satellite looks like. This is seen in Figure 4.1. The inaccuracy of this solution can be shown by the failure of the orbit to close. This orbit almost closes, so if the constant step-size of `0.0025` used here is reduced by just a small amount, the numerical approximation of this orbit does indeed close.

This fact can be illustrated by solving the Orbit Problem with `Erk` with the same parameters again, except with a step-size half the size as the one used above:

```
Erk.erk(new Orbit(), span, x, 0.00125, new Btableau(''erk4''),
''Orbit_erk2.txt'', ''Stats_Off'');
```

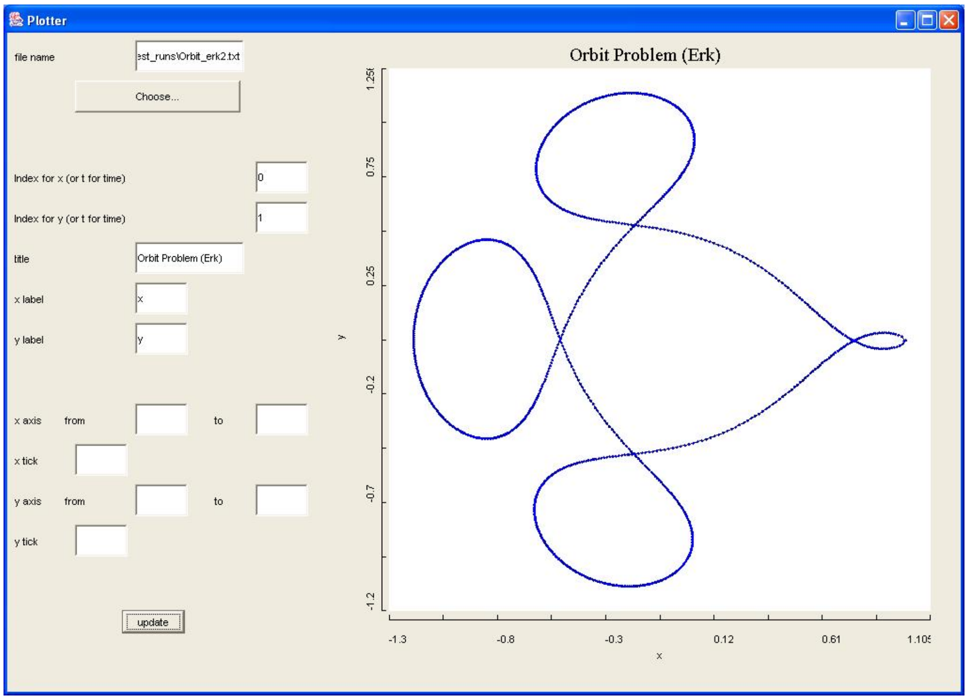

Figure 4.2: Orbit Problem with `Erk`, step-size reduced

The `Erk` solver will solve the problem to give the output:

```
final t = 17.065000000002602
final solution =
0.9850064655279436
-0.0073047833815033155
-1.4974285695163625
-0.929455259876993
```

The plot of this solution is given in Figure 4.2, where we see that the computed orbit is indeed closed.

For a more accurate and efficient solution, the Orbit Problem is solved with step-doubling by solving it with `ErkSD`:

```
ErkSD.erk_sd(new Orbit(), span, x, 0.0025, new Btableau(''erk4''),
1.0E-8, 1.0E-6, ''Orbit_erksd.txt'', ''Stats_Off'');
```

which gives the output:

