Implementation of Simple Real-Time Systems in Java using the Advance Action Approach

John Knipping

Katholieke *Universiteit* Nijmegen

Experimental
Informatics
for
Technical
Applications

Supervisor: Dr. B.P.F. Jacobs
Master's Thesis No. 430, May 25, 1998

Department of Computer Science
Faculty of Mathematics and Informatics

Implementation of Simple Real-Time Systems in Java using the Advance Action Approach

Master's Thesis No. 430

John Knipping

Supervisor:
Dr. B.P.F. Jacobs

Experimental Informatics for Technical Applications
Department of Computer Science
Faculty of Mathematics and Informatics
University of Nijmegen

May 25, 1998

Preface

When the time arrived to start thinking about a graduation project to finish my study of Computer Science at the University of Nijmegen, I started to gather information about possible projects. One of these projects concerned the implementation of a formalism, called the *Advance Action Approach*, in the JAVA programming language. This graduation project was supervised by dr. Bart Jacobs.

I was particularly interested in this project, since it involved a combination of several scientific disciplines, namely Computer Science, Mathematics and Physics. For the latter two I had to do some research on differential equations and their approximation methods, and I had to refresh my knowledge of mechanics. Another aspect, which I found attractive in this graduation project, was the use of the relatively new object oriented programming language JAVA. In this way I was able to gain some more experience in this language.

I would like to thank dr. Bart Jacobs for supporting me throughout this graduation project. Further, special thanks to my girlfriend Vivian for loving and supporting me, and spending a large amount of her time to correct my English.

Finally, I would like to thank my parents Jan and Riekie, and my brother Geraldo. They have given me all the support I could have wished.

John Knipping,
Nijmegen, May 1998.

Contents

List of Figures

Chapter 1

Introduction

In the papers [1] and [2] a formalism is described in abstract mathematical terms to introduce *real-time* aspects in an object-oriented setting. In short this formalism means that a *real-time*-class must have a special *advance-method* which, in a given state, defines how an object of that class has developed after a certain amount of units of time. Here it concerns objects with autonomous activity. The above formalism is called *Advance Action Approach*, or *AAA* for short.

The aim of this graduation project is to give concrete form to this *AAA* formalism in the JAVA programming language [3, 4, 5] of Sun Microsystems, Inc. To be more specific, the project concerns the implementation, in the JAVA programming language, of a number of exemplaric models according to the *Advance Action Approach*, by the approximation of differential equations, which describe the behaviour of the model, in combination with internal and external events. Furthermore, research is being done to the possibilities and problems that might occur using the *Advance Action Approach* in the JAVA programming language. This research will focus mainly on the hybrid aspects that occur during (discrete) internal and external events.

This thesis discusses the research which was being done and gives an explanation of the results of this graduation project. It will focus among other things on the possibilities of using the *Advance Action Approach* when implementing simple *real-time* systems in the JAVA programming language.

There are many interpretations of what a *real-time* system exactly is. However, they all have in common the notion of response time, the time taken for the system to generate output from some associated input. In [6] Burns and Wellings quote [7]. A *real-time* system is:

> …any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.

Further Burns and Wellings add:

> …the correctness of a real-time system depends not only on the logical result of

> the computation, but also on the time at which the results are produced.
>
> …hard real-time systems are those where it is absolutely imperative that responses occur within the specified deadline. A soft real-time system is by extension one in which the deadlines are desirable but not imperative.

The simple *real-time* systems, which are implemented during this graduation project, are a number of exemplaric physical models of which the behaviour can be described by one or more differential equations. These simple *real-time* systems are typically *soft real-time* systems, tasks are performed by the system as fast as possible, but they are not constrained to finish by or within specific times. However, the system keeps track of response times, and may warn the user or stop the application, when certain threshold response times are not met.

Throughout this thesis three models will be discussed. It concerns the model of a pendulum, a bouncing ball and the model of a double pendulum. These models of simple *real-time* systems are discussed from physical and mathematical background to simulation in the simulationtool OMSIM and implementation in the JAVA programming language.

Chapter 2

The Advance Action Approach

This chapter discusses the main idea of the *Advance Action Approach*. It will focus on the characteristics of the *Advance Action Approach*, including the JAVA code which provides an implementation. Further this chapter will focus on internal and external events in the models mentioned in chapter 1.

2.1 The main idea

Applications running on a computer normally do not run at a constant speed. There will always be a number of processes which compete for time to use the Central Processing Unit (CPU), according to a certain scheduling algorithm. Further, these processes can have different priorities. It should be clear that the available CPU time for a proces depends on these factors. Another important factor is the speed of the CPU. A proces running on a fast CPU will be finished in a shorter amount of time than a proces running on a slower CPU.

The *Advance Action Approach* provides a way to run applications apparently at constant speed, irrespective, to some extend, of available CPU time and speed of the CPU. The main idea is the extension of the coalgebraic approach to hybrid specifications in [1, 2] by making use of an advance action (a special function), which describes the influence of time on the state space of a system:

$$\mu : (States) \times (Time) \to (States).$$

To be used in a computerprogram the above is translated into the following idea:

- Record the current time.
- Record the difference between the current time and the time of the last advance action.
- Compute a new state from the current state together with the difference between the current time and the time of the last advance action.

This is not a new idea. For example, it is used in the BouncingHead demo which is included in the Java Development Kit (JDK) 1.0.2. In this graduation project however, the *Advance Action Approach* is exploited systematically for structuring applications, and the use of differential equations, that describe the motion of a system, in combination with internal and external events is introduced.

2.2 The Java code

When the idea of the *Advance Action Approach* is implemented in a computerprogram written in the JAVA programming language, the code will look like this:

The main control loop:

```
while (condition) {
  for (int i = 0; i < current_number_of_objects; i++) {
    object[i].advance();
  }
}
```

In class of object[i] use:

```
long current_time, delta, last_advance;
long max_time_step;

public void advance() {
  current_time = System.currentTimeMillis();
  delta = current_time - last_advance;
  last_advance = current_time;
  if (delta > max_time_step) {
    // take measures,
    // e.g. give warning only,
    // stop the application,
    // or increase the priority
  }
  advance(delta);
}

public void advance(double d) {
  // compute a new state from the current state
  // together with the time interval d
}
```

A simple example that explains the code is a one handed clock. When the clock is in a certain state, due to the last call of the `advance()` method, a new call of the `advance()`

method results in a computation of `delta`, which is the difference between the current time and the time of the last advance action. If `delta` is larger than the predefined amount of `max_time_step`, appropriate measures can be taken. After this, a new state is computed from the current state together with the time interval `delta`. This means that, when `delta` is equal to $t\ ms$, in the new state of the clock the hand will indicate a time that is $t\ ms$ later than the previous indicated time. In this way the clock will run apparently at constant speed, independent of how much computing resources it gets: if it gets much processor time, `delta`'s will be small so that the clock's hand takes only small steps. But if it gets little processor time the `delta`'s will be larger, and hence the clock's hand will make a larger step in each advance.

A variation of the *Advance Action Approach* is used in the AnimatorApplication demo which is included in the JDK 1.1.3. A variable *delay* is being used which indicates the time between two successive animation frames. If there is still some time left after all the processing has been done, the application sleeps until the time between two successive animation frames is equal to *delay* milliseconds. This ensures a constant frame rate, which is not the case when using the *Advance Action Approach*. The disadvantage of the approach in this demo is that when the time between two successive animation frames is exceeded, nothing is being done. The application proceeds normally without any sleeptime and without any adjustment to the system, so it does not run apparently at constant speed, like applications which make use of the *Advance Action Approach*.

2.3 Characteristics

There are a number of characteristics of the *Advance Action Approach*, which are summarized below:

- The *Advance Action Approach* provides a way to run applications apparently at constant speed, irrespective of available CPU time and speed of the CPU.
- *Advance Action Approach* programs are like simulations. They compute according to certain control laws, for example a differential equation which describes the motion of a certain model. They can do this *real-time*, or with increased or decreased speed.
- The dynamics of an object, as described for example by a collection of differential equations, can be used directly in the code of the class of the object. Computing the state after Δt means solving, by some approximation method, the differential equations over the interval Δt. This leads to a high-level way to capture the intended behaviour.
- By testing Δt one knows how much computing resources the system gets. This may lead to appropriate soft or hard action, possibly involving adjustment of priorities.
- External (discrete) events can be handled as follows:
  - record the time when they happen
  - adjust the system to this movement

  - handle the event
  - and proceed normally.

  This involves a typically hybrid combination of discrete and continuous phenomena. An event may change the initial values of differential equations, affecting the continuous behaviour after the event.

- The *Advance Action Approach* is specifically appropriate in object-oriented programming where one deals with multiple objects, each with their own private state space. Objects with dynamic behaviour belong to classes which have an advance action describing the influence of time on the objects state space. The uniformity of this approach makes it even possible to use (multiple) inheritance and introduce a single, deferred or virtual, *Advance Action Approach* class containing the common code.

How the *Advance Action Approach* can best be combined with parallellism is not clear yet. This topic is out of the scope of this graduation project.

2.4 Internal and external events

The motion of the models mentioned in chapter 1, a pendulum, a bouncing ball and a double pendulum, can be described by a collection of differential equations. When a model that makes use of the *Advance Action Approach* is implemented, these equations can be used directly in the code. There are also a number of hybrid aspects in these models, which means that discrete and continuous dynamics are combined. The discontinuities in the control law are caused by actions from within the system, or by actions from outside the system. The first of these are known as internal events, and the latter are called external events.

In this graduation project, among other things, research is being done on the possibilities and problems that might occur when these models are implemented in the JAVA programming language, using the *Advance Action Approach* in combination with the handling of internal and external events. Therefore, in these models a number of internal and external events are defined. In the pendulum model it is possible to give a tap on the pendulum, which is a typical external event. The model of the bouncing ball has a typical internal event, namely the bounce of the ball on the floor. Finally, the double pendulum model combines an internal and an external event. It is possible the two pendulums collide, which is an internal event, and it is possible to give a tap on each of the pendulums, which is a typical external event.

Chapter 3

Differential equations

This chapter discusses the basic concepts and ideas of differential equations. It will focus on ordinary differential equations of first and higher order. Further, it focusses on differential equations which cannot be solved with an exact method.

3.1 Basic concepts and ideas

Differential equations are of fundamental importance in engineering mathematics. This is due to the fact that many physical laws and relations can be described mathematically in the form of a differential equation. The motion of the models discussed in this thesis can also be described by differential equations.

By an *ordinary* differential equation, often shortened to ODE, a relation which involves one or several derivatives of an unspecified function y of x with respect to x is meant. The relation may also involve y itself, given functions of x, and constants.

The term ordinary distinguishes the differential equation from a *partial* differential equation, which involves partial derivatives of an unspecified function of two or more independent variables. However, this graduation project only focusses on ordinary differential equations.

An ordinary differential equation is said to be of *order* n, if the nth derivative of y with respect to x is the highest derivative of y in that equation. The notion of the order of a differential equation leads to a useful classification into equations of first order, second order, etcetera.

Ordinary differential equations can be divided into two large classes, the *linear* equations and the *nonlinear* equations. For example, a first order differential equation is said to be linear if it can be written in the form

$$y' + f(x)y = r(x).$$

A second order differential equation is said to be linear if it can be written in the form

$$y'' + f(x)y' + g(x)y = r(x).$$

The characteristic feature of such an equation is that it is linear in y and its derivatives, while f, g and r may be any given functions of x. Any equation of the first and second order which cannot be written in the corresponding form is said to be nonlinear.

Moreover, if $r(x) = 0$ for all x, the equation is said to be *homogeneous*. Otherwise, it is said to be *nonhomogeneous*.

In the case of a third or higher order differential equation these definitions are analogous to that in the case of an equation of the first or second order.

The last concept which should be mentioned is that of the *initial value problem*. This means a differential equation together with an initial condition. To solve such a problem a particular solution of the equation, satisfying the given initial condition, has to be found.

3.2 First order differential equations

Differential equations of the first order can be written as:

$$F(x, y, y') = 0.$$

This way of writing is called the *implicit* form. Not always, but in many cases a first order equation can also be written in the *explicit* form:

$$y' = f(x, y).$$

The way one chooses to write a differential equation does not matter. Writing it in another way does not alter the differential equation, it is just another way of presenting the same equation.

A function

$$y = f(x)$$

is called a *solution* of a given first order differential equation on some interval, say $a < x < b$ (perhaps infinite), if it is defined and differentiable throughout the interval, and if it is such that the equation becomes an identity when y and y' are replaced by f and f' respectively. For example, the function

$$y = f(x) = e^{2x}$$

is a solution of the first order equation

$$y' = 2y$$

for all x, because

$$f'(x) = 2e^{2x},$$

and by inserting f and f' the equation is reduced to the identity

$$2e^{2x} = 2e^{2x}.$$

A differential equation may have more than one solution, even infinitely many solutions. For example, the function

$$y = f(x) = ce^{2x},$$

where c is an arbitrary constant, is also a solution of the first order equation

$$y' = 2y,$$

as the reader may verify. It is customary to call such a function which contains an arbitrary constant a *general* solution of the corresponding differential equation of the first order. If a definite value is assigned to that constant, then the solution is called a *particular* solution. It also has to be mentioned that in some cases there may be further solutions of a given equation which cannot be obtained by assigning a definite value to the arbitrary constant in the general solution. Such a solution is called a *singular* solution. However, singular solutions rarely occur in engineering problems.

3.3 Higher order differential equations

Differential equations of the second order play a basic role in many different engineering problems. A function

$$y = g(x)$$

is said to be a solution of a differential equation of the second order on some interval, perhaps infinite, if $g(x)$ is defined and twice differentiable throughout that interval, and if it is such that the equation becomes an identity when we replace the unspecified function y and its derivatives in the equation by g and its derivatives. This definition is analogous to that in the case of a differential equation of the first order.

For example, the functions

$$y = cos\ x \qquad \text{and} \qquad y = sin\ x$$

are solutions of the differential equation

$$y'' + y = 0$$

for all x, since

$$\begin{array}{rcrcll} (cos\ x)'' + cos\ x & = & -cos\ x + cos\ x & = & 0 & \text{and} \\ (sin\ x)'' + sin\ x & = & -sin\ x + sin\ x & = & 0. & \end{array}$$

Differential equations of the second order can also be written in the implicit form

$$G(x, y, y', y'') = 0$$

and in many cases in the explicit form

$$y'' = g(x, y, y').$$

This also holds for third and higher order differential equations. If a second order differential equation does not contain the dependent variable y explicitly, it is of the form

$$G(x, y', y'') = 0.$$

By setting $y' = z$ this equation will be reduced to a first order equation in z,

$$G(x, z, z') = 0.$$

From a solution of z of this new equation, a solution of y of the original equation can be obtained by integration. It is also possible to reduce a second order differential equation, which does not involve the independent variable x explicitly, to a first order equation in $z = y'$, in which y is the independent variable.

