TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

D I P L O M A R B E I T

Implementing the Markovian Model in Life Insurance

Ausgeführt am Institut für
Wirtschaftsmathematik
Forschungsgruppe für Finanz- und Versicherungsmathematik
der Technischen Universität Wien

unter der Anleitung von
O.Univ.Prof. Mag.rer.soc.oec. Dr.phil. Walter Schachermayer

durch
Elisabeth Stoißer
0107789
Rochusgasse 15a
1030 Wien

Wien, im März 2009

.....
Elisabeth Stoißer

Acknowledgements

This diploma thesis would not have been possible without the collaboration of the Institute of Mathematical Methods in Economics at the Technical University of Vienna and FJA Feilmeier & Junker Ges.m.b.H., a subsidiary of the IT software company FJA AG.
Special thanks go to Dipl.-Math. Axel Helmert, the managing director of FJA Feilmeier & Junker Ges.m.b.H. and Head of Business Division Mathematics of FJA AG, who also was the initiator of this project, and to O.Univ.Prof. Mag.rer.soc.oec. Dr.phil. Walter Schachermayer. They gave me the opportunity to write my diploma thesis on such a practice-oriented but also mathematically profound topic.
I also want to thank Dr. Christian Weber (FJA) for his continuous support concerning all mathematical questions and the practical application in C, as well as Gerhard Wondrak (FJA) for supporting me on any technical questions and problems.
Finally I want to express my gratitude to my parents Christiane and Dr. Bernd Stoißer and to my boyfriend Dipl.-Ing. Wolfgang Weirich who all gave me so much encouragement and support, as well as to Elisabeth Weirich whose treatments on the basis of Three In One Concepts always raised my spirits in times of exhaustion.

Abstract

By using the Markovian model in life insurance, life insurance contracts can be described through a stochastic process. The state of a contract at any time over the insurance period is given by the value of the corresponding path of the process at this time. Each of the so-called transition probabilities indicates the probability for the process to have a transition from one specified state to another one. As the stochastic process modelling the life insurance contracts is chosen to be a Markov process, future states of a concrete contract only depend on the corresponding path's present state but not on previously visited states. Benefits and contributions of individually defined amounts at individually specified points in time or even continuously effected payments can be defined for such a contract. For remaining in one state or for certain transitions the specified payments become due.
Besides precisely analysing the theoretical background of the Markovian model, the practical application of Thiele's differential equation constitutes an essential part of this diploma thesis. With the objective of being able to calculate the reserve for such individually created contracts, a routine has been written in the C programming language.

Kurzfassung

Die Anwendung des Markov-Modells in der Lebensversicherung erlaubt die Darstellung von Lebensversicherungsverträgen mittels eines stochastischen Prozesses, wobei der Zustand eines konkreten Vertrags zu jedem Zeitpunkt der Vertragslaufzeit durch den Wert des zugehörigen Pfades dieses Prozesses gegeben ist und den einzelnen Zustandsübergängen sogenannte Übergangswahrscheinlichkeiten zugeordnet werden. Der verwendete stochastische Prozess ist sogar ein Markovprozess, das heißt, die zukünftige Entwicklung des Prozesses hängt nur vom aktuellen Vertragszustand ab, nicht aber davon, wie der Prozess den derzeitigen Zustand erreicht hat. Für einen derart modellierten Vertrag können nun Leistungen und Beiträge, die bei Verbleiben des Vertrags in bestimmten Zuständen oder bei bestimmten Übergängen fällig werden, individuell definiert werden: Die Zahlungszeitpunkte sind frei wählbar beziehungsweise besteht die Möglichkeit stetig geleisteter Zahlungen, außerdem kann die Höhe jeder einzelnen Zahlung beliebig festgelegt werden.
Neben der genauen Analyse der zugrundeliegenden Theorie für das Markov-Modell beschäftigt sich diese Arbeit vor allem mit der praktischen Umsetzung der Thieleschen Differentialgleichung zur Berechnung des Reserveverlaufs für nach diesem Modell individuell gestaltete Lebensversicherungsverträge. Hierfür wurde ein Programm in der Programmiersprache C entwickelt.

Contents

Chapter 1

Introduction

For any insurance contract, the insurance company incurs certain defined expenses (declared insurance benefits and arising costs) but also collects a specified revenue (contributions paid by the insured person for the insurance coverage) over the insurance period of the contract. Both expenses and revenue are effected just under certain conditions (e.g. staying alive or having died) the performance of which can be described by probabilities. So the insurance company has certain expectations concerning the commitments it has to pay and the contributions it collects. These expectations at any time t during the insurance period are described by the prospective reserve: The prospective reserve is the difference between the expected discounted expenses and the expected discounted revenue. In other words, the reserve is the value the insurer has to provide to balance the difference between expected expenses and revenue at any point in time t over the insurance period.

The central theme of this diploma thesis is the calculation of the prospective reserve in life insurance. With the objective of considering very individually created contracts, i.e. contracts having benefits and contributions of individually defined amounts at individually specified points in time or even continuously effected payments, the following insurance model has been chosen:

In contrast to classical life insurance, in this model a finite set $S = \{1, \ldots, n\}$ describes the so-called *state space* consisting of the possible states for the considered insurance contract. So for T denoting the insurance period, at any time $t \in T$ the insured person or, more precisely, the contract is in one state $i \in S$.
A simple example is $S = \{*, \dagger\}$ where $*$ denotes the state "alive" and $\dagger$ the state "dead". $S = \{*, \diamond_1, \diamond_2, \ldots, \diamond_m, \dagger\}$ constitutes a more complex state space with $*$ signifying the state