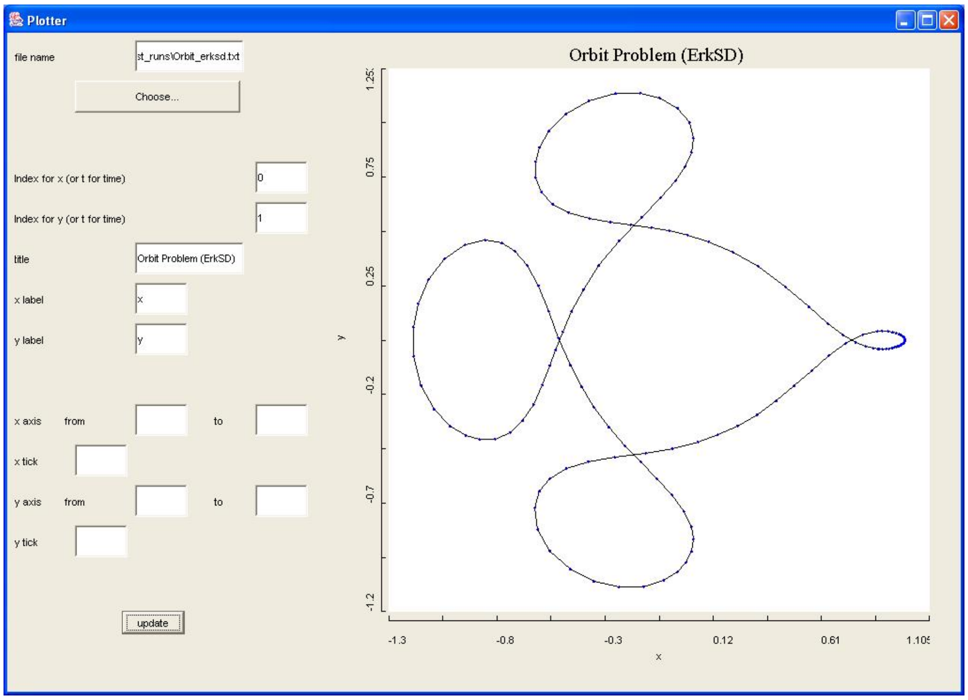

Figure 4.3: Orbit Problem with `ErkSD`

```
final t = 17.065216501579627
final x =
0.9939266554705055
-4.769568698945902E-5
-0.008956863308958826
-2.0130749116835287

# of rejections = 2
# of accepted steps = 139
```

If the file `Orbit_erksd.txt` is then plotted, it gives the solution seen in Figure 4.3. When a solution is plotted, each of the points connected by lines is a solution point in the discrete solution of this IVP. Notice in this graph that the points are spread out on the smoother parts of the solution, whereas they are more dense on the sharp turns, such as upon opening and closing the orbit. This is exactly the advantage that step-size control offers: small steps when accuracy demands it, and larger steps where allowed. As we can see, a superior solution is generated and at only a fraction of the work.

Another way to solve the Orbit Problem with step-size control is to solve it with the embedded solver `DormandPrince`:

```
double[] atol = new double[4];

for(int i= 0; i< 4; i++)
   atol[i] = 1.0E-8;

double[] rtol = new double[4];

for(int i= 0; i< 4; i++)
   rtol[i] = 1.0E-6;

DormandPrince.dormand_prince(new Orbit(), span, x, -1, atol,
rtol, ``Orbit_dopr.txt'', ``StiffDetect_Off'', ``EventLoc_Off'',
``Stats_Off'');
```

This call to `DormandPrince` results in the output:

```
Initial step:  1.599673699019173E-6

      ⋮

final t = 17.065216501579627
final solution =
0.9940358112094593
1.058742671455895E-4
0.017141100650923558
-1.9958238228363971

# of rejections = 38
# of accepted steps = 178
average step size = 0.09587200281786307
```

The differences between this solution and the previous `ErkSD` solution are that it was solved with a higher-order method, and it had an initial step-size selected by the `Initsss` routine. We see that the final value has comparable accuracy to the final value of the `ErkSD` solution. The difference in the solutions is so small that its plot (that of `Orbit_dopr.txt`) is almost indistinguishable from the previous plot; compare Figure 4.3 and Figure 4.4.