Just like for the second order differential equations, all higher order differential equations can be rewritten as a system of multiple first order equations. It is convenient to write such a system in vector form. For example, the differential equation

$$\frac{d^3y}{dx^3} = f(x, y, \frac{dy}{dx}, \frac{d^2y}{dx^2})$$

is, by the substitution

$$\eta_1 = y, \qquad \eta_2 = \frac{dy}{dx}, \qquad \eta_3 = \frac{d^2y}{dx^2},$$

transformed into the system

$$\begin{aligned} \frac{d\eta_1}{dx} &= \eta_2, \\ \frac{d\eta_2}{dx} &= \eta_3, \\ \frac{d\eta_3}{dx} &= f(x, \eta_1, \eta_2, \eta_3). \end{aligned}$$

Here $\frac{d^n y}{dx^n}$ is used as notation, since for higher order differential equations the notation of y with n apostrophes will be less readable.

3.4 Approximation

There are a number of exact methods to solve differential equations. However, these methods will not be discussed in this thesis for two reasons. The first is the fact that many differential equations cannot be solved by an exact method. In these cases one has to apply numerical methods to yield an approximate numerical value of the solution. The other reason is the fact that it is the aim of this graduation project to use the differential equations directly in the code as a control law describing the dynamics of the models. When using an exact method the differential equation is often rewritten, or the dynamics which are described by it are expressed in a formula, which does not describe the relation between x, y and the derivatives of y, but the relation between x and y.

This thesis will discuss a number of various kinds of approximation methods for differential equations, which are mentioned often in the literature. In this way an idea is given of the possibilities of various kinds of methods which could be used for the approximation of differential equations and the implementation in the JAVA programming language. Many other methods are available, but it would be impossible to discuss all of these. However, after having read this thesis the reader should be able to implement any other desirable approximation method in the JAVA programming language, which could be used for approximating the differential equations describing the dynamics of the models discussed in this thesis, or any other models.

Chapter 4

Approximation methods

This chapter discusses the numerical approximation methods, which will be used for the approximation of the differential equations describing the motion of the models discussed throughout this thesis. The methods which it concerns are the Euler–Cauchy method, the Midpoint method, the fourth order Runge–Kutta method and the fourth order Runge–Kutta–Nyström method.

4.1 Used methods

To make use of differential equations in the code, which in this case will be written in the JAVA programming language, y and its derivatives have to be computed after a given time. As stated in chapter 3, some differential equations can be solved with an exact method, but quite often it is not possible to solve them in this way. In these cases numerical methods have to be applied to yield an approximate numerical value of the solution.

The approximation methods which are discussed in this thesis, and which are implemented during this graduation project to be used in the models, are the:

- Euler[1]–Cauchy[2] method [8, 10]
- Midpoint method [10]
- Fourth order Runge–Kutta[3] method [8, 10]
- Fourth order Runge–Kutta–Nyström[4] method [8]

In the next paragraphs these methods will be discussed. The examples used in these discussions are taken from [8] and [10].

[1] Leonhard Euler (1707–1783), Swiss mathematician
[2] Louis Cauchy (1789–1857), French mathematician
[3] Carl Runge (1856–1927) and Wilhelm Kutta (1867–1944), German mathematicians
[4] E.J. Nyström, Finnish mathematician

4.2 Euler–Cauchy method

The Euler–Cauchy method is a simple procedure. The differential equation, for example $y' = x + y$, has to be rewritten in the following way:

$$f(x, y) = x + y \qquad \text{where } y' = f(x, y).$$

Rewriting does not alter the differential equation, it is just another way of presenting the same equation, so it can still be used directly in the code.

With an initial state, x_0 and y_0, y_n can be approximated by the following formula:

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

where h is the stepsize used in the approximation. This stepsize depends on the desired accuracy of the approximate values. After every step taken, x will be increased with an amount of h. With y_{n+1} also an approximate value of y'_{n+1} is known, it is equal to

$$y' = f(x, y) = x + y.$$

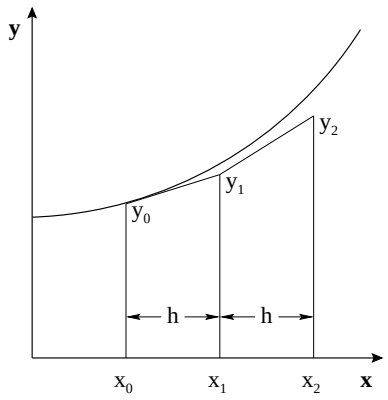

Figure 4.1: Euler–Cauchy method

The Euler–Cauchy method is illustrated in figure 4.1. The interval $[a, b]$ is divided into subintervals of length h, where $x_i = a + ih$. If y_0 is known, approximate values y_1, y_2, etcetera, can be found by approximating the derivative at the point (x_n, y_n) with the difference quotient

$$y' = \frac{y_{n+1} - y_n}{h} = f(x_n, y_n).$$

This leads to the formula, which was already mentioned above, by which the value of y_n can be approximated. The remaining approximation methods which are discussed in this chapter are not illustrated with a figure. They are all based on the same principle of omission of

the further terms in the infinite *Taylor series*[5] and using a certain stepsize. The only difference is that they are more complex than the Euler–Cauchy method.

4.3 Midpoint method

Unlike the Euler–Cauchy method the Midpoint method uses two auxiliary quantities instead of one. This results in a more accurate method. Here the differential equation, say for example $y' = y$, has to be rewritten in the following way:

$$f(x, y) = y \qquad \text{where } y' = f(x, y).$$

With an initial state, x_0 and y_0, y_n can be approximated by the following formula:

$$y_{n+1} = y_n + dy_2.$$

But first the auxiliary quantities

$$\begin{aligned} dy_1 &= hf(x_n, y_n), \\ dy_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}dy_1), \end{aligned}$$

have to be computed, where h is the stepsize used in the approximation. With y_{n+1} also an approximate value of y'_{n+1} is known, it is equal to

$$y' = f(x, y) = y.$$

4.4 Fourth order Runge–Kutta method

A still more accurate method is the fourth order Runge–Kutta method, which uses four auxiliary quantities. It is called a fourth-order method because it can be shown that the truncation error (see paragraph 4.7) of the method is of the order h^5 [11]. Here the differential equation, say for example $y' = x + y$, has to be rewritten in the following way:

$$f(x, y) = x + y \qquad \text{where } y' = f(x, y).$$

With an initial state, x_0 and y_0, y_n can be approximated by the following formula:

$$y_{n+1} = y_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}.$$

[5] Brook Taylor (1685–1731), English mathematician

But first the four auxiliary quantities

$$\begin{aligned} k_1 &= hf(x_n, y_n), \\ k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1), \\ k_3 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2), \\ k_4 &= hf(x_n + h, y_n + k_3), \end{aligned}$$

have to be computed, where h is the stepsize used in the approximation. With y_{n+1} also an approximate value of y'_{n+1} is known, it is equal to

$$y' = f(x, y) = x + y.$$

4.5 Fourth order Runge–Kutta–Nyström method

The fourth order Runge–Kutta–Nyström method is a generalization of the fourth order Runge–Kutta method. It also uses four auxiliary quantities, but the Runge–Kutta–Nyström method is an approximation method for second order differential equations, instead of differential equations of the first order. Usually, second and higher order differential equations are reduced to a system of multiple first order equations. As stated in chapter 3, this is done by substitution and transforming of the differential equation into a vector form. However, the Runge–Kutta–Nyström method provides a way to approximate second order differential equations, so one does not have to rewrite the equation in multiple first order equations. It is worthwhile to do some research on such a method as well, to be able to make a comparison between the use of a second order differential equation which is used directly in the code, and a second order differential equation which is rewritten in vector form and used in the code, for example.

When using the Runge–Kutta–Nyström method the differential equation, say for example $y'' = \frac{x+y+y'+2}{2}$, can be rewritten in the following way:

$$f(x, y, y') = \frac{x + y + y' + 2}{2} \qquad \text{where } y'' = f(x, y, y').$$

With an initial state, x_0, y_0 and y'_0, y_n and y'_n can be approximated by the following formulas:

$$\begin{aligned} y_{n+1} &= y_n + h(y'_n + \frac{k_1 + k_2 + k_3}{3}), \\ y'_{n+1} &= y'_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{3}. \end{aligned}$$

However, first the auxiliary quantities

$$\begin{aligned}
k_1 &= \frac{1}{2}hf(x_n, y_n, y'_n), \\
k_2 &= \frac{1}{2}hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}h(y'_n + \frac{1}{2}k_1), y'_n + k_1), \\
k_3 &= \frac{1}{2}hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}h(y'_n + \frac{1}{2}k_1), y'_n + k_2), \\
k_4 &= \frac{1}{2}hf(x_n + h, y_n + h(y'_n + k_3), y'_n + 2k_3),
\end{aligned}$$

have to be computed, where h is the stepsize used in the approximation. With y_{n+1} and y'_{n+1} also an approximate value of y''_{n+1} is known, it is equal to

$$y'' = f(x, y, y') = \frac{x + y + y' + 2}{2}.$$

4.6 Choosing a suitable approximation method

The Euler–Cauchy and the Midpoint method provide reasonable results in a short amount of time. They can best be used in a system in which the accuracy is not most important, or in a system in which a reasonable but, even more important, fast result is required.

The fourth order Runge–Kutta and Runge–Kutta–Nyström method both provide satisfactory results, but take a larger amount of time due to the four auxiliary quantities which have to be computed. The Runge–Kutta method is suitable for first order differential equations, the Runge–Kutta–Nyström method for second order differential equations. In case of a second order differential equation it is most convenient to use the Runge–Kutta–Nyström approximation method. In this way the differential equation can be used directly in the code, without rewriting it into its vector form. When it concerns a first order differential equation it is better to use the Runge–Kutta method.

When dealing with a third or higher order differential equation, in any case the equation has to be rewritten into its vector form. Furthermore, to be able to make a comparison in this graduation project between the Runge–Kutta–Nyström method and the other methods with a second order differential equation written in its vector form, any second order equation which may appear will also be approximated when it is written in its vector form.

4.7 Error in the approximation

The fourth order Runge–Kutta and Runge–Kutta–Nyström method are two of the standard algorithms to solve differential equations. They provide an excellent balance of power, precision and simplicity to program. There are two types of error involved in a step:

- round-off error
- truncation error

Round-off error is due to the floating-point arithmetic usually used when the method is implemented on a computer. It depends on the number and type of arithmetical operations used in a step, and on the computer on which the algorithm is implemented. Truncation error is present even with infinite-precision arithmetic, because it is caused by truncation of the infinite *Taylor series* to form the algorithm. It depends on the stepsize, the order of the method, and the problem being solved.

These same errors hold for the Euler–Cauchy and the Midpoint method. Except, the truncation error will be larger due to a sooner truncation of the infinite *Taylor series*.

Chapter 5

The models

This chapter will focus on the physical and the mathematical background of the models of the simple *real-time* systems, which are discussed throughout this thesis: the model of a pendulum, a bouncing ball and the model of a double pendulum.The physical background concerns the assumptions that are made and the relevant parameters in the model. The mathematical background concerns the derivation of the differential equations, which describe the motion of these models.

5.1 The pendulum

5.1.1 Physical background

The model discussed first is the pendulum model, with damping. It consists of a bob, a string and a point of attachment, in such a way that the pendulum can make a full circle (see figure 5.1). Damping refers to the net resistive force caused by a combination of factors, such as:

- air resistance
- friction in the attachment to the bob
- friction in the point of attachment of the string

In this model the following assumptions are made:

- The string is rigid, in other words the string is inextensible and straight.
- The string is massless, the mass of the string is negligible compared to the mass of the bob.
- Two assumption are made about damping:

- The effects of all sources of damping can be taken together and treated as a single force.
- The damping force is proportional to the speed of the bob.

The relevant parameters in the model of the damped pendulum are the length of the string, the mass of the bob and the damping constant, whose value represents the strength of the net resistive force. A large damping constant indicates large damping, while a damping constant of zero indicates no damping. In the latter case it concerns the model of what is known as the simple pendulum [12, 13].

5.1.2 Mathematical background

Since the motion of a pendulum is rotational, a polar coordinate system is chosen with the origin at the point where the string is attached. The polar axis is straight down and the positive angular direction is counterclockwise.

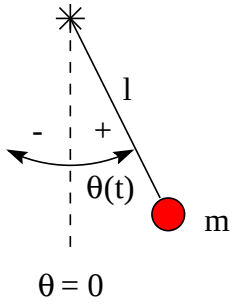

Figure 5.1: Diagram of the pendulum

The variables used are:

- θ: the angular displacement of the bob
- l: the length of the string
- m: the mass of the bob
- t: the time

As stated before it is assumed that the damping force is proportional to the speed of the bob. The speed of the bob is equal to $l\theta'$. If the damping constant is called γ, then the damping force is

$$-\gamma l\theta'.$$

By using *Newton's second law*[1],

[1] Isaac Newton (1642–1727), English scientist and mathematician

$$\vec{F} = m\vec{a},$$

the differential equation of the model of the damped pendulum can be derived. This law is a vector equation. For the pendulum it has two components. One in the tangential direction, tangential to the arc described by the motion of the bob. The other in the radial direction, perpendicular to the arc described by the motion of the bob. Since the bob does not move in the radial direction, because of the constant length of the string, there can be concentrated on the tangential component

$$F_\theta = ma_\theta.$$

The gravity pulls straight down, so not all of its force is felt in the tangential direction. That is why the tangential component of the gravity has to be used.

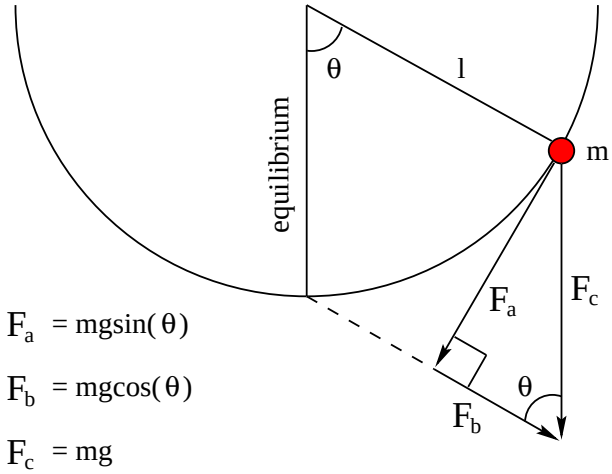

Figure 5.2: Tangential component of the gravity

In figure 5.2 it can be seen that the tangential component is equal to

$$F_\theta = -mgsin(\theta).$$

This force is negative due to its tendency to move the pendulum back towards the equilibrium position.

In the case of the damped pendulum, the tangential component is equal to $-mgsin(\theta)$ plus the damping force, so the tangential component of the model of the damped pendulum is

$$F_\theta = -\gamma l\theta' - mgsin(\theta).$$

To rewrite the tangential acceleration, a_θ, the distance traveled by the bob, when the pendulum moves through an angle θ, has to be known. In other words, the length of its arc has

to be known when the angle is known. Therefore, radian measure is used. The arclength is equal to

$$l\theta.$$

The acceleration is found by taking the second derivative of the arclength. This equals

$$l\theta''.$$

If the expressions for F_θ and a_θ are put into the formula $F_\theta = ma_\theta$, the following equation is obtained:

$$-\gamma l\theta' - mgsin(\theta) = ml\theta''.$$

By rewriting, the differential equation, which describes the motion of the model of a damped pendulum, is obtained.

$$\theta'' + \frac{\gamma}{m}\theta' + \frac{g}{l}sin(\theta) = 0.$$

To be complete, it is also necessary to specify the initial state. In this case an initial angle of displacement, say $\theta(0) = \theta_0$, and an initial angular velocity, say $\theta'(0) = v_0$, are needed.