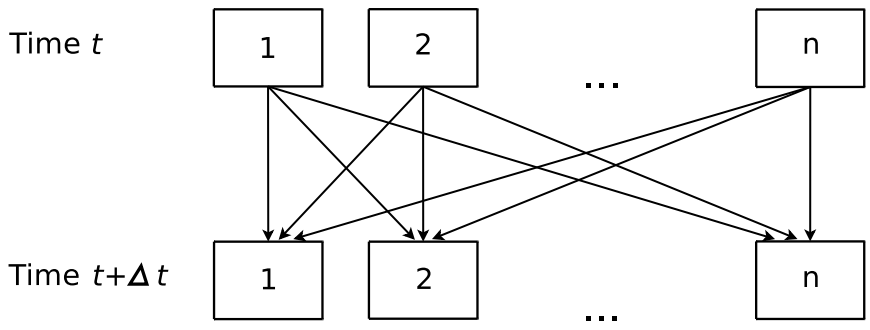

Figure 1.1: States of an insurance contract changing between t and Δt

"active", $\diamond_k$ for $k \in \{1, \ldots, m\}$ denoting the state "invalid for the k-th year" and † describing the state "dead" as before.
When using S as the set of individually defined states also the following state space is conceivable: $S = \{**, *\dagger, \dagger*, \dagger\dagger\}$ represents the set of the possible states of a contract for two insured persons. In this case, the elements of S are composed of all possible combinations of the two persons' single states $*$ ("alive") and † ("dead").
Figure 1.1 shows possible changes of a contract's state from time t to $t + \Delta t$ for $S = \{1, \ldots, n\}$.

The state of the contract at any time t over the insurance period is represented by a stochastic process $\{X_t(\omega)\}_{t \in T}$ with values in S. Transitions between the states are described by probabilities. The contractual payments are defined depending on the states: For remaining in one state or for certain transitions, the specified payments become due.

The stochastic process representing the contract's state model shall have a special quality: Assuming that a future state of the contract only depends on the present state but not on previous states is an intelligent and useful possibility of modelling an insurance contract. A stochastic process fulfilling this property is called *Markov process*. So this special quality leads to the expression *Markovian model in life insurance* which describes a stochastic life insurance model with a Markov process representing the corresponding state model.

Considering contracts with payments at individually specified points militates for a time-continuous insurance model. Any discretisation with regard to the payment times leading to a time-discrete model would reduce the degree of individuality for the contracts.

Furthermore, a time-continuous model allows the definition of time-continuous payments which are not so relevant for practical application but very important for theoretical considerations and for comparative examples sheding light on the correlation of continuous and different discrete payments.

Besides introducing the relevant theoretical aspects of the Markovian life insurance model, an essential part of this diploma thesis is the practical application of the deduced formulas. For this reason, one chapter is dedicated to the required numerical methods and in two chapters the important considerations for practical application are described and some examples are given.

The structure of this diploma thesis is as follows: Chapter 2 is dedicated to the Markov process after which not only the Markovian life insurance model but also the title of this diploma thesis is named. Starting from a stochastic process, all the relevant properties of the Markov process are described in this chapter which prepares for using the Markovian model as the considered stochastic life insurance model in Chapter 3. There, the construction of the Markovian model is successively deduced and explained in order to arrive at a formula for calculating the prospective reserve. As this formula is constituted by an ordinary differential equation, Chapter 4 deals with the different methods for numerically solving ordinary differential equations. My contribution to this diploma thesis can be found in Chapter 5 and 6: In Chapter 5, the important considerations for implementing Thiele's differential equation as well as the structure of the created routine are explained and some examples are given. Chapter 6 deals with the integrated consideration of the character of an insurance contract and explains the arising changes with respect to both the theoretical background and the practical application. Appendix A contains the source code of the created routine.

Chapter 2

Markov Processes

As it has already been mentioned in Chapter 1, a stochastic life insurance model defined for a finite set of states

$$S = \{1, \ldots, n\},\ n \in \mathbf{N} \tag{2.1}$$

constitutes the starting point for this diploma thesis. In this model, a stochastic process with values in S describes the state of the insured person or of the insurance contract over the insurance period. Transitions between the states are characterised by probabilities.
In this chapter, based on parts of chapter 2 in [Koller, 2000], important definitions and properties of Markov processes are described. So this chapter prepares for using the Markovian model as the considered stochastic life insurance model which is explained in chapter 3. Basic knowledge in measure theory and probability theory is assumed.

Definition 2.1 (**Stochastic process**). *Let $(\Omega, \mathcal{A}, \mathcal{P})$ be a probability space, $(S, \mathcal{S})$ a measurable space and T a set. Then, a family $\{X_t\}_{t \in T}$ of random variables $X_t : \Omega \to S$, $t \in T$, i.e.*

$$X : \Omega \times T \to S,\ (\omega, t) \mapsto X_t(\omega)$$

constitutes a stochastic process *on $(\Omega, \mathcal{A}, \mathcal{P})$ with* state space *S.*
For every $\omega \in \Omega$,

$$X(\omega) : T \to S,\ t \mapsto X_t(\omega)$$

defines a trajectory *or* path. *These paths are assumed to be* càdlàg *(continuous on the right, limit on the left).*

2.1 The Markov process and its fundamental characteristics

In the following, $S \neq \emptyset$ will be a countable set and $T \subset \mathbf{R}$ will denote an interval.
A Markov process is a special stochastic process fulfilling the Markov property:

Definition 2.2 (**Markov process**). *Let $\{X_t\}_{t \in T}$ denote a stochastic process on $(\Omega, \mathcal{A}, \mathcal{P})$ with values in S. X is called* Markov process *if the* Markov property

$$P[X_{t_{n+1}} = i_{n+1} | X_{t_1} = i_1, \ldots, X_{t_n} = i_n] = P[X_{t_{n+1}} = i_{n+1} | X_{t_n} = i_n] \tag{2.2}$$

holds $\forall\, n \in \mathbf{N}$, $t_1 < \ldots < t_{n+1} \in T$ and $i_1, \ldots, i_{n+1} \in S$ with $P[X_{t_1} = i_1, \ldots, X_{t_n} = i_n] > 0$.