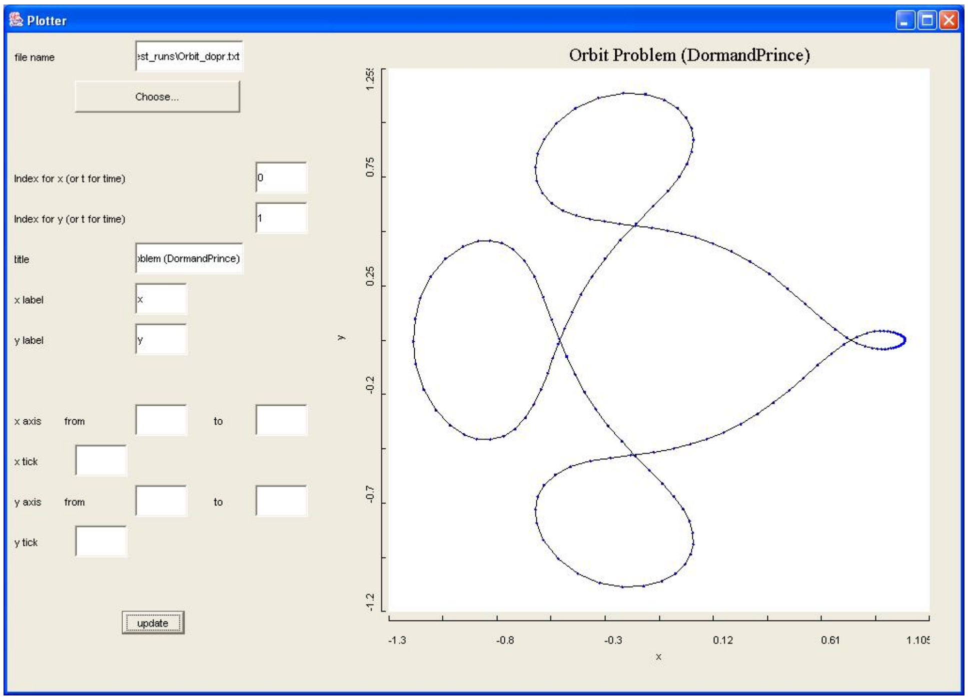

Figure 4.4: Orbit Problem with `DormandPrince`

Although the step-size control of `DormandPrince` may give a solution where points are spread out in some parts and very dense in others, values of the solution with a uniform distribution of points can be achieved with interpolation. To solve the Orbit Problem with `DormandPrince` so that it does interpolation to give equispaced output, it is called as above, except that the span object is defined as:

```
Span span = new Span(0.0, 17.0652165015796255889172062 49, 0.05);
```

so that a point is output at every 0.05 units of time. The solution is still the same, but the output points are very different, as is seen in Figure 4.5.

4.2 The Bouncing Ball Problem

The Bouncing Ball Problem is an IVP that represents the motion of a ball in two dimensions. The ball is assumed to be moving under a constant gravitational field, so that it follows a parabola, unless it comes in contact with an obstacle such as the