For more information on the physical and mathematical background of the differential equation describing the motion of the damped pendulum see [14].

5.2 The bouncing ball

5.2.1 Physical background

The second model which is discussed is that of the bouncing ball, with damping. In this case damping only refers to the air resistance. The following assumptions are made in this model:

- At the moment of bouncing the ball does not deform.
- At the moment of bouncing there is a constant percentage of energy-loss.
- The ball has a contant velocity in the horizontal direction.
- The damping force is proportional to the square of the velocity of the ball.

The relevant parameters in the model of the bouncing ball are the mass of the ball and the damping constant, which only refers to the damping due to the air resistance. A large damping constant indicates large damping, while a damping constant of zero indicates no damping due to the air resistance. In all cases there will still be another kind of damping, namely the energy-loss due to a bounce of the ball.

5.2.2 Mathematical background

First a direction for the velocity has to be chosen. The upward direction will indicate a positive velocity and the downward direction a negative velocity. The differential equation describing the velocity of a ball moving in the downward or in the upward direction can be derived, in the same manner as for the pendulum model, by using *Newton's second law*,

$$\vec{F} = m\vec{a}.$$

In this case $\vec{F}$ is the resultant of the forces acting on the ball at any instant. These forces are a force in the downward direction and a force in the opposite direction of the direction in which the ball is moving. The latter of these forces is the force acting on the ball due to the air resistance.

The downward force is equal to $-mg$, where m is the mass of the ball and g is the gravitational acceleration. The force acting on the ball due to the air resistance is equal to $-kwv^2$, where k is the damping constant and w indicates the direction of the damping. If the damping direction is upward w is equal to -1, if it is downward w is equal to 1.

This leads to the following formula:

$$m\vec{a} = -mg - kwv^2.$$

Since the acceleration only acts in the vertical direction, $\vec{a}$ can simply be replaced by a. The acceleration is the derivative of the velocity, so $a = v'$.

Together with *Newton's second law* this leads to the following formula:

$$mv' = -mg - kwv^2.$$

By rewriting, the differential equation, which describes the velocity of a ball moving in the downward or the upward direction, can be derived.

$$v' = -g - \frac{k}{m}wv^2.$$

To describe the motion of a ball moving in the downward or in the upward direction another differential equation, describing the height of the ball, has to be added to the equation describing the velocity of the ball. This is a very simple differential equation, which is equal to

$$y' = v.$$

The system formed by these two differential equations describes the motion of a ball moving in the downward or in the upward direction. To be complete, it is also necessary to specify the

initial state. In this case an initial height, say $y(0) = y_0$, and an initial velocity, say $v(0) = v_0$, are needed.

For more information on the physical and mathematical background of the differential equations describing the motion of a falling object see [9].

5.3 The double pendulum

With the double pendulum, which is discussed throughout this thesis, a different model is meant than the one which is mentioned at various places in the literature. In this literature the system in figure 5.3 (a) is meant with a double pendulum. However, in this thesis a system of two pendulums, which have the same length and are attached to the same point, is meant. These pendulums both move independent of eachother, except when there is a collision of the pendulums. This model is depicted in figure 5.3 (b).

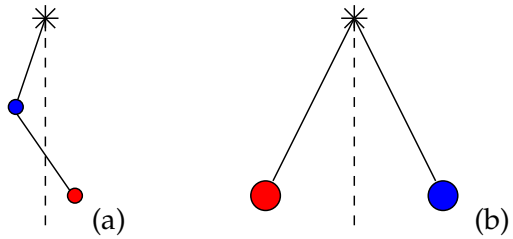

Figure 5.3: The double pendulum

5.3.1 Differential equations

Since for both pendulums in the model discussed in this thesis the same physical and mathematical background holds as for the single damped pendulum, the motion of both pendulums can be described by the same differential equation as for the single damped pendulum:

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

with an initial angle of displacement and an initial angular velocity.

The only difference is the effect on both pendulums in case of a collision. In this case two principles hold:

- the principle of conservation of momentum [15, 16],
- and the principle of conservation of energy [15, 16].

It is assumed that at the moment of a collision of the bobs of both pendulums, the angle between the vectors of the moving bobs is equal to 180 degrees if the pendulums move in opposite direction, and is equal to 0 degrees if the pendulums move in the same direction.

5.3.2 Principles of conservation

The principle of conservation of momentum says that the total momentum of an isolated system is constant. This means that the total momentum of the system at time t,

$$P = p_1 + p_2 = m_1 v_1 + m_2 v_2,$$

is equal to the total momentum of the system at a later time t',

$$P' = p'_1 + p'_2 = m_1 v'_1 + m_2 v'_2.$$

Thus, $P = P'$, and therefore

$$m_1 v_1 + m_2 v_2 = m_1 v'_1 + m_2 v'_2,$$

where v_1 and v_2 are the velocities of, respectively, pendulum one and pendulum two before the collision, and v'_1 and v'_2 are the velocities of those after the collision.

The other principle is that of conservation of energy. This principle says that the total amount of energy of an isolated system is constant. This means that the total amount of energy of the system at time t,

$$U = E_k + E_p = \frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 + E_p,$$

is equal to the total amount of energy of the system at a later time t',

$$U' = E'_k + E'_p = \frac{1}{2} m_1 (v'_1)^2 + \frac{1}{2} m_2 (v'_2)^2 + E'_p.$$

Here E_k indicates the kinetic energy of the system and E_p indicates the potential energy of the system. Since the position of both bobs just before the collision can be seen as equal to the position of both bobs just after the collision, the potential energy just before the collision is equal to the potential energy just after the collision, so $E_p = E'_p$.

Knowing this, the equation can be written as

$$\frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 + E_p = \frac{1}{2} m_1 (v'_1)^2 + \frac{1}{2} m_2 (v'_2)^2 + E_p,$$

where v_1 and v_2 are the velocities of, respectively, pendulum one and pendulum two before the collision, and v'_1 and v'_2 are the velocities of those after the collision.

5.3.3 Solution of the principles of conservation

The principles of conservation of momentum and of conservation of energy have to be programmed in the model of the double pendulum. It is possible to use the formulas, which are found in the previous paragraph, directly in the code and calculate the solution. However, this will take quite a large amount of time and will lead to two solutions of which the right one has to be chosen. Since the collision concerns an event, a discrete moment in time, the adaptation of the system has to be done as fast as possible, so the real world will be approximated as good as possible. Therefore, the formulas of the principles of conservation of momentum and of conservation of energy are rewritten into one general solution for a collision of the system.

From the principle of conservation of momentum it follows that

$$m_1 v_1 + m_2 v_2 - m_2 v_2' = m_1 v_1',$$

and thus

$$v_1' = \frac{m_1 v_1 + m_2 v_2 - m_2 v_2'}{m_1}. \tag{5.1}$$

From the principle of conservation of energy it follows that

$$m_1 v_1^2 + m_2 v_2^2 - m_1 (v_1')^2 = m_2 (v_2')^2,$$

which leads to

$$\frac{m_1 v_1^2 + m_2 v_2^2 - m_1 (v_1')^2}{m_2} = (v_2')^2,$$

and thus leads to

$$v_2' = \sqrt{\frac{m_1}{m_2} v_1^2 + v_2^2 - \frac{m_1}{m_2} (v_1')^2}. \tag{5.2}$$

By substituting v_2' in equation 5.1 by equation 5.2, the formula for v_1', which represents the velocity of pendulum one after the collision, can be found after numerous steps. These steps will not be shown in this thesis, the given result can be checked by the interested reader.

The formula, which results from this substitution and which has two solutions, is

$$v_1' = \frac{v_1 + \frac{m_2}{m_1} v_2 \pm \frac{m_2}{m_1} \sqrt{v_2^2 + v_1^2 - 2 v_1 v_2}}{1 + \frac{m_2}{m_1}}. \tag{5.3}$$

To find the formula of v_2', which represents the velocity of pendulum two after the collision, the principle of conservation of momentum has to be rewritten into

$$m_1 v_1 + m_2 v_2 - m_1 v_1' = m_2 v_2',$$

which leads to

$$v_2' = \frac{m_1 v_1 + m_2 v_2 - m_1 v_1'}{m_2}.$$

By substitution of v_1' by equation 5.3 the formula, which represents the velocity of pendulum two after the collision, can be found after numerous steps. These steps will not be shown in this thesis, it is up to the reader to check the given result, which also has two solutions:

$$v_2' = \frac{v_1 + \frac{m_2}{m_1} v_2 \pm \sqrt{v_2^2 + v_1^2 - 2 v_1 v_2}}{1 + \frac{m_2}{m_1}}. \tag{5.4}$$

To come to a single solution the term $\sqrt{v_2^2 + v_1^2 - 2 v_1 v_2}$ of equation 5.3 and equation 5.4 has to be rewritten into

$$\sqrt{(v_2 - v_1)(v_2 - v_1)} \qquad \text{or} \qquad \sqrt{(v_1 - v_2)(v_1 - v_2)},$$

which is equal to, respectively,

$$v_2 - v_1 \qquad \text{or} \qquad v_1 - v_2.$$

By substituting these terms in equation 5.3 and in equation 5.4 the following four formulas, which all have two results, are obtained:

$$v_1' = \frac{v_1 + \frac{m_2}{m_1} v_2 \pm \frac{m_2}{m_1}(v_2 - v_1)}{1 + \frac{m_2}{m_1}} \tag{5.5}$$

$$v_1' = \frac{v_1 + \frac{m_2}{m_1} v_2 \pm \frac{m_2}{m_1}(v_1 - v_2)}{1 + \frac{m_2}{m_1}} \tag{5.6}$$

$$v_2' = \frac{v_1 + \frac{m_2}{m_1} v_2 \pm (v_2 - v_1)}{1 + \frac{m_2}{m_1}} \tag{5.7}$$

$$v_2' = \frac{v_1 + \frac{m_2}{m_1} v_2 \pm (v_1 - v_2)}{1 + \frac{m_2}{m_1}} \tag{5.8}$$

By evaluating the two result of equation 5.5 and the two results of equation 5.6, it is found that both equations lead to the same two results, namely

$$v_1' = \frac{v_1 + 2\frac{m_2}{m_1} v_2 - \frac{m_2}{m_1} v_1}{1 + \frac{m_2}{m_1}} \qquad \text{and} \tag{5.9}$$

$$v_1' = \frac{v_1 + \frac{m_2}{m_1} v_1}{1 + \frac{m_2}{m_1}} = v_1. \tag{5.10}$$

By evaluating the two result of equation 5.7 and the two results of equation 5.8, it can also be found that both equations lead to the same two results, namely

$$v_2' = \frac{v_1 + \frac{m_2}{m_1}v_2 - v_2 + v_1}{1 + \frac{m_2}{m_1}} \quad \text{and} \tag{5.11}$$

$$v_2' = \frac{v_1 + \frac{m_2}{m_1}v_2 + v_2 - v_1}{1 + \frac{m_2}{m_1}} = v_2. \tag{5.12}$$

The equations 5.10 and 5.12 lead to the solution of the velocity before the collision of, respectively, pendulum one and pendulum two. When equation 5.9, which is a solution of the velocity of pendulum one after the collision, is rewritten, it equals

$$v_1' = \frac{(1 - \frac{m_2}{m_1})v_1 + 2\frac{m_2}{m_1}v_2}{1 + \frac{m_2}{m_1}}. \tag{5.13}$$

For equation 5.11 the same can be done. This leads to the following formula, which is a solution of the velocity of pendulum two after the collision:

$$v_2' = \frac{2v_1 + (\frac{m_2}{m_1} - 1)v_2}{1 + \frac{m_2}{m_1}}. \tag{5.14}$$

To summarize, the single solution of a collision of the system of the double pendulum is as follows: the velocity of pendulum one after the collision is equal to equation 5.13, and the velocity of pendulum two after the collision is equal to equation 5.14. These equations satisfy both conservation principles and can be used directly in the code. In this way the new velocities can be calculated at the collision event as fast as possible.

Chapter 6

Simulation in OmSim

This chapter discusses the simulation, in the simulationtool OMSIM [17, 18], of the model of the pendulum, the bouncing ball and the model of the double pendulum. It will focus on the behaviour of these models, especially on the behaviour when a discontinuity occurs as a result of an internal or an external event.

6.1 The OmSim simulationtool

OMSIM is an interactive environment for defining and simulating dynamical models based on the modeling language OMOLA [17, 18, 19]. OMOLA combines a mathematical and logical behaviour representation with the powerful structuring mechanisms of object-oriented programming languages. The main advantage of this object-oriented modeling methodology is that it supports reuse of models in several different ways [17].

OMOLA and OMSIM are particularly useful in the modeling and simulation of hybrid systems. These systems combine discrete and continuous dynamics. They involve a combination of automata theory and differential equations. A typical hybrid system is a thermostat keeping the temperature in a room close to the goal temperature that can be set by the user. There are different control laws describing the temperature in the room as a function of time, depending on whether the heater is switched on or off. If the temperature rises above the goal temperature then the heater will be switched off, and if the temperature falls below the goal, then the heater will be switched on. These discontinuities in the control law through internal actions are based on internal pre-programmed decisions. Further, the user can set a new goal temperature, causing a discontinuity as a result of an external action [1].

The models discussed in this thesis are also typical hybrid systems, just as the thermostat. The motion of these systems as a function of time can be described by differential equations. It is also possible that a discontinuity will occur as a result of an internal or an external action.

OMSIM provides a number of approximation methods to solve differential equations, which describe the dynamics of a system. Three methods are provided to solve ordinary differential equations (ODEs). These are the Euler-Cauchy method (see paragraph 4.2) and two Runge-

Kutta methods (see paragraph 4.4). In OMSIM they are called, respectively, *Euler*, *Dopri45r* and *RKsuit*. The idea behind these Runge-Kutta methods and the one discussed in paragraph 4.4 is the same, what the exact difference is between these methods is not explained in this thesis. To solve differential algebraic equations (DAEs), which also include partial differential equations, two methods are provided. These are *Radau5* [20], a Runge-Kutta method for differential algebraic equations (see paragraph 4.4), and the *Dasrt* method, a variant of the DAE solver *Dassl* [21], which has a builtin root solver. These approximation methods will not be explained in this thesis.

6.2 The pendulum

To simulate the pendulum system an OMOLA model library, containing a model of the pendulum, was written. When this model is being used the damped pendulum can be simulated. By setting the damping constant parameter of the damped pendulum to zero it is also possible to simulate an undamped pendulum.