The right-hand side of (2.2) clearly shows that this probability only depends on the last visited state X_{t_n} but not on the states of the process before t_n.

Now, several properties of this special stochastic process are described, starting with the definition of the transition probabilities.

Definition 2.3 (**Transition probabilities**). *For a Markov process $\{X_t\}_{t \in T}$ with values in S, the* transition probability *from state i at time s to state j at time t, $s \leq t$, $i, j \in S$ for $P[X_s = i] > 0$ is the conditional probability*

$$p_{ij}(s,t) := P[X_t = j | X_s = i] \tag{2.3}$$

For $P[X_s = i] = 0$, $p_{ij}(s,t)$ is defined to be 0.

On the one hand, when choosing a mathematical model to describe certain phenomena in reality, a desired quality of the model is to reflect the considered situations as accurately as possible. On the other hand, the model should be quite easy to handle in a mathematical sense.
For the case of the Markov process modelling life insurance contracts, the desired property of mathematical simplicity is fulfilled as the Markovian model allows easy computation of relevant probabilities and expected values. Nevertheless, the Markovian model is very suitable for describing a life insurance contract with its different states and possible transitions. A special quality of the Markov process is that it allows the treatment of contracts

with both continuous and discrete payment functions.
Sometimes the information about the last visited state of the contract is not sufficient for calculating required probabilities (e.g. when thinking of the state "invalid" for which the probabilities of dying and reactivating depend on how long the contract has already been in this state). In such a case, either the Markovian model with differently defined states (e.g. "invalid for the k-th year", $k = 1, \ldots, m$) can be applied or a completely different model has to be used.

The following theorem describes the composition of the transition probabilities when taking into account the state of the process at an intermediate point $t \in [s, u]$.

Theorem 2.4 (**Chapman-Kolmogorov equation**). *Let $\{X_t\}_{t \in T}$ be a Markov process with values in S, s, t, $u \in T$ with $s \leq t \leq u$ and i, $k \in S$ with $P[X_s = i] > 0$. Then*

$$p_{ik}(s, u) = \sum_{j \in S} p_{ij}(s, t) p_{jk}(t, u) \tag{2.4}$$

Proof. To prove Theorem 2.4, the set

$$S^* := \{j \in S : P[X_t = j | X_s = i] > 0\} = \{j \in S : P[X_t = j, X_s = i] > 0\} \tag{2.5}$$

is used instead of S to avoid division by 0. Equation (2.5) holds because $P[X_s = i] > 0$.

$$\begin{aligned}
p_{ik}(s, u) &= P[X_u = k | X_s = i] \\
&= \sum_{j \in S^*} P[X_u = k, X_t = j | X_s = i] && (2.6) \\
&= \sum_{j \in S^*} P[X_u = k | X_t = j, X_s = i] P[X_t = j | X_s = i] \\
&= \sum_{j \in S^*} P[X_u = k | X_t = j] P[X_t = j | X_s = i] && (2.7) \\
&= \sum_{j \in S^*} p_{jk}(t, u) p_{ij}(s, t) \\
&= \sum_{j \in S} p_{ij}(s, t) p_{jk}(t, u) && (2.8)
\end{aligned}$$

(2.6) holds because at an intermediate point t, $s \leq t \leq u$, the process is in any state $j \in S^*$. $j \in S \setminus S^*$ does not make any contribution because $P[X_t = j | X_s = i] = 0$ implies $P[X_u = k, X_t = j | X_s = i] = 0$. In (2.7), $P[X_u = k | X_t = j, X_s = i]$ is replaced

by $P[X_u = k|X_t = j]$ because of the Markov property. In (2.8), S^* can be replaced by S again because $p_{ij}(s,t) = 0$ for $j \in S \setminus S^*$. □

Definition 2.5 (**Transition matrix**).

$$\{P(s,t)\}_{s,t\in T,\, s\le t} = \left\{ \begin{pmatrix} p_{11}(s,t) & p_{12}(s,t) & \dots & p_{1n}(s,t) \\ p_{21}(s,t) & p_{22}(s,t) & \dots & p_{2n}(s,t) \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1}(s,t) & p_{n2}(s,t) & \dots & p_{nn}(s,t) \end{pmatrix} \right\}_{s,t\in T,\, s\le t} \tag{2.9}$$

is called a family of stochastic transition matrices *if the following four characteristics are fulfilled* $\forall\, s,\, t \in T,\, s \le t$*:*

- $p_{ij}(s,t) \ge 0 \quad \forall\, i,\, j \in S$
- $\sum_{j\in S} p_{ij}(s,t) = 1 \quad \forall\, i \in S$
- $p_{ij}(s,s) = \delta_{ij} \quad \forall\, i,\, j \in S$ *with* δ_{ij} *denoting the* Kronecker delta $\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \ne j \end{cases}$
- $p_{ik}(s,u) = \sum_{j\in S} p_{ij}(s,t)p_{jk}(t,u) \quad \forall\, i,\, k \in S,\ s,\, t,\, u \in T,\ s \le t \le u$
  which is equal to $P(s,u) = P(s,t)P(t,u)$ *in matrix notation.*

In the definition above, the property of being a stochastic matrix results from the facts that all elements of the matrix are non-negative and that all of its rows sum to unity. For $s = t$, $P(s,s)$ is equal to the identity matrix $\begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix}$, in the following referred to as I_n.