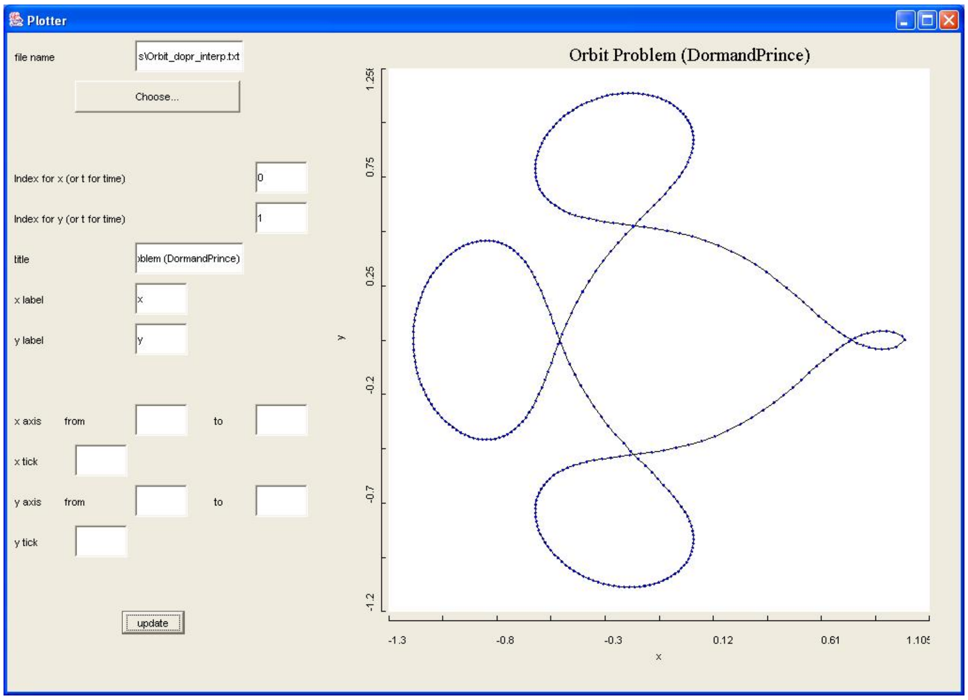

Figure 4.5: Orbit Problem with `DormandPrince`, interpolation on

ground or the wall that are part of the problem definition. The initial conditions of this problem are the initial positions and velocities of the ball. The time span is arbitrary. We solve on a time span long enough for several collisions to take place. This non-stiff problem is one that can be used to test the accuracy of event location routines because an event occurs every time the ball collides with a surface.

The mathematical representation of the Bouncing Ball Problem is as follows. The ODE describing the motion of the ball is:

$$\begin{aligned}
\dot{y}_1 &= y_3 \\
\dot{y}_2 &= y_4 \\
\dot{y}_3 &= 0 \\
\dot{y}_4 &= -G \\
G &= 9.80665,
\end{aligned}$$

where y_1 and y_2 make up the position of the ball in two dimensions (y_1 is the horizontal position, y_2 is the vertical position), and y_3 and y_4 make up the velocity. G is the earth's gravitational constant. The event function that determines when the ball collides with the ground or a wall is:

$$\begin{aligned} g_1 &= y_2 \\ g_2 &= 300 - y_1. \end{aligned}$$

The component g_1 describes that when y_2 (the vertical position) equals 0, the ball hits the ground, and the g_2 component describes that when y_1 (the horizontal position) equals 300, the ball hits a wall. The idea here is that event locator of `odeToJava` is to monitor the values of g_1 and g_2, and report an event whenever one of these components equals 0. The initial value of this IVP that will be used for the example is:

$$\begin{aligned} y_1(t_0) &= 0 \\ y_2(t_0) &= 10 \\ y_3(t_0) &= 40 \\ y_4(t_0) &= 0. \end{aligned}$$

The time span of the Bouncing Ball Problem that is used for the example is:

$$\begin{aligned} t_0 &= 0 \\ t_f &= 60. \end{aligned}$$

The implementation of the Bouncing Ball Problem can be found in the `functions` directory as `Ball_ODE.java`. To solve this problem with `DormandPrince`, taking advantage of the event location that it offers is quite complex, as is shown by this piece of Java code:

```
double[] x = new double[4];

x[0] = 0.0;
x[1] = 10.0;
x[2] = 40.0;
x[3] = 0.0;

double[] atol = new double[4];
for(int i= 0; i< 4; i++)
   atol[i] = 1.0E-6;

double[] rtol = new double[4];
for(int i= 0; i< 4; i++)
   rtol[i] = 1.0E-4;

double t0 = 0.0;
double tStar = t0;
double tf = 60.0;

ODE function = new Ball_ODE();
double hfe = 1.0E-14;
double hInit = -1;

for(int i= 0; i< 8; i++) {

   double[] profile =
   DormandPrince.dormand_prince(function, new Span(tStar,
   tf, 0.01), x, hInit, atol, rtol,
   ``Ball_ODE_dopr_locBounce.txt'', ``StiffDetect_Off'',
   ``EventLoc_Halt'', ``Stats_Off'', ``Append'');

   hInit = profile[0];
   tStar = profile[1];

   x[0] = profile[2];
   x[1] = profile[3];

   if((x[0] <= 305.0) && (x[0] >= 295.0)) {
     x[2] = profile[4] * -0.9;
     x[3] = profile[5];
   }

   if((x[1] <= 5.0) && (x[1] >= -5.0)) {
     x[2] = profile[4];
     x[3] = profile[5] * -0.9
   }
```

```
    x[0] = x[0] + hfe * function.f(tStar, x)[0];
    x[1] = x[1] + hfe * function.f(tStar, x)[1];
    x[2] = x[2] + hfe * function.f(tStar, x)[2];
    x[3] = x[3] + hfe * function.f(tStar, x)[3];

    tStar = tStar + hfe;
}
```