In the model of the damped pendulum the corresponding differential equation is used:

$$y'' + \frac{c}{m}y' + \frac{g}{l}sin(y) = 0.$$

The initial values and parameters in the model which have to be set by the user are:

- $y(0)$: the angle of displacement, with unit rad
- $y'(0)$: the angular velocity, with unit $\frac{rad}{s}$
- $y''(0)$: the angular acceleration, with unit $\frac{rad}{s^2}$
- t: the time a tap is given on the pendulum, with unit s
- v: the amount of change in the angular velocity when a tap is given on the pendulum, with unit $\frac{rad}{s}$

Other parameters used in the model which have a default value are:

- g: the acceleration due to the gravity, which has a default value of $9.8\ \frac{m}{s^2}$
- l: the length of the pendulum, which has a default value of $0.5\ m$
- m: the mass of the bob, which has a default value of $0.4\ kg$
- c: the damping constant, equal to γ, which has a default value of $0.1\ \frac{kg}{s}$

These parameters can also be set by the user.

There are four events defined in the model: Tap, Init, InitCond and Stop. The Tap event represents the moment that a tap is given on the pendulum, causing a discontinuity in the motion of the pendulum. This is a typical external event, since it is caused by an action from outside the system. Init is a special event, since it will always be called by the simulator at the start of a new simulation. The model makes use of that, by scheduling the event Tap to occur in t time units after the Init event. So in the model, the pendulum can only be given a tap after a predefined time. The other two events InitCond and Stop are used to check the initial values of the parameters. If the user has set an initial condition which is not permitted, the simulation is terminated.

```
ONEVENT Tap DO
  new(y') := y' + v;
END;
```

It has to be noticed that the effect of a tap on the pendulum is a change in the angular velocity of the pendulum with a predefined amount of v. This is due to the change of momentum. The momentum is equal to the product of the bob's mass and velocity. By a change of momentum the velocity of the bob will change, because the mass of the bob has a constant value. Because the tap concerns an event, the physical assumption is made that the change of momentum and thus the change of velocity is instantaneous. A more detailled discussion of the OMOLA model is given in paragraph 8.3.

When running the simulation of the damped pendulum in the OMSIM-tool using the *RKsuit* approximation method with the following initial values and parameters:

$$\begin{array}{lcllcl} y(0) & = & 2.5\ rad & t & = & 18.0\ s \\ y'(0) & = & 0.0\ \frac{rad}{s} & v & = & 8.5\ \frac{rad}{s} \\ y''(0) & = & 0.0\ \frac{rad}{s^2} & & & \end{array}$$

and for g, l, m and c the default value, the graph in figure 6.1 can be obtained for y, which represents the angle of displacement. It is also possible to obtain graphs for y' and y'', which represent the angular velocity and the angular acceleration, respectively. These graphs are not included in this thesis.

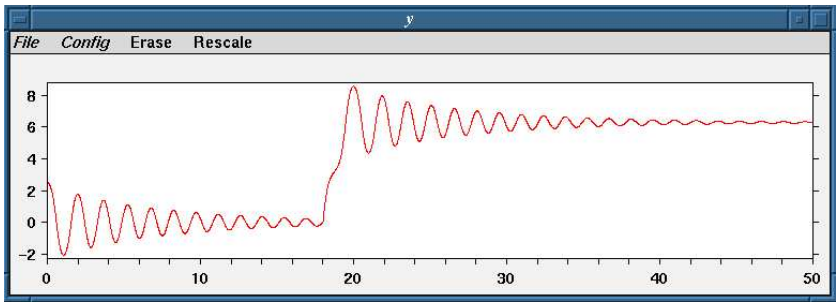

Figure 6.1: Angle of displacement of the damped pendulum (OMSIM)

The graph in figure 6.1 shows an angle of displacement which decreases when proceeding in time. This is due to the damping which occurs on the pendulum. After 18 seconds a tap is

given on the pendulum. The tap increases the angular velocity with an amount of $8.5\ \frac{rad}{s}$ as explained before. In this case this results in an increasing angle of displacement. Because the change of the angular velocity is large enough, the angle of displacement exceeds $\pi\ rad$, or 180 degrees, which means the pendulum makes a full circle. This can be seen in the graph in figure 6.1 by the angle of displacement which increases to an equilibrium of $2\pi\ rad$. Of course such a tap can also be given on an undamped pendulum, of which the damping constant is zero, but this results in an unlimited number of full circles of the pendulum, because of the lack of damping. This comes down to an ever increasing or decreasing graph, since with every full circle the graph increases or decreases with $2\pi\ rad$.

6.3 The bouncing ball

In the OMOLA model library of the bouncing ball system, the two differential equations, which describe the dynamics of the system, are used:

$$\begin{aligned} y' &= v, \\ v' &= -g - \frac{k}{m} w v^2. \end{aligned}$$

The initial values and parameters in the model which have to be set by the user are:

- $y(0)$: the height, with unit m
- $y'(0) = v(0)$: the velocity, with unit $\frac{m}{s}$
- $y''(0) = v'(0)$: the acceleration, with unit $\frac{m}{s^2}$

Other parameters used in the model which have a default value are:

- g: the acceleration due to the gravity, which has a default value of $9.8\ \frac{m}{s^2}$
- k: the damping constant, which has a default value of $0.01\ \frac{kg}{s}$
- m: the mass of the ball, which has a default value of $0.1\ kg$
- e: the percentage of energy-loss due to a bounce, which has a default value of $10\ \%$

These parameters can also be set by the user.

In this model there are four events defined: Init, InitCond, Stop and Bounce. For the Init, InitCond and Stop event the same holds as for these events in the model of the pendulum. The Bounce event defines the action which should be undertaken at the moment of a bounce of the ball on the floor. This is a typical internal event, since there is no action from outside the system which is causing the change of behaviour. To detect the moment of a bounce of the ball, the following event-condition is used in the OMOLA model:

```
ONEVENT y <= 0 AND v < 0 DO
 IF abs(v) < 0.1 THEN schedule(Stop, 0) ELSE schedule(Bounce, 0);
END;
```

At the moment of a bounce of the ball the height of the ball has to be equal to zero, but since y, which represents the height, is a continuous variable there cannot be a check on equality. Therefore, it is necessary to check on the condition $y <= 0$. Further, when the bouncing ball approaches the floor, and thus is moving in the downward direction, the velocity has a negative value. When both these conditions hold a bounce of the ball occurs. At the moment of a bounce a new velocity of the ball is computed. This new velocity has a positive value, since the ball has to move in the upward direction after a bounce. After a bounce of the ball the event-condition for a bounce to occur does not hold anymore, so it is not possible the event is fired an infinite number of times.

Another way to prevent the event to be fired an infinite number of times is to schedule the Stop event when the bouncing of the ball is almost finished. This is achieved by checking if the velocity of the ball is almost zero, when the ball bounces on the floor.

Further, it has to be noticed that the new velocity, which is computed at the moment of a bounce, is multiplied by $(1-e)$, in which e is the percentage of energy-loss due to the bounce. In this case it is assumed that 10 % of loss of energy results in 10 % of loss of velocity. When this assumption is made, the deviation will not be very large for a small value of the velocity. Finally, it has to be noticed that in this OMSIM model of the bouncing ball the ball can only move in the vertical direction, in contrast to the implementation of the model in JAVA, in which, besides the motion in the vertical direction, a constant velocity of the ball in the horizontal direction is used.

When running the simulation of the bouncing ball in the OMSIM-tool using the *Radau5* approximation method with the following initial values:

$$\begin{array}{lllll} y(0) & = & & & 1.5\ m \\ y'(0) & = & v(0) & = & 0.0\ \frac{m}{s} \\ y''(0) & = & v'(0) & = & -9.8\ \frac{m}{s^2} \end{array}$$

and for g, k, m and e the default value, the graph in figure 6.2 can be obtained for y, which represents the height of the ball. The acceleration of the ball starts at a value of $-9.8\ \frac{m}{s^2}$, since at the beginning it is equal to the acceleration due to the gravity. At the end of the simulation the acceleration that is experienced by the ball reaches an equilibrium of $-9.8\ \frac{m}{s^2}$, since every object on earth always experiences this gravitational acceleration. Although the acceleration still acts on the ball it is not moving anymore, which is due to the reaction force of the floor that is exerted on the ball. This situation is described by *Newton's third law*:

$$action = -reaction.$$

It is also possible to obtain graphs for y', which is equal to the velocity v, and for y'', which is equal to the acceleration v'. These graphs are not included in this thesis.

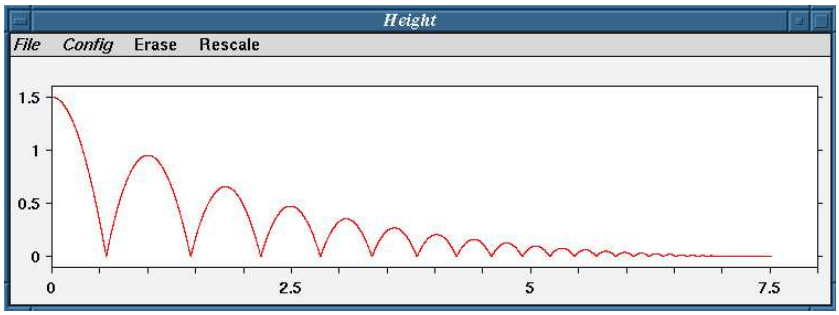

Figure 6.2: Height of the bouncing ball (OMSIM)

The graph in figure 6.2 shows the height of the ball which decreases when proceeding in time. There are two causes for this. One is the energy-loss due to a bounce of the ball, the other is the damping which occurs due to the air resistance. When the percentage of energy-loss and the damping constant are set to zero, no damping of any kind occurs on the ball and thus the ball will always bounce to the same height, provided that the approximation method is accurate enough.

6.4 The double pendulum

The OMOLA model library, which is used to simulate the double pendulum system, exhibits much similarity with the library of the single damped pendulum. In this model both pendulums make use of the differential equation which describes the motion of a single pendulum:

$$y_n'' + \frac{c_n}{m_n} y_n' + \frac{g}{l} sin(y_n) = 0 \qquad \text{where} \qquad n \in \{1, 2\}.$$

Also both pendulums make use of the corresponding initial values and parameters which have to be set by the user:

- $y_n(0)$: the angle of displacement of pendulum n, with unit rad
- $y_n'(0)$: the angular velocity of pendulum n, with unit $\frac{rad}{s}$
- $y_n''(0)$: the angular acceleration of pendulum n, with unit $\frac{rad}{s^2}$
- t_n: the time a tap is given on pendulum n, with unit s
- v_n: the amount of change in the angular velocity of pendulum n, when a tap is given on this pendulum, with unit $\frac{rad}{s}$

Other parameters used in the model which have a default value are:

- m_n: the mass of the bob of pendulum n, which has a default value of $0.4\ kg$

- c_n: the damping constant of pendulum n, which has a default value of $0.1\ \frac{kg}{s}$
- l: the length of both pendulums, which has a default value of $0.5\ m$
- g: the acceleration due to the gravity, which is equal for both pendulums, and which has a default value of $9.8\ \frac{m}{s^2}$

These parameters can also be set by the user. Further, there are two constants defined in the model. The first is *SmallestAngle*, which represents the smallest angle possible between the two pendulums, and which is used in the detection of a collision of the pendulums. The second is *PI*, which is equal to π and which is used in the detection of a number of events.

In this model there are seven events defined: Tap1, Tap2, BothTaps, Collision, Init, InitCond and Stop. The events Tap1 and Tap2 are defined in the same way as in the model of the single pendulum. The BothTaps event schedules Tap1 and Tap2 at the beginning of a simulation. The Collision event calculates the new velocities of both pendulums at the moment of a collision of these pendulums. The Init, InitCond and Stop event have the same purpose as in both other models.

In the OMSIM model of the double pendulum, the angle of displacement of both pendulums is limited to a range of $[-\pi, \pi]rad$. This provides a more convenient way to model the system, and it reduces the inaccuracy which occurs when a pendulum makes a full circle. Therefore, when one of the pendulums exceeds the limit of the angle of displacement of $\pi\ rad$, it is corrected by subtraction of $2\pi\ rad$. On the other hand, when one of the pendulums exceeds the limit of the angle of displacement of $-\pi\ rad$, it is corrected by addition of $2\pi\ rad$.

At first sight, the event-condition which holds for a collision is equal to

$$abs(y_1 - y_2) < SmallestAngle.$$

However, using only this condition can lead to an infinite number of collision events, since it is possible that after the calculation of the new velocities this condition still holds. To extend the condition it is necessary to determine other event-conditions which hold for a collision. Three possible cases have to be considered:

Case 1:	$y'_1 > 0$ and $y'_2 > 0$,			see figure 6.3 (a) and (b).
Case 2:	$y'_1 < 0$ and $y'_2 < 0$,			see figure 6.4 (a) and (b).
Case 3:	$y'_1 > 0$ and $y'_2 < 0$	or	$y'_1 < 0$ and $y'_2 > 0$,	see figure 6.5 (a) and (b).

In these cases y'_n equals the velocity of pendulum n. It is also possible that one of the velocities in a condition equals zero, but to preserve the readability of the three cases this is not written down here. As will turn out later on, this is of no importance, since it is only the difference between y'_1 and y'_2 that matters.

When considering the first case, just before the collision, two sets of conditions can be derived from figure 6.3. The first set, following from figure 6.3 (a) contains the conditions:

$$y'_1 - y'_2 > 0 \qquad \text{and} \qquad y_2 > y_1,$$

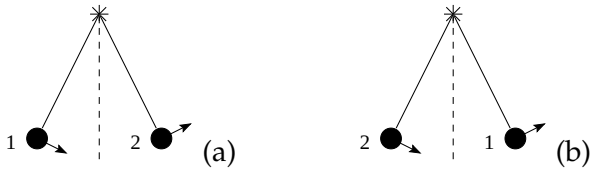

Figure 6.3: First case of a collision of the double pendulum

whereas figure 6.3 (b) leads to the second set of conditions:

$$y_2' - y_1' > 0 \qquad \text{and} \qquad y_1 > y_2.$$

These two conditions taken together lead to one condition, which holds just before a collision of the first case of the double pendulum, and which does not hold just after the collision:

$$(y_1 - y_2)(y_2' - y_1') > 0.$$

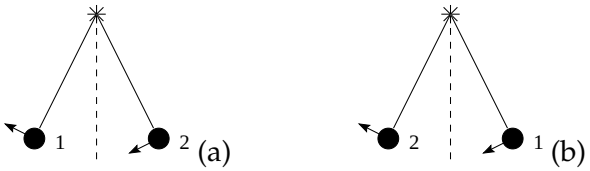

Figure 6.4: Second case of a collision of the double pendulum

Figure 6.4 relates to the moment just before a collision of the second case. The conditions following from figure 6.4 (a) are:

$$y_2' - y_1' < 0 \qquad \text{and} \qquad y_1 < y_2,$$

and figure 6.4 (b) leads to the conditions:

$$y_1' - y_2' < 0 \qquad \text{and} \qquad y_2 < y_1.$$

These two conditions taken together also lead to one condition, which holds just before a collision of the second case of the double pendulum, and which does not hold just after the collision:

$$(y_1 - y_2)(y_2' - y_1') > 0.$$

This condition appears to be the same as the condition of the collision of the first case.