Definition 2.6 (**Transition matrix for the Markov process**). *A Markov process has transition matrices* $P(s,t)$ *with* $s,\, t \in T,\ s \le t$ *if*

$$P[X_t = j|X_s = i] = p_{ij}(s,t)$$

holds $\forall\, s,\, t \in T,\ s \le t$ *and* $i,\, j \in S$ *with* $P[X_s = i] > 0$.

2.2 Continuous time Markov processes

With the aim of being able to treat any individually customised life insurance contract, a time-continuous consideration is required. So continuous time Markov processes are discussed in this section, starting with the introduction of the so-called transition intensities.

Definition 2.7 (**Transition intensities**). *Let* $\{P(s,t)\}_{s,t\in T,\, s\leq t}$ *be a family of transition matrices and* $\{X_t\}_{t\in T}$ *a Markov process having those transition probabilities.*
$\{P(s,t)\}_{s,t\in T,\, s\leq t}$ *and the Markov process are called* regular *if the* transition intensities

$$\mu_{ij}(t) = \lim_{\triangle t \searrow 0} \frac{p_{ij}(t,t+\Delta t)}{\Delta t} \quad \forall\, i,\, j \in S,\ i \neq j \tag{2.10}$$

and

$$\begin{aligned} \sum_{\substack{j\in S\\ j\neq i}} \mu_{ij}(t) &= \sum_{\substack{j\in S\\ j\neq i}} \lim_{\triangle t \searrow 0} \frac{p_{ij}(t,t+\Delta t)}{\Delta t} \\ &= \lim_{\triangle t \searrow 0} \frac{1-p_{ii}(t,t+\Delta t)}{\Delta t} \quad \forall\, i \in S \end{aligned}$$

exist $\forall\, t \in T$ *and if they are continuous in* t.

In Definition 2.7, the intensity denoted by $\mu_{ij}(t)$ refers to a transition from state i to state j at time t whereas $\sum_{\substack{j\in S\\ j\neq i}} \mu_{ij}(t)$ describes the intensity of leaving state i and passing over into any other state $j \neq i$.

The transition intensities $\mu_{ij}(t)$ can be read as the right-hand derivatives of the corresponding transition probabilities. Because $P(t,t) = I_n$,

$$\begin{aligned} \mu_{ij}(t) &= \lim_{\triangle t \searrow 0} \frac{p_{ij}(t,t+\Delta t) - p_{ij}(t,t)}{\Delta t} && (2.11)\\ &= \lim_{\triangle t \searrow 0} \frac{p_{ij}(t,t+\Delta t) - \delta_{ij}}{\Delta t} && (2.12)\\ &= \frac{\partial}{\partial u^+} p_{ij}(t,u)\bigg|_{u=t} \quad \forall\, i,j \in S \end{aligned}$$

holds for $i,\, j \in S$ and even $i = j$ with $\frac{\partial}{\partial u^+} f(u)$ denoting the right-hand derivative of f at u. As $\delta_{ij} = 0$ for $i \neq j$, obviously (2.11) and (2.12) are equivalent to (2.10) in Definition 2.7.

Although for all calculations in the remaining part of this chapter and in the next chapter only $\mu_{ij}(t)$, $i \neq j$ are of interest, also for $i = j$ in [Koller, 2000] an intensity is defined through

$$\mu_{ii}(t) = -\sum_{\substack{j \in S \\ j \neq i}} \mu_{ij}(t) \quad \forall\, i \in S, \tag{2.13}$$

completing the introduction of $\mu_{ij}(t)$, $j \neq i$.
The definition of $\mu_{ii}(t)$ in (2.13) can be explained by the following consideration:
When taking the sum of the intensities $\forall\, j \in S$ (including $j = i$), (2.12) leads to

$$\begin{aligned} \sum_{j \in S} \mu_{ij}(t) &= \sum_{j \in S} \lim_{\triangle t \searrow 0} \frac{p_{ij}(t, t+\Delta t) - \delta_{ij}}{\Delta t} \\ &= \lim_{\triangle t \searrow 0} \frac{\sum_{j \in S} p_{ij}(t, t+\Delta t) - 1}{\Delta t} \quad \forall\, i \in S \end{aligned} \tag{2.14}$$

which holds because a derivative and a finite sum can be interchanged.
$\sum_{j \in S} p_{ij}(t, t+\Delta t) = 1$ implies that (2.14) is equal to 0 which is equivalent to the definition of $\mu_{ij}(t)$ in (2.13).

For $i \neq j$,

$$\begin{aligned} \mu_{ij}(t)\Delta t &\approx p_{ij}(t, t+\Delta t) \\ &= P[X_{t+\Delta t} = j | X_t = i] \quad \text{for } P[X_t = i] > 0 \end{aligned} \tag{2.15}$$

can be taken as the infinitesimal transition probability $i \to j$ of the interval $[t, t+\Delta t]$. Another way of phrasing (2.15) is

$$p_{ij}(t, t+\Delta t) = \mu_{ij}(t)\Delta t + o(t) \tag{2.16}$$

where $o(x)$ signifies that for a function $f(x)$ being of order $o(x)$, $\lim_{x \searrow 0} \frac{f(x)}{x} = 0$.

$$\Lambda(t) = \begin{pmatrix} \mu_{11}(t) & \mu_{12}(t) & \dots & \mu_{1n}(t) \\ \mu_{21}(t) & \mu_{22}(t) & \dots & \mu_{2n}(t) \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{n1}(t) & \mu_{n2}(t) & \dots & \mu_{nn}(t) \end{pmatrix} \tag{2.17}$$

denotes the matrix containing the transition intensities with

$$\Lambda(t) = \lim_{\triangle t \searrow 0} \frac{P(t, t+\Delta t) - I_n}{\Delta t} = \frac{\partial}{\partial u^+} P(t, u)\Big|_{u=t}$$