The `if` clauses change the values of some of the solution values depending on whether the ball has hit the ground or the wall, where the `0.9` is the coefficient of restitution of the ball. A small forward Euler step is done after every event so that a duplicate point is not computed. Running this code produces the following output:

```
Initial step:  3.169786384922226E-5

     ⋮

an event occured at t = 1.4280869812290238
done
final t = 1.4280869812290238
final solution =
57.123479249160944
1.509903313490213E-13
40.0
-14.004749194469605

# of rejections = 0
# of accepted steps = 8
average step size = 0.17851087265362797

     ⋮

an event occured at t = 3.9986435474412643
done
final t = 3.9986435474412643
final solution =
159.94574189765055
1.8385293287792592E-13
40.0
-12.604274275022622
```

```
# of rejections = 0
# of accepted steps = 3
average step size = 0.8568521887374102
```

⋮

```
an event occured at t = 6.312144457032293
done
final t = 6.312144457032293
final solution =
252.48577828129172
2.398081733190338E-14
40.0
-11.343846847520503

# of rejections = 0
# of accepted steps = 2
average step size = 1.1567504547955092
```

⋮

```
an event occured at t = 7.500000000000001
done
final t = 7.500000000000001
final solution =
300.0
5.208770767434047
40.0
-1.4394213976758157

# of rejections = 0
# of accepted steps = 2
average step size = 0.5939277714838491
```

⋮

```
an event occured at t = 8.394295275664234
done
final t = 8.394295275664234
final solution =
267.80537007608757
7.993605777301127E-15
-36.0
-10.209462162768471
```

```
# of rejections = 0
# of accepted steps = 2
average step size = 0.44714763783211176
```

⋮

```
an event occured at t = 10.26823101243298
done
final t = 10.26823101243298
final solution =
200.34368355241278
4.39648317751562E-14
-36.0
-9.18851594649159

# of rejections = 0
# of accepted steps = 2
average step size = 0.9369678683843672
```

⋮

```
an event occured at t = 11.954773175524842
done
final t = 11.954773175524842
final solution =
139.62816568110574
9.281464485866309E-14
-36.0
-8.269664351842376

# of rejections = 0
# of accepted steps = 2
average step size = 0.8432710815459261
```

⋮

```
an event occured at t = 13.472661122307523
done
final t = 13.472661122307523
final solution =
84.98419959692923
2.398081733190338E-14
-36.0
-7.44269791665823
```

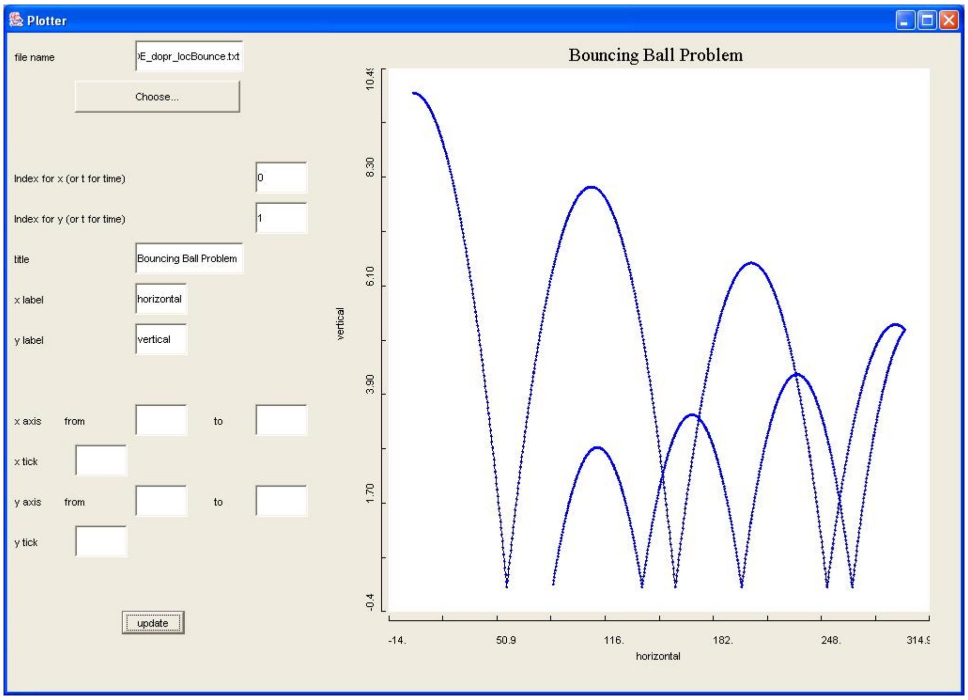

Figure 4.6: Bouncing Ball Problem with `DormandPrince`

```
# of rejections = 0
# of accepted steps = 2
average step size = 0.758943973391335
```