Finally, when considering the moment before a collision of the third case, from figure 6.5 (a) the following conditions can be derived:

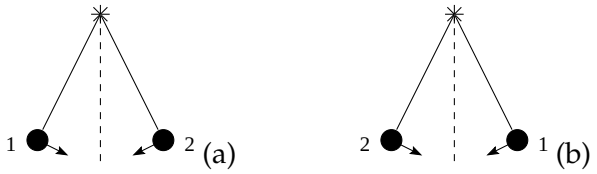

Figure 6.5: Third case of a collision of the double pendulum

$$y_1' - y_2' > 0 \qquad \text{and} \qquad y_2 > y_1,$$

and the condition following from figure 6.5 (b) are:

$$y_2' - y_1' > 0 \qquad \text{and} \qquad y_1 > y_2.$$

Taken together, these conditions again lead to one general condition, which is the same as for the first and the second case:

$$(y_1 - y_2)(y_2' - y_1') > 0.$$

Since the same general condition holds for all three cases, it can be used, together with the condition $abs(y_1 - y_2) < SmallestAngle$, to detect the moment of a collision of the double pendulum.

Because of the limited range of the angle of displacement of $[-\pi, \pi]\ rad$, a collision of the double pendulum at the top has another event-condition, which is equal to

$$abs(y_1 - y_2) > 2\pi - SmallestAngle \qquad \text{and} \qquad (y_1 - y_2)(y_1' - y_2') > 0.$$

This condition holds just before a collision at the top, and does not hold just after such a collision. It is derived in the same manner as for a collision which takes place at another position than at the top, but this is left to the interested reader.

Although the two conditions mentioned above ensure the detection of a collision, it has to be noticed that it is possible a collision will be missed, especially at a higher angular velocity, when the simulation is run in the OMSIM simulation tool. This is due to the stepsize which is being used in the approximation methods. When the event-condition of a collision holds just between two moments of approximation of the position of both pendulums, and not at these moments, a detection of the collision will be missed. When decreasing the stepsize the chances for this to happen, become smaller.

When the simulation of the double pendulum is run in the OMSIM-tool using the *Dasrt* approximation method with the following initial values and parameters:

$$y_1(0) = -2.0\ rad, \qquad m_1 = 0.5\ kg,$$
$$y_2(0) = 2.0\ rad, \qquad m_2 = 0.3\ kg,$$

$y_n'(0)$, $y_n''(0)$, t_n and v_n, in which $n \in \{1, 2\}$, all equal to zero, and for the other values, c_n, g and l the default value, the graph in figure 6.6 can be obtained for y_n, which represents the angle of displacement of both pendulums. The red line represents pendulum one, the blue line pendulum two. Only the first 15 seconds of the simulation are depicted, since this enables the events of interest to be clearly visualized in the graph. In this simulation it is also possible to obtain graphs for y_n' and y_n'', which represent the angular velocity and the angular acceleration of both pendulums, respectively. These graphs are not included in this thesis.

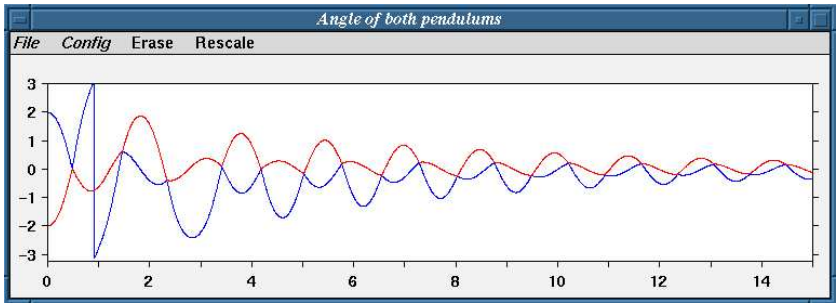

Figure 6.6: The double pendulum with different masses (OMSIM)

In figure 6.6 the collisions of the pendulums can be clearly noticed. After the moment of the first collision it can be seen that the displacement of the second pendulum becomes larger than the displacement of the first pendulum, and even larger than it was at the beginning of the simulation. This can be explained by the larger mass of pendulum one, and therefore the larger momentum, which is partly gained by pendulum two at the moment of collision. This results in a higher angular velocity as well.

The displacement of pendulum two even becomes that large that it makes a full circle. In the graph this can be noticed by the value of the displacement of pendulum two, which jumps from $\pi\ rad$ to $-\pi\ rad$. This range of $[-\pi, \pi]\ rad$ was defined, as discussed before, because of the convenience of modeling and to reduce the inaccuracy.

When the simulation is run with another set of values and parameters:

$$y_1(0) = -3.0\ rad, \quad t_1 = 0.0\ s, \quad v_1 = 0.0\ \tfrac{rad}{s},$$
$$y_2(0) = 1.0\ rad, \quad t_2 = 13.0\ s, \quad v_2 = 10.0\ \tfrac{rad}{s},$$

$y_n'(0)$, $y_n''(0)$, in which $n \in \{1, 2\}$, all equal to zero, and for the other values, m_n, c_n, g and l the default value, the graph in figure 6.7 can be obtained for y_n. The red line represents pendulum one, the blue line pendulum two. In this graph only the first 30 seconds are included, for the same reason as for the graph in figure 6.6. Graphs for y_n' and y_n'' can also be obtained, but are not included in this thesis.

The graph in figure 6.7 shows the angle of displacement of the double pendulum, when both masses are equal. Since the initial displacement of pendulum one to the left is larger than that of pendulum two to the right, the pendulums collide in such a way that they alternately get

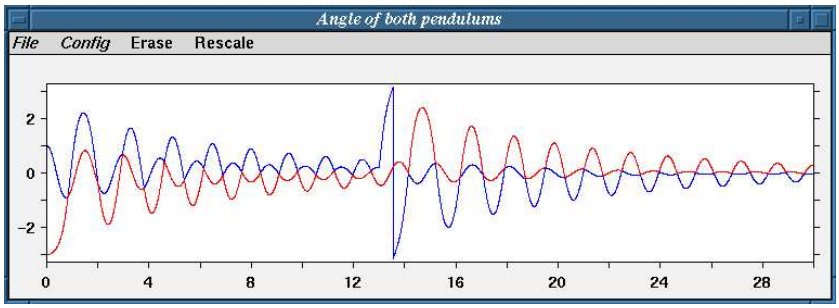

Figure 6.7: The double pendulum with a tap (OMSIM)

the larger (abolute value of) displacement. In the beginning of the simulation neither pendulum makes a full circle, because the angular velocity simply does not increase sufficiently after the occurrence of a collision.

At time 13 seconds a tap is given on pendulum two, in such a way that its angular velocity increases with $10\ \frac{rad}{s}$. This can be seen in the graph by the sudden increase of the angle of pendulum two. The angle even becomes that large, that the pendulum makes a full circle. At this moment the graph jumps from $\pi\ rad$ to $-\pi\ rad$, as was also mentioned in the text belonging to figure 6.6.

Chapter 7

The approximation methods in Java

This chapter discusses the JAVA code, which provides an implementation of the approximation methods mentioned in chapter 4. It will consider the implementation of the separate approximation methods, and the structure of the classes used for the approximation methods in its entirety. Further, this chapter will focus briefly on the rest of the JAVA code of the models.

7.1 The separate approximation methods

The implementation in JAVA code of the approximation of a differential equation exhibits a certain resemblance with the discussed theory in chapter 3 and 4. The auxiliary quantities, which are discussed in the theory, as well as the differential equation itself, can be found again in the JAVA code of all approximation methods.

The programming of these approximation methods is relatively simple. In a loop the auxiliary quantities on a given time can be computed. In this computation the differential equation is used. From these auxiliary quantities the new value of y can be computed. It is also possible to compute the higher derivatives of y from these auxiliary quantities. At the end of a computation the time is increased with the stepsize h.

As an example a part of the JAVA code for the Runge–Kutta approximation method, discussed in paragraph 4.4, is given:

```
// differential equation
public double diffeq(double x, double y) {
  return x + y;
}

while (x < x_next) {
  k1 = h * diffeq(x, y);
  k2 = h * diffeq(x + 0.5 * h, y + 0.5 * k1);
```

```
  k3 = h * diffeq(x + 0.5 * h, y + 0.5 * k2);
  k4 = h * diffeq(x + h, y + k3);

  y += (k1 + 2.0 * k2 + 2.0 * k3 + k4) / 6.0;
  x += h;
}
```

This code example is an implementation of the example, which was discussed in paragraph 4.4. The resemblance between the theory and this implementation should be clear. The other approximation methods also exhibit this kind of resemblance.

It is also possible to implement these approximation methods in such a way, that higher order differential equations can be solved by them. This can be accomplished by the use of arrays. When this is done, the loop for the Runge–Kutta approximation method will look like the following:

```
while (x < x_next) {
  for (int i = 0; i < n; i++) {
    k1[i] = h * diffeq(x, y, i);
    t1[i] = y[i] + 0.5 * k1[i];
  }
  for (int i = 0; i < n; i++) {
    k2[i] = h * diffeq(x + 0.5 * h, t1, i);
    t2[i] = y[i] + 0.5 * k2[i];
  }
  for (int i = 0; i < n; i++) {
    k3[i] = h * diffeq(x + 0.5 * h, t2, i);
    t3[i] = y[i] + k3[i];
  }
  for (int i = 0; i < n; i++) {
    k4[i] = h * diffeq(x + h, t3, i);
  }
  for (int i = 0; i < n; i++) {
    y[i] += (k1[i] + 2.0 * k2[i] + 2.0 * k3[i] + k4[i]) / 6.0;
  }
  x += h;
}
```

In this loop, n is the number of first order differential equations, into which the higher order differential equation has to be reduced, according to the theory discusssed in paragraph 3.3. The integer i is used as an index of the differential equations of the system. This code makes use of the same auxiliary quantities as the code discussed above, except that the computation in this case is carried out on all first order differential equations, into which the higher order equation was reduced. In this way higher order differential equations can also be approximated, when an implementation of first order approximation methods in JAVA is used.

However, the implementation of the Runge–Kutta–Nyström method can not be used for the approximation of differential equations of other than second order. This method is particularly for the approximation of second order equations, and thus not suitable for equations of another order.

The other approximation methods mentioned in chapter 4, the Euler–Cauchy method and the Midpoint method, can be implemented in the same way as the Runge-Kutta method. However, this is not discussed in detail in this chapter. The correctness of the implementation of these methods was tested using the examples, which were discussed in chapter 4. In [8] and [10] a number of tables, which contain the resulting values of these differential equations, are printed. These tables were used to check the results of the tests, which were performed to check the correctness of the JAVA code.

7.2 Structure of the classes used for the approximation

The traditional way of writing software for the approximation of differential equations is to provide one subroutine, which can be called when a result is required. This approximation routine is mostly a black box module, which has an input and an output, but which does not reveal the implementation.

Further, when solving differential equations there is no optimal approximation method for all initial value problems. A choice between the different methods that are available, has to be made depending on the kind of problem, the accuracy of the results which are needed, and the time within which the results are required.

This leads to the conclusion that the way of implementing described above is not preferable. Instead an object-oriented approach, which is in consistency with the rest of the code, is chosen for. Furthermore, this object-oriented approach has the advantages that it allows reuse of certain classes, which are used for the implementation of the approximation methods, and that it allows other approximation methods, than the ones provided, to be implemented.

The ideas behind this approach, and the structure of the classes used for the implementation of the approximation methods, are rather similar to the experimental software project which is discussed in [22]. However, the implementation discussed in [22] is more extended, and is written in the C++ programming language, in contrast to this graduation project, which makes use of the JAVA programming language.

The inheritance graph for the classes used for the implementation of the approximation methods in this graduation project is depicted in figure 7.1.

`FirstOrderApprox` and `SecondOrderApprox` are interfaces which are used in the approximation of differential equations. An *interface* is a completely unimplemented prototype for a class. This means that no methods have been implemented in the class, in contrast to an *abstract class*, which has been left partially unimplemented. The benefits of using interfaces are much the same as the benefits of using abstract classes. They provide a means to define the protocols for a class without worrying about the details of the implementation. The fea-

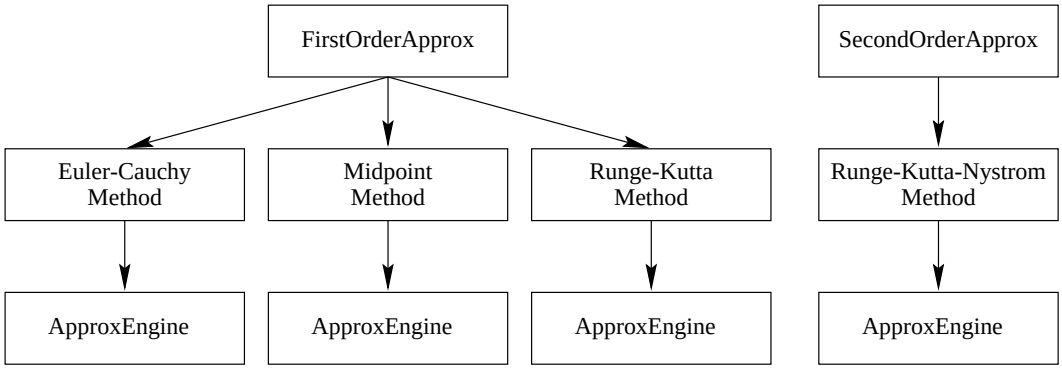

Figure 7.1: Inheritance graph of the classes used for approximation

ture that seperates interfaces from abstract classes is that a class can implement multiple interfaces, but can extend only a single abstract class. This feature of interfaces can be used to work around the single inheritance rule of JAVA. In the C++ programming language *multiple inheritance*, which means a class can be derived from multiple parent classes, is supported. In JAVA this concept is not supported, but the use of interfaces is a way to work around it.

The `FirstOrderApprox` and the `SecondOrderApprox` interface provide the prototypes for the implementation of the differential equation, which is possibly reduced to multiple first order equations for the `FirstOrderApprox` interface, the method which provides the implementation of the approximation method, and the method used to set the parameters and to call the approximation method.

The classes which implement the `FirstOrderApprox` and the `SecondOrderApprox` interface are the approximation methods. These classes are abstract classes, since not all prototypes of the methods have been implemented in these classes. In fact, only the approximation methods are implemented in these classes. Other approximation methods, than the ones provided, can be implemented by writing a new class, which implements the `FirstOrderApprox` or the `SecondOrderApprox` interface.

The methods, which provide the implementation of the differential equation, and the methods, which are used to set the parameters and to call the approximation method, are implemented in the `ApproxEngine` class. Another class, `ApproxControl`, is used to control the process of approximation in the models discussed throughout this thesis. This class also provides, and possibly controls, the value of the stepsize, which is used in the approximation methods. When a new model is implemented using the approximation classes, an `ApproxControl` class can be written, which is suitable for the model.

It is also possible to create a user interface for a model, in which a choice can be made for the approximation method that should be used. However, this is not implemented in the models of this graduation project.

7.3 The rest of the Java code

The JAVA code, which implements the rest of the models, consists of a class that implements the user interface, and a class that contains the methods of the *Advance Action Approach* and that performs the drawing. The latter of these classes implements the `Runnable` interface, which provides a means to make use of threads. *Threads* allow a program to have multiple sequences of execution within it. In this way multiple tasks can be performed concurrently. This concept is particularly useful for the incoming inputs of the user interface. The use of threads makes it possible to use the buttons and the scrollbars of the user interface during execution of the model.