To see how the transition probabilities and the transition intensities are related to each other, now the Kolmogorov forward and backward equations are formulated and proved:

Theorem 2.8 (**Kolmogorov differential equations**). *The* Kolmogorov forward equation

$$\frac{\partial}{\partial t}P(s,t) = P(s,t)\Lambda(t) \tag{2.18}$$

$$\frac{\partial}{\partial t}p_{ij}(s,t) = -p_{ij}(s,t)\sum_{\substack{k\in S\\ k\neq j}}\mu_{jk}(t) + \sum_{\substack{k\in S\\ k\neq j}}p_{ik}(s,t)\mu_{kj}(t) \quad \forall\, i,\, j \in S \tag{2.19}$$

and the Kolmogorov backward equation

$$\frac{\partial}{\partial s}P(s,t) = -\Lambda(s)P(s,t) \tag{2.20}$$

$$\frac{\partial}{\partial s}p_{ij}(s,t) = \sum_{\substack{k\in S\\ k\neq i}}\mu_{ik}(s)p_{ij}(s,t) - \sum_{\substack{k\in S\\ k\neq i}}\mu_{ik}(s)p_{kj}(s,t) \quad \forall\, i,\, j \in S \tag{2.21}$$

hold for a regular family $\{P(s,t)\}_{s,t\in T,\, s\leq t}$ *of transition matrices and a finite set* S*.*

The forward equation regards changes with respect to state j at time t whereas the backward equation refers to changes with respect to state i at time s. If the component-wise equations (2.19) and (2.21) are rewritten in differential form

$$d_t\, p_{ij}(s,t) = -p_{ij}(s,t)\sum_{\substack{k\in S\\ k\neq j}}\mu_{jk}(t)dt + \sum_{\substack{k\in S\\ k\neq j}}p_{ik}(s,t)\mu_{kj}(t)dt \quad \forall\, i,\, j \in S \tag{2.22}$$

$$d_s\, p_{ij}(s,t) = \sum_{\substack{k\in S\\ k\neq i}}\mu_{ik}(s)p_{ij}(s,t)ds - \sum_{\substack{k\in S\\ k\neq i}}\mu_{ik}(s)p_{kj}(s,t)ds \quad \forall\, i,\, j \in S, \tag{2.23}$$

the following interpretation taken from [Wolthuis, 1994] gives a good intuitive understanding:
Starting from state j at time s, the left-hand side of (2.22) can be interpreted as the change in the probability for the contract to be in state j over the interval $[t, t+\Delta t)$. The right-hand side can be read as the difference between the expected number of transitions over Δt from any state $k \neq j$ to state j and that from state j to any other state k, given that the contract is in state i at time s.
A similar interpretation can be given for (2.23) with respect to the change in the probability for the contract to be in state i over the interval $[s, s+\Delta s)$.

By means of the Kolmogorov forward and backward equations, $\{P(s,t)\}_{s,t\in T,\, s\le t}$ can be calculated for given $t \to \Lambda(t)$, $t \in T$.

Proof. To prove Theorem 2.8, first for both equations the difference quotients are built. After applying the Chapman-Kolmogorov equation (2.4), the limits for $\Delta t \to 0$ and $\Delta s \to 0$, respectively, are calculated.

- For $s \le t < t + \Delta t$ and $s,\ t,\ t + \Delta t \in T$,

$$\begin{aligned}
\frac{\partial}{\partial t}P(s,t) &= \lim_{\triangle t \searrow 0} \frac{P(s,t+\Delta t) - P(s,t)}{\Delta t} \\
&= \lim_{\triangle t \searrow 0} \frac{P(s,t)P(t,t+\Delta t) - P(s,t)P(t,t)}{\Delta t} \\
&= P(s,t) \lim_{\triangle t \searrow 0} \frac{P(t,t+\Delta t) - I_n}{\Delta t} \\
&= P(s,t)\Lambda(t)
\end{aligned}$$

- For $s < s + \Delta s \le t$ and $s,\ s + \Delta s,\ t \in T$,

$$\begin{aligned}
\frac{\partial}{\partial s}P(s,t) &= \lim_{\triangle s \searrow 0} \frac{P(s+\Delta s,t) - P(s,t)}{\Delta s} \\
&= \lim_{\triangle s \searrow 0} \frac{P(s+\Delta s,s+\Delta s)P(s+\Delta s,t) - P(s,s+\Delta s)P(s+\Delta s,t)}{\Delta s} \\
&= \lim_{\triangle s \searrow 0} \frac{I_n - P(s,s+\Delta s)}{\Delta s} P(s+\Delta s,t) \\
&= -\Lambda(s)P(s,t)
\end{aligned}$$

□

Chapter 3

Calculating the reserve for a stochastic life insurance model

In Chapter 2, several properties of Markov processes have been discussed and it has been mentioned to what extent a Markov process represents a suitable model for life insurance contracts.
In this chapter, the remaining components for creating a life insurance model are presented and analysed, with the aim of obtaining a formula for calculating the prospective reserve of life insurance contracts.
First, in a few lines the topic of interest and discounting is covered very briefly. A considerably more detailed discussion will take place for the payment functions specifying the benefits and contribution payments for life insurance contracts. Here, the stochastic model is successively built up: Starting with a deterministic payment stream and step-by-step including interest and the stochastic component finally leads to expectation values representing the prospective reserve of a contract.
The same structure of successively approaching the stochastic model can be found in [Koller, 2000] which serves as principal source for this chapter. The following three references which have been important sources when preparing for this diploma thesis represent further literature concerning stochastic life insurance models: In [Wolthuis, 1994], beside very detailed explanations also many examples help the reader get a comprehensive insight into the Markovian model in life insurance mathematics. A very detailed presentation of life insurance mathematics in general and in particular of using the Markovian model is given in [Milbrodt and Helbig, 1999]. Great importance is attached to profoundly dealing with the required fundamental theoretical principles of measure theory and analysis. In [Norberg, 2001], the basic facts are described in a very clear presentation which very well

completes the information given through the other chosen books.
As in Chapter 2, basic knowledge in measure theory and probability theory is assumed.