The plot of this solution is seen in Figure 4.6.

4.3 The van der Pol Problem

The van der Pol Problem [3, p. 255] is a stiff IVP with a periodic solution that originates in the theory of electrical circuits. This is a good problem to test the quality of stiffness detection routines or the effectiveness of stiff solvers.

The mathematical description of the van der Pol Problem is as follows. The ODE is:

$$\begin{aligned}\dot{y}_1 &= y_2\\ \dot{y}_2 &= \frac{(1-y_1^2)y_2 - y_1}{\varepsilon},\end{aligned}$$

$$\varepsilon = 10^{-6},$$

where the problem becomes more stiff as ε gets smaller. In our tests we take $\varepsilon = 10^{-6}$, making the problem very stiff. The initial conditions of the problem are:

$$\begin{aligned}y_1(t_0) &= 2.0\\ y_2(t_0) &= 0.0.\end{aligned}$$

The time span that is used in the following examples is:

$$\begin{aligned}t_0 &= 0.0\\ t_f &= 11.0.\end{aligned}$$

The implementation of the van der Pol Problem can be found as `VanDerPol.java` in the `functions` directory of the `odeToJava` package. First, the van der Pol Problem is solved with `DormandPrince` with its stiffness detection option, as is shown by the following piece of Java code:

```
Span span = new Span(0.0, 11.0);

double[] x = new double[2];

x[0] = 2.0;
x[1] = 0.0;

double[] atol = new double[2];
```

```
for(int i= 0; i< 2; i++)
   atol[i] = 1.0E-6;

double[] rtol = new double[2];

for(int i= 0; i< 2; i++)
   rtol[i] = 1.0E-6;

DormandPrince.dormand_prince(new VanDerPol(), span, x, -1,
atol, rtol, ‘‘’’, ‘‘StiffDetect_Halt’’, ‘‘EventLoc_Off’’,
‘‘Stats_Off’’);
```

Running `DormandPrince` on the van der Pol Problem produces the following output:

```
Initial step:  2.523829377920773E-8

      ⋮

problem is stiff due to MAXFCN at t = 0.0018807957286784975
solution at this time:
1.9987457030552078
-0.6673632723077605
# of checks due to MAXFCN: 1
# of checks due to ratio:  0
accepted:  1719
rejected:  281
```

Note that stiffness is detected very early in the process of solving the problem, due to the fact that a successful stiffness check is done after a small amount of time. This stiffness check is done so early because the number of function evaluations done in this amount of time is enough that a check for stiffness should be done.

The van der Pol Problem is stiff, so the effectiveness of IMEX-RK methods as stiff solvers can be tested. To solve the van der Pol Problem with `ImexSD`, the following piece of Java code is run:

```
ImexSD.imex_sd(new VanDerPol(), span, x, 1.0E-6, new
Btableau(‘‘imex443’’), 1.0E-6, 1.0E-6, ‘‘Vand_imexsd.txt’’,
‘‘Stats_Off’’)
```