The class which implements the user interface also initiates the class `ApproxControl`, which was discussed in paragraph 7.2, and the class that contains the methods of the *Advance Action Approach* and that performs the drawing. In the latter of these classes the same `ApproxControl` class is known. This `ApproxControl` class initiates the `ApproxEngine`, which extends the approximation method, and which implements the `FirstOrderApprox` or the `SecondOrderApprox` interface.

Further, the class that implements the user interface also handles the events. This is performed as in a typical *soft real-time* system, events are handled by the system as fast as possible, but they are not constrained to finish by or within specific times. However, the handling of events is not discussed in detail in this paragraph. A detailed discussion of the handling of internal and external events is given in paragraph 8.3.

Since the events are handled as fast as possible, and thus not necessary at the moment the event occurs, there can be a small deviation after the occurrence of the event, compared to the motion of the models in the real world. However, since the models make use of an approximation method for the differential equations which describe the motion, there will always be a small deviation compared to the reality. When an event is handled faster, the deviation will be smaller. This can be accomplished by the use of a *Just In Time* (JIT) compiler, which can improve the performance of JAVA by 10 to 30 times. Another benefit of such a compiler is that the stepsize can be reduced, since a faster execution of the program means that more computations can be performed. This decrement of the stepsize means that the approximation of the differential equations will be more accurate, and that the chances of missing a detection of a collision in the model of the double pendulum, which was discussed in paragraph 6.4, become smaller.

7.4 Just In Time (JIT) compiler

Compared to natively compiled programs written in languages such a C, C++ or PASCAL, JAVA programs have a relatively slow execution speed. Although JAVA programs have been criticized because of this, they also have a big advantage, which native programs do not have, namely platform independence. To overcome the reduced performance of JAVA programs a *Just In Time* (JIT) compiler, which compiles JAVA bytecode executables into native programs

just before execution, can be used.

To explain the concept of a *Just In Time* (JIT) compiler and how it fits into the JAVA runtime system, first the JAVA Virtual Machine has to be explained. In [23] the JAVA Virtual Machine is defined as:

> An imaginary machine that is implemented by emulating it in software on a real machine. Code for the Java Virtual Machine is stored in `.class` files, each of which contains code for at most one public class.

Further, [23] provides the hardware platform specification to which all JAVA code is compiled. This specification enables JAVA programs to be platform independent, because the compilation is done for an imaginary machine. This means that when a JAVA program is compiled, it is compiled to be executed on the Virtual Machine. This in contrast to natively compiled programs, which are compiled to run on a real, non-virtual hardware platform. The Virtual Machine itself has characteristics very much like a physical microprocessor, but it consists entirely of software. The purpose of it is to allow JAVA programs to compile to a uniform executable format, as defined by the Virtual Machine, that can run on any platform. However, at some point, all programs written in JAVA have to run on a particular underlying hardware platform. Therefore, it is up to the JAVA interpreter of each specific hardware platform to ensure the running of code compiled for the JAVA Virtual Machine. The way this works is that JAVA programs make calls to the Virtual Machine, which in turn routes them to appropriate native calls on the underlying platform.

Just as all microprocessors, the JAVA Virtual Machine has an instruction set that defines the operations it can perform. The JAVA compiler generates bytecode executables from JAVA source files. These bytecodes, which can be seen as machine language for the Virtual Machine, are always stored in `.class` files. In figure 7.2 (a) the role of the Virtual Machine in the JAVA environment is depicted.

The use of a *Just In Time* (JIT) compiler slightly alters the role of the Virtual Machine by compiling JAVA bytecode directly into native platform code, instead of interpreting the bytecodes and making the appropriate native calls on the underlying platform. In this way the execution speed can be improved significantly, since the native code can be executed directly on the underlying platform.

Figure 7.2 (b) depicts how the *Just In Time* (JIT) compiler alters the role of the Virtual Machine in the JAVA environment. Notice that instead of the Virtual Machine calling the underlying native operating system, it calls the *Just In Time* (JIT) compiler. The *Just In Time* (JIT) compiler in turn generates native code that can be passed on to the native operating system for execution. The benefit of this arrangement is that a *Just In Time* (JIT) compiler can be integrated into a system, without affecting any other part of the JAVA runtime system. This means a *Just In Time* (JIT) compiler can be plugged in by the user to benefit from the improvement of the performance of JAVA by 10 to 30 times, without any other work or side effects.

Finally, it should be noticed that also browsers are available which have the option of a *Just In Time* (JIT) compiler. The most familiar of these are Netscape Navigator and Microsoft's

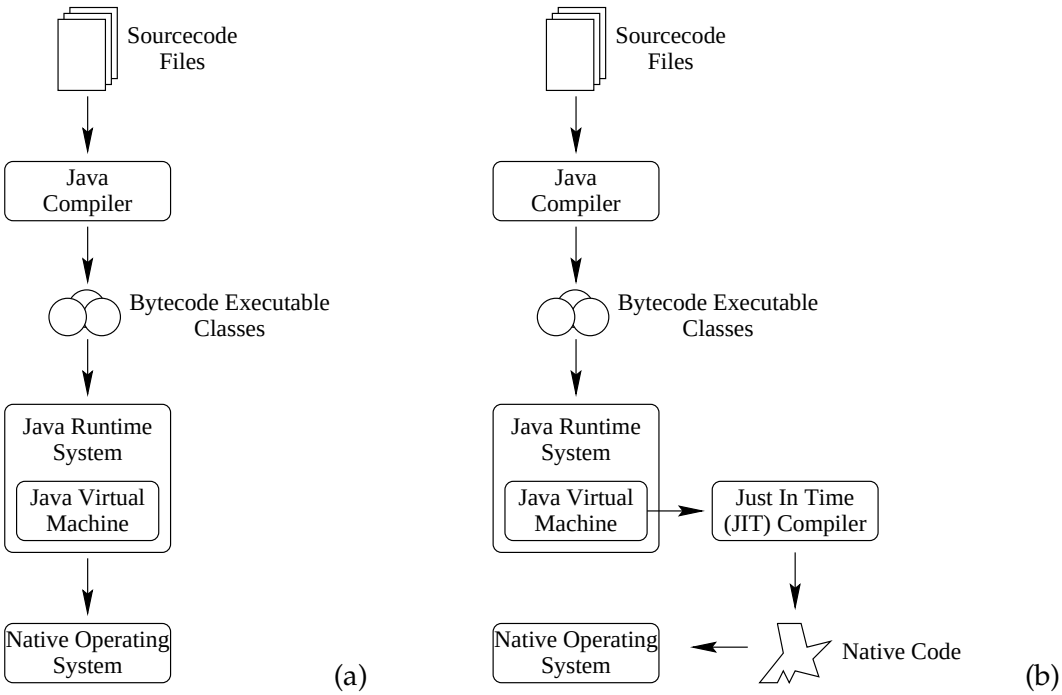

Figure 7.2: The JAVA Virtual Machine and the *Just In Time* (JIT) compiler
These figures were taken from [5].

Internet Explorer.

Chapter 8

Evaluation of the Java code

This chapter evaluates the main implementation problems which have been encountered during the course of this graduation project. Since these problems arise from the `currentTimeMillis()` method and the `repaint()` method, these will be focussed on. Furthermore, the characteristics of the JAVA code of the internal and external events are being discussed and compared with the characteristics of the OMSIM simulations.

8.1 The currentTimeMillis() method

When the applications are run on a Windows 95 operating system on a Pentium 133, and `delta`, which is the time delay between two `advance()` method calls and thus the time between two successive frames, is examined, it can be noticed that the range of `delta` is limited to a number of specific values, according to the following distribution:

`delta` (ms)	average number of times (out of a total of 1000)
0	815
50	94
60	90
other	1

Here *other* is a value, other than 0, 50 or 60 ms, of the form $(a\ 50 + b\ 60)$, where $a, b \in \mathrm{N}$.

At first, the value of `delta` of 0 ms appears strange, but of course it is possible the calculations are performed within a time of 1 ms, when a fast CPU is used. However, the fact that only the other specific values of `delta` appear and nothing in between of these values, cannot be explained by this. In trying to find an explanation of these other values, the timer used in the models, the `currentTimeMillis()` method, was tested on different platforms using a small application, which polled the output of the `currentTimeMillis()` method minus the starting time every 10 ms. This resulted in the data listed in table 8.1.

Poll number	Windows 95 on a Pentium 133	UNIX on a SPARC station	Red Hat Linux 4.2 on a Pentium 133
1	50	18	21
2	110	86	90
3	110	105	110
4	160	128	130
5	160	145	150
6	160	169	170
7	220	186	190
8	220	208	210
9	220	225	230
10	270	246	250
11	270	297	270
12	270	316	290
13	270	336	310
14	330	356	330
15	330	376	350
16	330	395	370
17	330	417	390
18	380	436	410
19	380	456	430
20	380	476	451

Table 8.1: Output of the `currentTimeMillis()` method minus the starting time every $10\ ms$, when the test is performed on Windows 95 running on a Pentium 133, on UNIX running on a SPARC station, and on Red Hat Linux 4.2 running on a Pentium 133

From table 8.1 it can be seen that the `currentTimeMillis()` method only exhibits unexpected behaviour, when the test is performed on a Windows 95 operating system running on a Pentium 133. Namely, the outcomes remain constant for a number of polls, and only yield answers of the form $(a\ 50 + b\ 60)$, where $a, b \in$ N. When the test is performed on a UNIX operating system running on a SPARC station, this behaviour cannot be noticed. Also on the Red Hat Linux 4.2 operating system, running on the same Pentium 133 as the Windows 95 operating system, this behaviour does not present itself.

A possible explanation of this behaviour can be found in the basic time tick of the early MS-DOS, which has a frequency of $18.2\ Hz$, which is equal to a time tick every $54.945\ ms$. Probably this time tick is still used within Windows 95, and thus used by the Java Development Kit of Sun Microsystems, Inc. It is possible that this time tick, which takes place every $54.945\ ms$, is rounded to 50 or $60\ ms$, which also can explain the slightly more frequent appearance of the `delta` of $50\ ms$, since $54.945\ ms$ is closer to $50\ ms$ than to $60\ ms$. When no time tick takes place during the calculation, a `delta` of $0\ ms$ appears.

Inquiring on the JAVA newsgroups *comp.lang.java.programmer* and *comp.lang.java.help* confirmed this possible explanation. It was also mentioned that there are certain ways around this in Windows 95, but that Sun Microsystems, Inc. did not make use of these ways in the Java Development Kit (for Windows 95).

When looking at the applications this behaviour does not appear to be a problem, since the motion of the models is apparently the same as on the other operating systems, according to the reality. However, the motion of the models does differ from that in the real world. When there is a `delta` of $0\ ms$, the object in the new frame has exactly the same position as the object in the previous frame. Since multiple `delta`'s of $0\ ms$ appear after eachother, also multiple successive frames are drawn with the object at the same position, although the new position should be another position equal to that in the real world. Thus, it is most certainly a problem.

As long as there is no solution for this problem in the Windows 95 port of the Java Development Kit of Sun Microsystems, Inc. or within Windows 95 itself, there will not be a reliable way to ensure that the dynamics of the models are reasonably close to those in the real world. For the UNIX and the Linux port of the Java Development Kit this problem does not exist.

8.2 The repaint() method

The `repaint()` method, which is called when a new state of a model has been calculated and a new frame has to be drawn, causes a call to the `update()` method as soon as possible. This means that the `update()` method, in most cases, will not be called at once when the drawing of a new frame is needed, since it is likely the loop

```
while(condition) {
  advance();
  repaint();
}
```

proceeds before the request to draw a new frame is completely handled. In some cases even the next `repaint()` method is called before the preceeding call to `repaint()` is completely handled.

In the models discussed throughout this thesis it needs to be ensured every call to the `advance()` method, which calculates a new state of a model, is immediately followed by the drawing of the corresponding new frame. In this way the new frame is drawn before the loop proceeds.

To obtain this behaviour, for every `update()` method one `repaint()` method has to be called. This can be established by the use of a boolean flag, which is set to TRUE by the `update()` method when the drawing of the new frame is done, and which is set to FALSE in the loop, just before the `repaint()` method is called. The disadvantage of this approach is

that the loop has to check the boolean flag constantly, whether the drawing of the new frame is done. This consumes CPU time, which could have been used for other purposes.

Another way to establish this behaviour, without unnecessary consumption of CPU time, is the use of a monitor [24]. In JAVA every object with *synchronized* methods is a monitor. The monitor lets only one thread at a time execute a *synchronized* method on an object. The monitoring can be established in the following way, by using a *synchronized* `run()` method, that contains the loop in which the `advance()` method and the `repaint()` method are called, and a *synchronized* `update()` method:

```
public synchronized void run() {
  last_advance = System.currentTimeMillis();
  while(condition) {
    advance();
    repaint();
    try {wait();} catch(InterruptedException e) {}
  }
}

public synchronized void update(Graphics g) {
  erase_the_screen();
  draw_the_frame();
  notify();
}
```

In this way the `run()` method and thus the loop have to wait, after the `repaint()` method has been called, until the `run()` method is signalled to proceed by the monitor. When the `update()` method, which was called by the `repaint()` method, is done with the drawing of the new frame, the monitor is notified other threads can be executed, and thus the `run()` method and its loop can proceed execution. The advantage of this approach is that, in contrast to the use of a boolean flag, no CPU time is consumed, which could have been used for other purposes.

8.3 Internal and external events in the Java code

As mentioned before in paragraph 2.4, a number of internal and external events are defined in the models discussed throughout this thesis. In the pendulum model a tap can be given on the pendulum, which is a typical external event. In the OMSIM simulation this is represented by a change of the amount of the angular velocity at a predefined moment of time. In this way the tap can be scheduled, to occur at this moment, at the beginning of the simulation.

```
ONEVENT Init DO
 schedule(Tap, t);
END;
```

```
ONEVENT Tap DO
 new(y') := y' + v;
END;
```

However, using this approach it is not possible to give multiple taps during a single run of a simulation. Also the notion of momentum is not considered. In the JAVA code of this model these two deficiencies do not present themselves. The button to give a tap on the pendulum can be pushed at any moment during the simulation, which means the momentum defined by a scrollbar is given to the pendulum. Then this event is handled as fast as possible, by calculating the change of velocity that results from the momentum which is given to the pendulum. This is done by the following method from the class `ApproxControl`, in which m equals the mass of the bob of the pendulum:

```
public void setMomentum(double momentum) {
  velocity += (momentum / m);
}
```

In this model the next state of the system is computed by an approximation of the next position of the pendulum, in which possibly a changed angular velocity is used due to a tap. Therefore, no particular actions have to be undertaken in the loop in the `ApproxControl` class:

```
public void approx(double d) {
  time_next = time + (d / 1000);
  while (time < time_next) {
    angle_engine.approx();
    time += stepsize;
  }
}
```

In this method the parameter d represents the `delta` of the *Advance Action Approach*, which is the difference between the current time and the time of the last advance action. It has to be divided by 1000, since d describes milliseconds. The approximation is done using the predefined stepsize, until the loop reaches `time_next`, which is the time of the next state.