3.1 Interest and discounting

The interest is an important component of life insurance contracts. Choosing a certain interest or interest model for a life insurance contract affects the calculated discounted values and therefore the premiums paid by the insured person. A too low interest implies very high contribution payments whereas a too high interest can cause the insurer's ruin.

An interest model can either be deterministic or stochastic. Thiele's differential equation representing the central equation for calculating the prospective reserve of a life insurance contract is formulated at the end of this chapter. Because this equation requires deterministic interest the following definition only refers to the deterministic case:

Definition 3.1. *The deterministic function $\delta(t)$ describes the* force of interest *at time t. The corresponding* discounting *function is denoted by*

$$v(s,t) = exp\left(-\int_s^t \delta(\tau)d\tau\right) \tag{3.1}$$

For $s = 0$, instead of $v(0,t)$ the expression $v(t)$ will be used.

3.2 Deterministic payment functions

For every insurance contract, the contractual payments define at which point of time which amounts have to be paid, depending on the state of the contract or the specified transition at this time. Again, the state space is denoted by $S = \{1, \ldots, n\}$, $n \in \mathbf{N}$ and $T \subset \mathbf{R}$ signifies the insurance period.

In this section, only deterministic payment streams are considered which describe certain amounts that are paid for being in a specified state or for a transition from one state to another one. For now, any stochastic component describing the state of a contract over the time is left out. So the question of whether the payments effectively are realised or not and the influence of probabilities are not taken into account in this section.

To get a first insight in how payments will be defined in later parts of this chapter, I start with the following definition:

Definition 3.2.

- *Assume that the contract is in state $i \in S$ over $[0,t] \subset T$. Then $a_{ii}(t)$ denotes the sum of the payments realised in state i over the interval $[0,t]$.*
- *$a_{ij}(t)$ for $i,j \in S$, $i \neq j$ describes the payment which is realised for a transition from state i to state j at time $t \in T$.*

Notice that $a_{ii}(t)$ and $a_{ij}(t)$ differ in so far that for every t, $a_{ij}(t)$ is a single payment whereas $a_{ii}(t)$ signifies the accumulated payments until time t.

For example, the classical term insurance has the state space $S = \{*, \dagger\}$ consisting of the two states $* =$ "alive" and $\dagger =$ "dead". $a_{**}(t)$ decreases by the amount of the contribution payment at each contribution payment time. $a_{*\dagger}(t)$ represents the benefit paid if the insured person dies at time t. $a_{\dagger\dagger}(t)$ and $a_{\dagger *}(t)$ are equal to 0 $\forall\, t \in T$.

In order to analyse important properties of deterministic payment functions, for the remaining part of this section I will use a more generally defined function A which does not refer to any state of the contract.
The definition of this payment function first requires the introduction of the following property:

Definition 3.3 (**Function of bounded variation**). *Let $T \subset \mathbf{R}$ be an interval and $f : T \to \mathbf{C}$ a function with $\mathbf{C}$ denoting the set of all complex numbers. Then the* total variation *of f over T is defined through*

$$Va(f,T) := \sup\left\{\sum_{i=1}^{n} |f(b_i^{(n)}) - f(a_i^{(n)})|\right\}$$

where the supremum is taken over all partitions $a_1^{(n)} < b_1^{(n)} \leq a_2^{(n)} < b_2^{(n)} \leq \ldots \leq a_n^{(n)} < b_n^{(n)}$, $n \in \mathbf{N}$ of the interval T.
f is of bounded variation over T if $Va(f,T) < \infty$.

Definition 3.4 (**Deterministic payment stream**). *A right-continuous function*

$$A : T \to \mathbf{R},\ t \mapsto A(t)$$

of bounded variation is called deterministic payment function *or* deterministic payment stream.

In Definition 3.4, $A(t)$ can be read as the accumulated payments until time t. By using this interpretation, the successive approach to the full life insurance model will continue.

As A is a function of bounded variation, some useful properties now are explained:

- For $A : T \to \mathbf{R}$ there exists a uniquely defined signed measure on the Borel-σ-algebra $\sigma(T)$ of T which I also denote by A. For this measure,

$$A((a, b]) = A(b) - A(a)$$

- A can uniquely be written as the difference $A = A^+ - A^-$ of two right-continuous, non-decreasing finite functions A^+ and A^- which increase on disjoint sets. A^+ and A^- then uniquely define non-negative measures on $(T, \sigma(T))$.
  With respect to a life insurance contract, $A = A^+ - A^-$ can be interpreted as the difference between insurance benefits (A^+) and contributions (A^-), both regarded as positive amounts.