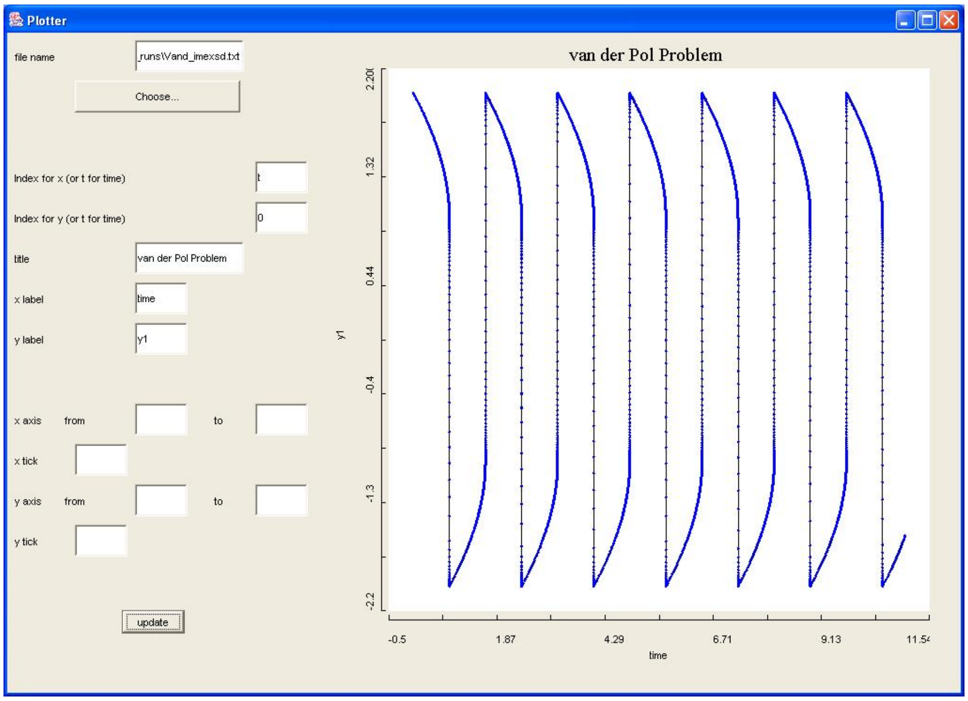

Figure 4.7: van der Pol Problem with `ImexSD`

When this code is run, the following output is produced:

```
final t = 11.0
final u =
-1.5900741541279617
1.040401280189962

# of rejections = 11
# of accepted steps = 7262
```

The plot of the periodic solution contained in ``Vand_imexsd.txt'' is seen in Figure 4.7. This IVP is quite stiff and it is recognized as stiff early in the solution of it with `DormandPrince`. However, it is solved easily with `ImexSD`, with only 11 step rejections for the 7262 accepted steps. This shows how much more effective IMEX-RK methods are than explicit methods for solving stiff problems.

Chapter 5

Conclusion

5.1 Summary

We now summarize the contributions of this thesis. In Chapter 1, ODEs and IVPs are defined, and explicit Runge-Kutta methods are briefly introduced. Then the concepts of step-size control, interpolation, event location, stiffness, and stiffness detection are briefly introduced, leading to the introduction of IMEX-RK methods. There is also mention of some languages in which there are existing implementations of numerical IVP solvers. Finally, `odeToJava` is briefly introduced.

In Chapter 2, much of the theory behind `odeToJava` is explained. The forward Euler method is covered in detail, and the concept of order of accuracy is explained. Explicit Runge-Kutta methods are discussed in detail, starting with an explanation of the classic four-stage fourth-order method. The explanation extends to the general s-stage method, where the Butcher tableau is introduced. Step-size control is explained, where the embedded step-size control and step-doubling methods are described in detail. Interpolation is covered, and the Dormand-Prince interpolant is introduced. Event location, a feature whose implementation requires interpolation, is explained in detail, and some information specific to `odeToJava`'s event locator is given. Stiffness detection is described by briefly introducing numerical instability and the concept of stiffness, then by explaining the stiffness detection routine that is used by the `DormandPrince` solver. IMEX-RK methods, used to solve stiff problems, are then covered in detail.

Chapter 3 describes the implementation of `odeToJava` and its use by focusing on each solver. When a solver is described, all of its possible parameters are listed and explained in detail, giving complete coverage of its features. The constant-step `Erk` solver is introduced first, where its basic parameters are discussed. Many of the solvers have parameters in common with `Erk`. The explicit Runge-Kutta triple solver `ErkTriple` is then described. This solver is the most versatile; consequently it also has the largest list of parameters. There are only a few parameter types that fall outside of the set of parameters that this solver has. The `DormandPrince` solver is then covered. It is a solver that implements the embedded method of Dormand and Prince [1]. `ErkSD`, the explicit Runge-Kutta solver that implements step-doubling is described. The constant-step IMEX-RK solver `Imex` is then discussed. This solver is similar to `Erk` except that it solves with a linearly implicit IMEX-RK method. `ImexSD`, a linearly implicit IMEX-RK solver that implements step-doubling is then explained. This solver allows the user to solve a stiff IVP with step-size control. Finally, we describe the `Plotter`, a tool that allows the user to plot solutions to IVPs after their solutions are output to a file.