The model of the bouncing ball only contains an internal event, namely the bounce of the ball on the floor. The conditions for this bounce to occur are that the height has to be equal to zero and that the velocity has a negative value, which means the ball is moving in the downward direction. In the OMOLA model this is translated into the following events:

```
ONEVENT y <= 0 AND v < 0 DO
 IF abs(v) < 0.1 THEN schedule(Stop, 0) ELSE schedule(Bounce, 0);
END;

ONEVENT Bounce DO
 new(v) := abs((1 - e) * v);
END;
```

The condition of the height has to be smaller than or equal to zero, since the height is a continuous variable that cannot be checked on equality. The Stop event is scheduled when the height is smaller than or equal to zero, and the velocity of the ball is almost zero, since this means the ball is almost not bouncing anymore. In this way it is prevented the event will be fired an infinite number of times. The Bounce event calculates the new velocity after a bounce of the ball, using a certain percentage of energy-loss represented by the variable e. Another variable, which is being used in the differential equation of this model, is the damping direction w, which was discussed in paragraph 5.2. This continuous variable has to be equal to -1 or 1, depending on the sign of the velocity. Therefore, the statement `w = sign(v);` is included in the model.

The JAVA code of the model of the bouncing ball makes use of a slightly different method `approx(double d)` in the class `ApproxControl`, than the model of the pendulum does:

```
public void approx(double d) {
  time_next = time + (d / 1000);
  while (time < time_next) {
    dampingDirectionDecision();
    velocityengine.approx();
    heightengine.approx();
    BounceDetection();
    time += stepsize;
  }
}
```

In this case two approximations are performed, namely of the differential equation of the velocity and of that of the height. First the differential equation of the velocity has to approximated, since the value of the velocity, at a certain time, has to be known to approximate the value of the height at that time. Before these approximations are performed, the direction of the damping has to be decided. This is done by the following method, which is called in the loop before the approximation methods are called:

```
private void dampingDirectionDecision() {
  w = (velocity <= 0) ? -1 : 1;
}
```

This method provides the same functionality as the statement `w = sign(v);` in the OMOLA model. After the new velocity and the new height have been approximated, the method `BounceDetection()` is called. This method checks the conditions for the occurrence of a bounce of the ball at the time of these new values. It resembles the event which was defined in the OMSIM simulation:

```
private void BounceDetection() {
  if (height <= 0 && velocity < 0) {
```

```
    velocity = Math.abs((1 - e) * velocity);
  }
}
```

In the model of the double pendulum both internal and external events exist. As mentioned before, the internal event of the model is a collision of the pendulums, and the external event is a tap on one of the pendulums. The latter of these is handled in the same way as in the OMOLA model of the single pendulum. The taps, which may be given on each of the pendulums, are scheduled at the beginning of the simulation to occur at a predefined moment of time. The notion of momentum is not considered, instead the amount of the angular velocity is changed at this moment. These deficiencies do also not present themselves in the JAVA code of this model.

```
ONEVENT BothTaps DO
 schedule(Tap1, t1);
 schedule(Tap2, t2);
END;

ONEVENT Tap1 DO
 new(y1') := y1' + v1;
END;

ONEVENT Tap2 DO
 new(y2') := y2' + v2;
END;
```

The collision event in this OMSIM simulation makes use of the conditions that were derived in paragraph 6.4:

```
ONEVENT abs(y1 - y2) < SmallestAngle AND
        (y1 - y2) * (y2' - y1') > 0 DO
 IF abs(y1) < 0.75 * SmallestAngle AND abs(y1') < 0.25 AND
    abs(y2) < 0.75 * SmallestAngle AND abs(y2') < 0.25
 THEN schedule(Stop, 0)
 ELSE schedule(Collision, 0);
END;

ONEVENT abs(y1 - y2) > 2*PI - SmallestAngle AND
        (y1 - y2) * (y1' - y2') > 0
CAUSE Collision;
```

The first of these events can take place anywhere on the circle that each of the pendulums make, except at the top of this circle. The second event can only take place at the top of this circle. Therefore, the Stop event only has to be scheduled when the first event holds, and both pendulums are at the bottom of the circle that each of the pendulums make, with a velocity that approaches zero. The Collision event, which is caused by these two events, makes use of the equations which were derived in paragraph 5.3.3:

```
ONEVENT Collision DO
 new(y1') := ((1 - m2/m1) * y1' + 2 * (m2/m1) * y2') / (1 + m2/m1);
 new(y2') := (2 * y1' + (m2/m1 - 1) * y2') / (1 + m2/m1);
END;
```

Further, the OMOLA model defines a number of events that ensure the angle of displacement is limited to a range of $[-\pi, \pi]\ rad$, which was discussed in paragraph 6.4. Therefore, the following events are defined:

```
ONEVENT y1 > PI DO
 new(y1) := y1 - 2*PI;
END;

ONEVENT y2 > PI DO
 new(y2) := y2 - 2*PI;
END;
```

```
ONEVENT y1 < -PI DO
 new(y1) := y1 + 2*PI;
END;

ONEVENT y2 < -PI DO
 new(y2) := y2 + 2*PI;
END;
```

In the JAVA model of the double pendulum a tap on one of the pendulums is handled in the same way as in the JAVA model of the single pendulum. The method `approx(double d)` in the class `ApproxControl` in the double pendulum model is defined as follows:

```
public void approx(double d) {
  time_next = time + (d / 1000);
  while (time < time_next) {
    angle1_engine.approx();
    recalcAngleOne();
    angle2_engine.approx();
    recalcAngleTwo();
    collisionDetection();
    time += stepsize;
  }
}
```

First, two distinct `ApproxEngines` approximate the angle of displacement of both pendulums. The sequence in which these approximations are performed does not matter, since both angles of displacement have to be computed before the pendulums are drawn. However, when the angle of displacement of a pendulum exceeds π or $-\pi\ rad$, it has to be recalculated before both pendulums are drawn. This is performed by the `recalcAngleOne()` and the `recalcAngleTwo()` methods:

```
private void recalcAngleOne() {
  if (angle1 > PI) {
    angle1 -= doublePI;
```

```
private void recalcAngleTwo() {
  if (angle2 > PI) {
    angle2 -= doublePI;
```

```
  }                                    }
  else if (angle1 < -PI) {             else if (angle2 < -PI) {
    angle1 += doublePI;                  angle2 += doublePI;
  }                                    }
}                                    }
```

After the approximation of the new state of the system of the double pendulum, it has to be evaluated whether it results in a collision of the pendulums. If this is the case the velocities of both pendulums have to be adjusted, according to the equations which were derived in paragraph 5.3.3. This evaluation, whether a collision takes place, is performed by the method `collisionDetection()`:

```
private void collisionDetection() {
  if ((Math.abs(angle1 - angle2) < 0.075 &&
      (angle1 - angle2) * (velocity2 - velocity1) > 0) ||
      (Math.abs(angle1 - angle2) > 2 * Math.PI - 0.075 &&
      (angle1 - angle2) * (velocity1 - velocity2) > 0)) {
    v1 = ((1 - m2/m1) * velocity1 + 2 * (m2/m1) * velocity2)
         / (1 + m2/m1);
    v2 = (2 * velocity1 + (m2/m1 - 1) * velocity2)
         / (1 + m2/m1);
    velocity1 = v1;
    velocity2 = v2;
  }
}
```

In this method a *SmallestAngle* of $0.075\ rad$, which was determined by experiment, is used. Further, two auxiliary quantities `v1` and `v2` are used to calculate the new velocities after the collision.

The detection of a collision also can be missed in the JAVA code of the double pendulum, for the same reason that was discussed in paragraph 6.4. This could be prevented in the JAVA model by means of collision detection of two objects. The easiest way to do this is to consider whether the two images overlap. However, this is not very accurate, especially if the image contains transparent pixels. Another approach is to check each pixel in each image, whether two non-transparent pixels are at the same location. The disadvantage of this approach is the high cost of this operation. Therefore, it is not suited for an interpreted language such as JAVA.

The last option is to associate with each image a smaller rectangle which lies inside the image, as depicted in figure 8.1. When an appropriate rectangle is chosen for each image, it will cover most of the colored pixels and not too much of the transparent pixels. The advantage is that comparison of two rectangles to determine whether they overlap, is much quicker than comparing all the colored pixels within the image.

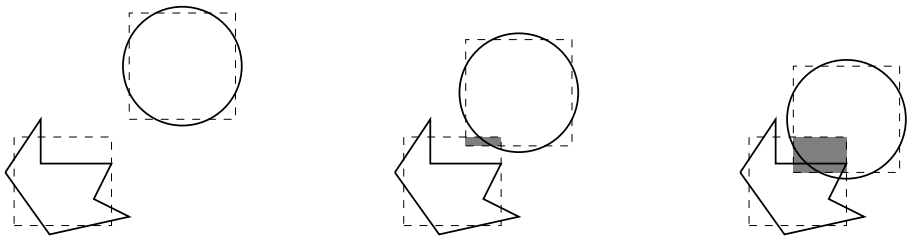

Figure 8.1: Collision detection using a rectangle on the image
This figure was taken from [5].

However, this way of collision detection of two objects is not used in the model of the double pendulum. The benefits, of never missing a collision of the pendulums, do not compensate the inaccuracy of this way of collision detection. Although the detection of a collision event can sometimes be missed by using the code that was described above, it is the most appropriate way to ensure the model acts as it would do in the real world. Furthermore, the collision event can only be missed at a very high angular velocity, which means the model is most certainly appropriate for simulations in which the angular velocity is not of an extreme extent.

The use of a *Just In Time* (JIT) compiler, which was discussed in chapter 7, can improve this situation, since the stepsize used in the approximation can be decremented. This is a result of the improvement of the performance of JAVA by 10 to 30 times, which means that more computations can be performed in the same time. This decrement of the stepsize means that the chances of missing a detection of a collision, which was discussed in paragraph 6.4, become smaller.

Chapter 9

Optimization of Java code

This chapter discusses the results of the tests which were performed to find the average number of frames per second of the models discussed throughout this thesis. It will also focus on the optimization of the JAVA code to improve the performance of the models, which means the increment of the average number of frames per second. Further, the importance of optimizing of JAVA code in general will be discussed.

9.1 Frames per second before optimization

What all forms of animation have in common is that they create some kind of perceived motion by showing successive frames at a relatively high speed. Computer animation usually shows 10 to 20 frames per second. By comparison, traditional hand-drawn animation uses anywhere from 8 frames per second, for poor quality animation, to 12 frames per second, for standard animation, to 24 frames per second, for short bursts of smooth and realistic motion [4]. A typical example of the use of a frame rate of 24 frames per second is a modern movie. A movie has a constant frame rate, in contrast to a computer animation, which exhibits an irregular frame rate. This irregular frame rate is a consequence of the competition for available CPU time by numerous processes. Though the *Advance Action Approach* provides a way to run applications apparently at constant speed, in some cases a hesitation of the application can be noticed due to this competition. Because of the irregular frame rate of a computer animation, it is only meaningful to consider the average frame rate.

To improve the quality of the animation, double buffering is used in the models discussed throughout this thesis. Instead of drawing directly on the drawing area, where it can be seen, a second offscreen drawing area is created on which the next frame is drawn. When the frame is completed, it is copied onto the screen where it can be seen. In this way animation flicker is eliminated. The reason for this improvement is that the whole image is copied at once onto the screen, without extra computions during drawing on the screen. In this way the viewer perceives that all the work is done instantaneously, even though more work is done between frames. In spite of the improvement of the quality of the animation, double buffering does not increase the average number of frames per second.

Approximation method	Frames per second pendulum	Frames per second bouncing ball	Frames per second double pendulum
Runge–Kutta	19	$11\frac{1}{2}$	$13\frac{1}{2}$
Midpoint	$19\frac{1}{2}$	12	16
Euler–Cauchy	$19\frac{1}{2}$	$12\frac{1}{2}$	$17\frac{1}{2}$
Runge–Kutta–Nyström	$19\frac{1}{2}$	–	$16\frac{1}{2}$

Table 9.1: Indication of the average number of frames per second of the pendulum, the bouncing ball and the double pendulum application before optimization, when using different approximation methods and a stepsize of 0.01

A number of tests, to find the average number of frames per second of the models, were performed on a UNIX operating system running on a SPARC station. The models used in the tests were compiled using the $-O$ option of the JAVA compiler, which performs some simple optimizations by inlining certain method calls. In this way the performance of small, frequently executed methods is improved by eliminating the overhead of the method call. Further, the stepsize used in in the performance of the tests is of interest. Using a small stepsize means that the loop

```
while (time < time_next) {
  approxengine.approx();
  time += stepsize;
}
```

in the method `approx(double d)` in the class `ApproxControl` will be executed more often, and thus the method takes a larger amount of time. On the other hand, using a larger stepsize means the loop will be executed less often, and thus a shorter amount of time will be consumed by the same method. Finally, it has to be mentioned that these tests did not make use of a *Just In Time* (JIT) compiler, which has been discussed in chapter 7. The average number of frames per second of the models which resulted from these tests are listed in table 9.1.

It is difficult to measure accurate data in these tests, since the load of the CPU differs in time. Therefore, the data in table 9.1 have to be seen as an indication of the average number of frames per second of the models. Further, it has to mentioned that only a table of the average number of frames per second of the applications is given in this thesis. A table containing the same data of the applets is not given, since both tables lead to the same conclusions.

To approximate the motion of the models, using the Runge–Kutta, Midpoint and Euler–Cauchy methods, the second order differential equations of the pendulum and the double pendulum are rewritten into a set of multiple first order equations. The Runge–Kutta–Nyström method is used to approximate the second order differential equations of the pendulum and the double pendulum directly. For the bouncing ball this method cannot be used, since the differential equations describing the motion of the bouncing ball are first order equations.

From table 9.1 it follows that the Midpoint method, the Euler–Cauchy method and the Runge–Kutta–Nyström method result in a larger average number of frames per second than the Runge–Kutta method, and thus take less time to approximate the system of equations. Further, it can be noticed that the bouncing ball has the smallest average number of frames per second and the pendulum model the largest. This is mainly a consequence of the number of differential equations which are needed to describe the motion of a model. The pendulum makes use of one differential equation and the bouncing ball and the double pendulum make use of two differential equations. Approximating a larger number of equations will consume more time. Another cause is the area of the image which has to be updated. Updating a larger area will take more time, which is the case in the model of the bouncing ball.

9.2 The importance of optimizing

As an interpreted language, executing bytecode instead of natively compiled code, JAVA has a reduced performance compared to many other languages. To improve the performance of JAVA code, low-level optimization techniques can be used, next to the $-O$ option of the JAVA compiler and a *Just In Time* (JIT) compiler. Besides the improvement of the performance, also the limitation of the resources, for example available memory or bandwidth of a network, is a reason to optimize JAVA code. However, there are a few general matters which should be considered before optimizing:

- Optimizing can cause new, and possibly subtle, bugs in the code.
- Optimization tends to make code harder to understand and maintain.
- The highest improvements are in most cases changing the algorithm.
- A lot of time can be spent optimizing, with little improvement in the performance.
- Normally about 10 percent of the code takes about 90 percent of the execution time[1]. Optimizing this 10 percent of the code has the most effect on the performance.