When considering a payment function as introduced in Definition 3.4, the *value of this payment stream* is of great interest. By using the discounting function (3.1) the present value of A can be calculated:

Definition 3.5 (**Present value**). *For a deterministic payment stream $A : [0, \infty) \to \mathbf{R}$, $t \geq 0$ and $v \in L^1(A)$, i.e. v integrable on A, the* value of A at time t *is*

$$V(t, A) := \frac{1}{v(t)} \int_{(0,\infty)} v(s) dA(s) = \int_{(0,\infty)} exp\left(\int_s^t \delta(\tau) d\tau\right) dA(s)$$

Under the same assumptions, the prospective value *of A at time t is*

$$V^+(t, A) := \frac{1}{v(t)} \int_{(t,\infty)} v(s) dA(s)$$

As A is assumed to be deterministic, the present values $V(t, A)$ and $V^+(t, A)$ are deterministic functions, too. By contrast, contingent payment streams will be considered in the next section. Then $V(t, A)$ and $V^+(t, A)$ will be random variables for which also the calculation of expectation values will be of interest.

3.3 Stochastic payment functions

The deterministic payment function A which has been defined in the previous section is furnished with a stochastic component and so becomes a stochastic payment stream in this section.

Definition 3.6.

- *A* stochastic payment function *or* stream *is a stochastic process* $\{A_t\}_{t \in T}$ *for which almost all paths are right-continuous and of bounded variation.*
- *Let* $\{A_t\}_{t \geq 0}$ *be a stochastic payment stream on* $(\Omega, \mathcal{A}, \mathcal{P})$ *and* $F : [0, \infty) \times \Omega \to \mathbf{R}$ *a bounded, product measurable function. Then, for almost all* $\omega \in \Omega$

$$(F \cdot A)_t(\omega) = \int_{(0,t]} F(s, \omega) dA_s(\omega) =: \int_0^t F dA \tag{3.2}$$

and in differential notation

$$d(F \cdot A) = F dA$$

For fixed $\omega \in \Omega$, $F(\cdot, \omega) : [0, \infty) \to \mathbf{R}$ is measurable and $A(\omega)$ can be split up into $A^+(\omega)$ and $A^-(\omega)$ with the corresponding non-deterministic and non-negative measures $A^+(\omega)$ and $A^-(\omega)$. Then

$$\int_{(0,t]} F(s, \omega) dA_s(\omega) = \int_{(0,t]} F(s, \omega) dA_s^+(\omega) - \int_{(0,t]} F(s, \omega) dA_s^-(\omega)$$

With the aim of more precisely describing the payment streams for life insurance contracts, the payment functions a_{ii} and a_{ij} defined in section 3.2 are picked up again:

- $a_{ii}(t)$ for $i \in S$ represents the accumulated payments for a contract that has been in state i until time t without interruption. In the following, $a_{ii}(t)$ is assumed to be of bounded variation which implies $a_{ii}(t) = \int_0^t da_{ii}(s)$.

- $a_{ij}(t)$ is the single payment effected at time t if a transition from state i to state j occurs at time t, $i, j \in S$, $i \neq j$.

In addition to that, two more functions are defined now:

Definition 3.7. *Let $\{X_t\}_{t\in T}$ be a stochastic process on $(\Omega, \mathcal{A}, \mathcal{P})$ with values in S. For $i \in S$,*

$$I_i(t,\omega) = \begin{cases} 1 & \text{if } X_t(\omega) = i \\ 0 & \text{if } X_t(\omega) \neq i \end{cases} \tag{3.3}$$

is the indicator function *with respect to the process $\{X_t\}_{t\in T}$ at time t.*
The function

$$N_{ij}(t,\omega) = \#\{\tau \in (0,t) : X_{\tau^-} = i \text{ and } X_\tau = j\} \tag{3.4}$$

describes the number of transitions from i to j of the process over the interval $(0,t)$.

When describing the payment stream of a life insurance contract, for every path ω $I_i(t,\omega)$ provides the information whether the contract is in state i at time t or not. Analogously, for $\omega \in \Omega$, $N_{ij}(t,\omega)$ indicates a transition from state i to state j of the process by increasing by 1 at time t.

Definition 3.8. *For a life insurance contract with state space S and payment functions $a_{ii}(t)$ and $a_{ij}(t)$ being of bounded variation,*

$$A_{ii}(t,\omega) = \int_{(0,t]} I_i(s,\omega)\, da_{ii}(s) \tag{3.5}$$

$$A_{ij}(t,\omega) = \int_{(0,t]} a_{ij}(s)\, dN_{ij}(s,\omega) \tag{3.6}$$

and

$$A(t,\omega) = \sum_{i\in S} A_{ii}(t,\omega) + \sum_{i\in S}\sum_{\substack{j\in S \\ j\neq i}} A_{ij}(t,\omega) \tag{3.7}$$

are the corresponding contractual non-deterministic payment streams.

Whereas $a_{ii}(t)$ and $a_{ij}(t)$ indicate deterministic payment functions for the stay in one state or for a certain transition, the payment streams $A_{ii}(t,\omega)$, $A_{ij}(t,\omega)$ and $A(t,\omega)$ in Definition 3.8 describe the effectively realised payments for one possible path ω of the stochastic process modelling the contract.

So $A_{ii}(t,\omega)$ represents the accumulated payments until t for those periods in $(0,t]$ in which the contract is in state i for the path ω. Analogously, for the path ω the value of $A_{ij}(t,\omega)$ is equal to the accumulated payments for effectively occurred transitions from i to j over $(0,t]$. $A(t,\omega)$ represents the whole payment stream including all states and all transitions for a path ω.

In Definition 3.8, formula (3.2) of Definition 3.6 is applied: When defining $A_{ii}(t,\omega)$, $I_i(s,\omega)$ corresponds to the product measurable function F in Definition 3.6 and $a_{ii}(s)$ represents the payment stream $A_s(\omega)$ which, in this special case, is deterministic. For $A_{ij}(t,\omega)$, $a_{ij}(s)$ describes the function F which, in this case, is not furnished with a stochastic component and $N_{ij}(s,\omega)$ corresponds to the stochastic payment stream $A_s(\omega)$.