In Chapter 4, the numerical results of several well-known test problems are documented. First, the 3-body Orbit Problem is introduced, described mathematically, and then solved in several different ways, illustrating most of the features of the explicit methods of `odeToJava`. The orbit problem is solved with `Erk` and plotted, and then solved with `ErkSD` and plotted to show the advantages of step-size control over constant steps. It is then solved with `DormandPrince` and plotted to show the solution with embedded step-size control. The Orbit Problem is then solved with

`DormandPrince` again, but this time an interpolated solution is output and plotted. The Bouncing Ball Problem, a problem that is good for testing event location, is then introduced. It is solved with the `DormandPrince` solver with event location, so that each ball bounce event can be precisely located for an accurate solution. This solution is then output and plotted to show how the event location routine functions. The stiff van der Pol Problem is then described. This problem is solved with `DormandPrince` with stiffness detection to show how the feature detects stiffness in this IVP. The van der Pol Problem is then solved with `ImexSD` and plotted to show the solution of this stiff problem. We illustrate the effectiveness of using IMEX-RK methods on stiff problems.

5.2 Future Work

The package of numerical ODE solvers `odeToJava` is quite sophisticated and contains many features for custom solutions to IVPs. Despite this, there are ways that it can be extended and improved. A few ideas for extensions are described here.

One feature of `odeToJava` that could easily be improved is the event locator. As described earlier, it is quite primitive; when an event is detected, the solver halts with the solution at the time of the event. How the problem is handled after that event is the responsibility of the user. A few more features could be added to the solvers with event location so that they handle events more gracefully, making them easier to use.

It would be useful to add a generic solver that would automatically select an explicit or an IMEX-RK method, depending on whether stiffness is detected in the IVP. This would introduce a new degree of adaptivity into `odeToJava`, by which almost any problem, whether stiff or not, could be handled with this solver automatically. This would be useful for users who do not know much about `odeToJava` and are more concerned with just getting reliable output.

While `odeToJava` has solvers mainly for non-stiff problems, features could be added to make `odeToJava` better at handling stiff problems. First, other forms of stiffness detection than the one presented here could be added. There are also other stiff solvers besides IMEX-RK. For problems that are extremely stiff, a fully implicit solver would be a good addition to `odeToJava`.

There are also many other minor improvements that could be made to `odeToJava`, such as improving the interface so that it is even more user friendly. The code could also be refactored, and more object-oriented techniques could be used to improve the readability, modularization, and utility of the code.

Bibliography

[1] J. Dormand and P. Prince, *A family of embedded Runge-Kutta formulae*, J. Comp. Appl. Math., **6** (1980), 19–26.

[2] L.F Shampine, *Diagnosing stiffness for Runge-Kutta methods*, SIAM J. Sci. Stat. Comput., **12(2)** (1991), 260–272.

[3] R.K. Nagle, E.B. Saff, and A.D. Snider, *Differential Equations and Boundary Value Problems*, Addison Wesley Longman, 2000.

[4] L.F. Shampine and S. Thompson, *Event location for ordinary differential equations*, Comput. Math. Appl., **39** (2000), 43–54.

[5] U.M. Ascher, S.J. Ruuth and R.J. Spiteri. *Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations*, Appl. Numer. Math., **25** (1997), 151–167.

[6] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis: Second Edition*, Springer-Verlag, 1993.

[7] M.T. Heath, *Scientific Computing: An Introductory Survey*, McGraw-Hill, 1997.

[8] E. Hairer, S.P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer-Verlag, 1993.

[9] W.M. Lioen and J.J.B. de Swart, *Test set for initial value problem solvers*, CWI Library, 1999.