Finally, it should be mentioned that calling native code, for example C functions or assembly, may also improve the performance. However, a disadvantage is that platform independence is lost.

9.3 Optimization of the models

The most effective techniques used in the models to improve the performance, and thus the quality, of the animation, are the updating of only the relevant parts of the image and the monitoring of the `repaint()` and the `update()` methods, as discussed in paragraph 8.2. The first technique consumes less time than updating the whole image. The latter provides

[1] This is known as the 90–10 rule. However, some people prefer to call it the 80–20 rule.

the fastest, and thus the most suitable, way to ensure the `repaint()` request will be completed before proceeding with the computation of the next state of the system. Therefore, both these techniques ensure, in contrast to double buffering, an increment of the average number of frames per second.

Other optimizations, which are used at low-level in the JAVA code of the models, are the following:

- Normally, a `repaint()` request will call the `update()` method, which in turn erases the whole drawing area and calls the `paint()` method, which does the drawing. In the models discussed throughout this thesis, the `update()` method erases only the relevant parts of the drawing area, as discussed above, and draws the new image. In this way there will be one method call less on every `repaint()` request, since the `paint()` method does not have to be called. The `paint()` method, which is being called when a model is first being shown or which is being called automatically, for example when the window of a model is covered and uncovered by another window, calls the `update()` method in the model. This will take one more method call, but in general the `repaint()` method will be called much more often than the `paint()` method.

- Statements containing the `new` operator, which creates a new object by asking the computer for enough memory to store an object of the specified type, are used as much as possible outside the actual methods of the program, since creating new objects consumes a lot of time. Also existing objects are reused as much as possible, which, besides reducing the consumed time, has more advantages, since less memory is used and less garbage collection is needed, which both also increase the execution time.

- Integer arithmetic is used instead of floating point arithmetic whenever possible, since integer operations are much faster than floating point operations.

- The same calculations, which take place a number of times or which take place within a loop, are stored in a variable, which is used in the code instead of calculating the needed value every time. This will reduce the number of operations which are needed and thus reduces the consumed time.

- Using the `.` operator to access objects and instance variables takes a lot of time. In the code of the models big hierarchies of this operator are avoided whenever possible. Also objects and instance variables, which have to be accessed more often, are stored in a variable, which is used in the code whenever possible.

- The variable for the loop counter, used in the loops of the models, is a local integer variable. Further, the loops are restructured, by optimizing the compare operation, to improve the performance. The loop which was used first was of the following structure: `for (int i = 0; i < max; i++)`, and the loop used after optimizing has the following structure: `for (int i = max; --i >= 0;)`. The second loop has two advantages: the comparison is done against a constant, which is faster than com-

Approximation method	Frames per second pendulum	Frames per second bouncing ball	Frames per second double pendulum
Runge–Kutta	$25\frac{1}{2}$	$14\frac{1}{2}$	16
Midpoint	$26\frac{1}{2}$	15	19
Euler–Cauchy	27	$15\frac{1}{2}$	20
Runge–Kutta–Nyström	$25\frac{1}{2}$	–	20

Table 9.2: Indication of the average number of frames per second of the pendulum, the bouncing ball and the double pendulum application after optimization, when using different approximation methods and a stepsize of 0.01

parison with another variable, and the decrement operator directly with the comparison is faster than a special increment operation.

- The classes used in the models are made `final` whenever possible, which means other classes cannot inherit from these classes, and the methods used in these classes are automatically `final`. The methods used within the classes are declared `private`, `final` or `static` if possible. Only in this way the JAVA compiler can inline these methods and thus reduce the number of method calls, so that less time will be consumed.

Using these techniques and optimizations, to improve the performance of the model of the pendulum, the bouncing ball and the model of the double pendulum, the average numbers of frames per second listed in table 9.2 were measured on the same platform as of the tests which were done before the optimization. These data also have to be seen as an indication of the average number of frames per second.

Comparing table 9.1, which indicates the average number of frames per second before optimization, with table 9.2, which indicates the same data after optimization, it can be noticed the average number of frames per second increases considerably when these optimization techniques are applied.

Running the models on another platform has an effect on the average number of frames per second as well, which can be seen in table 9.3. In this table the average number of frames per second of the models, running on the Red Hat Linux 4.2 operating system on a Pentium 133, is shown. It can be noticed the average number of frames per second has increased considerably, due to the use of a faster CPU, and less processes which are competing for CPU time. In spite of the high average number of frames per second, sometimes a hesitation of the model can still be noticed, as a consequence of the competition for available CPU time by numerous processes. However, the applications run apparently at the same constant speed as on the UNIX operating system on a SPARC station. The same test on a Windows 95 operating system could not be performed, because of the problem of the `currentTimeMillis()` method in the Windows environment, which was discussed in paragraph 8.1. In this way no reliable time between two frames could be measured and thus no reliable average number of frames per second could be calculated.

Approximation method	Frames per second pendulum	Frames per second bouncing ball	Frames per second double pendulum
Runge–Kutta	67	$34\frac{1}{2}$	52
Midpoint	67	36	58
Euler–Cauchy	$67\frac{1}{2}$	37	60
Runge–Kutta–Nyström	67	–	$57\frac{1}{2}$

Table 9.3: Indication of the average number of frames per second of the pendulum, the bouncing ball and the double pendulum application after optimization, when using different approximation methods and a stepsize of 0.01, on the Red Hat Linux 4.2 operating system running on a Pentium 133

9.4 Other ways to optimize Java code

There are many other techniques, which could be used in optimization of JAVA code, besides the ones used in the models discussed throughout this thesis. Some of these techniques are not appropriate to be used in these models, other techniques make the code harder to understand or maintain. The latter techniques are not used in the models, since the maintainability and, mainly, the readability of the code is considered to be of more importance in this graduation project. A number of other techniques to optimize the JAVA code are the following:

- To find out where to start the optimizing and where the best improvements are made, a profiling tool could be used. The Java Development Kit of Sun Microsystems, Inc. has a profiling option build in the JAVA interpreter. The output of this profiling option can be analyzed using a profiling tool such as Profile Viewer[2] or HyperProf[3]. Java Workshop 2.0 of Sun Microsystems, Inc. also has a profiling option, with graphical output.
- Using a machine with multiple processors and a JAVA Virtual Machine that can spread threads across these processors, the performance can be improved by multithreading, either manually or through the use of a restructuring compiler, such as JAVAR.
- The use of synchronized methods should be avoided whenever possible, since calling a synchronized method is about 10 times slower than calling an unsynchronized method, and when using a *Just In Time* (JIT) compiler this number increases by 50 to 100 times. However, in the models discussed throughout this thesis, synchronized methods are used to monitor the `repaint()` and the `update()` methods. In this case the advantage of using synchronized methods exceeds the disadvantage of more consumed time.
- Exceptions should be avoided whenever possible, since they consume much time.
- A class from the JAVA API, or a method in such a class, can sometimes do more than necessary, which results in an increment of the execution time. In this case a specialized

[2] URL: http://www.inetmi.com/~gwhi/ProfileViewer/ProfileViewer.html
[3] URL: http://www.physics.orst.edu/~bulatov/HyperProf/

version of the class can be written or a subclass could be defined which overrides the method with a more efficient one.

- Whenever possible use classes from the JAVA API when they offer native machine performance that cannot be matched by using JAVA. For example, the `arraycopy()` method is much faster than using a loop to copy an array.
- Instead of using the `String` concatenation operator `+`, the use of a `StringBuffer` should be considered, since the `String` concatenation operator involves a lot of work. First a new `StringBuffer` is created, next the two arguments are added to it with the `append()` method, and finally the result is converted back with the `toString()` method.

The last technique mentioned, of the `String` concatenation operator `+`, could have been used in the models discussed throughout this thesis, but is not, since this technique would have affected the readability of the code too much.

These techniques are certainly not the only other techniques which could be used to optimize JAVA code. There are numerous other techniques, which could not all be mentioned in this thesis. The techniques, which are mentioned here, are the most commonly used in the optimization of JAVA code.

Conclusion

In the following part the conclusions regarding this graduation project are discussed. Furthermore, a number of suggestions for further research on the use of the *Advance Action Approach* in general, and particularly in the JAVA programming language, are discussed.

The *Advance Action Approach* provides a way to run applications apparently at constant speed, irrespective, to some extend, of available CPU time and speed of the CPU. Programs, that make use of the *Advance Action Approach*, compute their evolution according to certain control laws, for example differential equations which describe the motion of a certain model. The results of these differential equations can be approximated by various approximation methods, of which some are more suitable than others, depending on the kind of problem, the accuracy of the result, and how fast the result is required.

The implementation of the models, discussed throughout this thesis, in the JAVA programming language provides a way to use the differential equations, that describe the dynamics of the models, directly in the code without rewriting them. Furthermore, the structure of the classes used for approximation allows a choice for one of the provided approximation methods, and even allows other approximation methods, than the ones provided, to be implemented.

The models of the pendulum, the bouncing ball, and the double pendulum are typical hybrid models, which means that they combine discrete and continuous dynamics. Discontinuities in the control laws can be caused by (discrete) internal or external events. The latter are easier to handle. They are simply handled as fast as possible. In the models containing a pendulum, the Tap event is a typical external event. The class that implements the user interface handles an event from the Tap button as fast as possible by adjusting the angular velocity of the pendulum, taking into account the momentum which is indicated by the scrollbar.

The internal events are more difficult to handle. The conditions for such an event to happen should be considered thoroughly. This may be quite an effort, as was the case in deriving the condition for the detection of a collision of the double pendulum. Further, during every step of the approximation the conditions of the possible events have to be evaluated to decide whether action has to be taken. Furthermore, the position of this evaluation in the code has to be considered, since an inappropriate position may lead to inappropriate actions, and thus to unwanted results.

It is also possible that the detection of an event is missed, due to a stepsize of the approximation that is too large. It is therefore recommendable to use the smallest possible stepsize,

even more because of the fact that the results of the approximation are more accurate using a small stepsize. By using a *Just In Time* (JIT) compiler the execution can be increased 10 to 30 times, which enables a smaller stepsize. This *Just In Time* (JIT) compiler can be plugged in without any other work or side effects. However, missing an event cannot be completely avoided. It is only possible to diminish the chances for this to occur.

Sometimes complicated calculations, which take a large amount of time, have to be performed at the moment of an event. Since an event should be instantaneous, it is recommended to rewrite the calculation to a less timeconsuming one, or to make certain assumptions to decrease the time that is needed, as was done for the collision event of the double pendulum.

The main problems, which have been encountered during the course of this graduation project, were the handling of the `repaint()` request, for which an appropriate solution has been introduced, and the `currentTimeMillis()` method in the Windows 95 port of the Java Development Kit. The latter might be solved by writing a timer in native code, which is updated with a preferred frequency of $1000\ Hz$. The disadvantage of this is that the platform independency is lost. At the moment the Java Development Kit provides no solution for this problem.

As an interpreted language, executing bytecode instead of natively compiled code, JAVA has a reduced performance compared to many other languages. However, it is possible to improve the performance of JAVA code by using optimization techniques such as updating of only the relevant parts of the image, and the monitoring of the `repaint()` and the `update()` method. Further, there are numerous low-level optimization techniques, which can substantially improve the performance of JAVA code.

When the results of this graduation project are considered some suggestions can be given for further research. Since this graduation project has not focussed on changing of priorities, it may be worthwhile to investigate whether this can contribute to extend the usability of the *Advance Action Approach*. Another suggestion for further research is to investigate how the *Advance Action Approach* can best be combined with parallellism, since this is not clear yet. Therefore, it is recommended to do research on, for example, a JAVA Virtual Machine that can spread threads across multiple processors, and the use of a restructuring compiler, such as JAVAR.

Finally, some suggestions for applications of the *Advance Action Approach* can be given. There is a wide range of possibilities in the entertainment industry. For example, a version of *Tetris* has been written, in which the *Advance Action Approach* was used. Further, the *Advance Action Approach* enables simulations to be visualised, which means that the output is not given in numbers and graphs, but in a *real-time* movie-like simulation. This may be of interest in education.

Bibliography

[1] B. Jacobs.
Coalgebraic specifications and models of deterministic hybrid systems.
In M. Wirsing and M. Nivat, editors, *Algebraic Methods and Software Technology*, pages 520–535.
Springer, 1996.

[2] B. Jacobs.
Object-oriented hybrid systems of coalgebras plus monoid actions.
Theoretical Computer Science, To appear.

[3] Sun Institute.
Java Programming, Student Guide, 1997.

[4] M. Campione and K. Walrath.
The Java Tutorial: Object-Oriented Programming for the Internet.
Addison-Wesley, 1997.

[5] M. Morrison et al.
Java 1.1 Unleashed.
Sams.net Publishing, third edition, 1997.

[6] A. Burns and A. Wellings.
Real-Time Systems and Programming Languages.
Addison-Wesley, second edition, 1997.

[7] K. Schengili-Roberts.
Oxford Dictionary of Computing.
Oxford University Press, 1989.

[8] E. Kreyszig.
Advanced Engineering Mathematics.
Wiley, 1962.

[9] E. Kreyszig.
Advanced Engineering Mathematics.
Wiley, seventh edition, 1993.

[10] G. Dahlquist and Å. Björck.
Numerical Methods.
Series in Automatic Computation. Prentice-Hall, 1974.
Translated by N. Anderson.

[11] F.A. Willers.
Practical Analysis.
Dover, 1948.

[12] H. Betz, P.B. Burcham, and G.M. Ewing.
Differential Equations with Applications.
Harper & Row, second edition, 1964.

[13] F. Verhulst.
Nietlineaire Differentiaalvergelijkingen en Dynamische Systemen.
Epsilon, 1985.

[14] M.W. Hirsch and S. Smale.
Differential Equations, Dynamical Systems, and Linear Algebra.
Academic Press, 1974.

[15] H. Goldstein.
Classical Mechanics.
Addison-Wesley, second edition, 1980.

[16] M. Alonso and E.J. Finn.
Fundamental University Physics, volume I Mechanics.
Addison-Wesley, 1967.

[17] M. Andersson.
Object-Oriented Modeling and Simulation of Hybrid Systems.
PhD thesis, Department of Automatic Control, Lund Institute of Technology, 1994.

[18] M. Andersson.
OmSim and Omola Tutorial and User's Manual.
Department of Automatic Control, Lund Institute of Technology, 1995.

[19] S.E. Mattsson and M. Andersson.
The ideas behind omola.
In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design*, pages 23–29, 1992.

[20] E. Hairer, C. Lubich, and M. Roche.
The numerical solution of differential-algebraic systems by runge-kutta methods.
In *Lecture Notes in Mathematics*, volume 1409. Springer, 1989.

[21] K.E. Brenan, S.L. Campbell, and L.R. Petzold.
Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.
North-Holland, 1989.

[22] K. Gustafsson.
An object oriented implementation of software for solving ordinary differential equations.
In *Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 318–330, 1993.

[23] T. Lindholm and F. Yellin.
The Java Virtual Machine Specification.
Addison-Wesley, 1996.

[24] C.A.R. Hoare.
Monitors: An operating system structuring concept.
Communications of the ACM, pages 549–557, 1974.

Index