In differential notation, (3.5)-(3.7) are written as

$$\begin{aligned} dA_{ii}(t,\omega) &= I_i(t,\omega)da_{ii}(t) && (3.8)\\ dA_{ij}(t,\omega) &= a_{ij}(t)dN_{ij}(t,\omega) && (3.9)\\ &\text{and} \\ dA(t,\omega) &= \sum_{i\in S} dA_{ii}(t,\omega) + \sum_{i\in S}\sum_{\substack{j\in S\\ j\neq i}} dA_{ij}(t,\omega) && (3.10) \end{aligned}$$

It is clearly evident that $dA_{ii}(t,\omega)$ and $dA_{ij}(t,\omega)$ in (3.8)-(3.10) are the values by which the commitments of the insurer with respect to the insured person are increased at time t for path ω.

The present value of a payment stream (cf. Definition 3.5) can also be determined for every path of a non-deterministic payment stream. As only the prospective consideration of payment streams is of interest for the remaining part of this chapter, I just give the formula of the prospective present value with respect to the stochastic payment stream $A(t) = \{A(t,\omega)\}_{\omega\in\Omega}$:

$$\begin{aligned} V^+(t,A) &:= \frac{1}{v(t)}\int_{(t,\infty)} v(s)dA(s) && (3.11)\\ &= \frac{1}{v(t)}\sum_{i\in S}\int_{(t,\infty)} v(s)dA_{ii}(s) + \frac{1}{v(t)}\sum_{i\in S}\sum_{\substack{j\in S\\ j\neq i}}\int_{(t,\infty)} v(s)dA_{ij}(s) && (3.12) \end{aligned}$$

Obviously, in (3.11) and (3.12) $V^+(t,A)$ is a random variable.

3.4 Expectation values leading to the total prospective reserve

In this section, expectation values for the stochastic functions defined in the previous section will be introduced in order to arrive at a formula for the prospective reserve.

Definition 3.9. *Let* $\mathbf{F} = \{\mathcal{F}_\mathbf{t}\}_{\mathbf{t}\geq\mathbf{0}}$ *be a filtration and let the stochastic payment stream* A *and the discounting function* v *be adapted to* $\mathbf{F}$*. Then*

$$V_\mathbf{F}^+(t, A) = \mathbf{E}[V^+(t, A)|\mathcal{F}_t] \tag{3.13}$$

is called the prospective reserve *for* A *at time* $t \geq 0$*.*
Let more precisely the contract be represented by a Markov process $\{X_t\}_{t\geq 0}$ *and let* $\mathbf{F} = \{\mathcal{F}_t\}_{t\geq 0}$ *be the natural filtration generated by* $\{X_t\}_{t\geq 0}$*, i.e.* $\mathcal{F}_t = \sigma(\{X_s\}_{s\leq t})\ \forall t \geq 0$*. Then*

$$V_i^+(t, A) = \mathbf{E}[V^+(t, A)|X_t = i] \tag{3.14}$$

defines the prospective reserve for the Markov process $\{X_t\}_{t\geq 0}$ *and the payment stream* A *if* $v(t)$ *is deterministic or only depends on* X_t*.*

Obviously, in (3.14) the Markov property (cf. Definition 2.2) is used: For a Markov process, the conditional expectation with respect to $\mathcal{F}_t$ does only depend on the state X_t at time t.

In the following, the prospective reserve $V_i^+(t, A)$ (cf. (3.14)) will be used to calculate the expected values of different discounted payment streams finally leading to a concrete formula for calculating the prospective reserve.

Because the expectation values in Definition 3.9 need not necessarily exist, the following definition constitutes the qualities required for an insurance model having well-defined prospective reserves:

Definition 3.10 (**Regular insurance model**)**.** *A* regular insurance model *is defined as a model with*

- $\{X_t\}_{t\geq 0}$ *being a regular Markov process with finite state space* S
- $a_{ii}(t)$, $a_{ij}(t) : [0, \infty) \to \mathbf{R}$ *being the deterministic contractual payment functions of bounded variation as defined in section 3.3.*

- $\delta_i(t)$ *being right-continuous functions of bounded variation modelling the force of interest*

In the following, it is assumed that $\delta(t) = \sum_{i \in S} I_i(t)\delta_i(t)$ (*Markovian force of interest*) for $t \geq 0$ with $I_i(t)$ denoting the indicator function.

Definition 3.11. *Let i, j, k be states in S.*
The prospective reserve for remaining in one state j over the period (t, ∞) given that $X_t = i$ is

$$V_i^+(t, A_{jj}) = \mathbf{E}\left[\frac{1}{v(t)} \int_{(t,\infty)} v(\tau) dA_{jj}(\tau) |\, X_t = i\right] \tag{3.15}$$

and the prospective reserve for transitions from state j to state k over (t, ∞) given that $X_t = i$ is

$$V_i^+(t, A_{jk}) = \mathbf{E}\left[\frac{1}{v(t)} \int_{(t,\infty)} v(\tau) dA_{jk}(\tau) |\, X_t = i\right] \tag{3.16}$$

Then, the total prospective reserve for a given state i of the contract at time t is composed of $V_i^+(t, A_{jj})$ and $V_i^+(t, A_{jk})$ for the different states:

$$V_i^+(t, A) = \sum_{j \in S} V_i^+(t, A_{jj}) + \sum_{j \in S} \sum_{\substack{k \in S \\ k \neq j}} V_i^+(t, A_{jk}) \tag{3.17}$$

With the aim of calculating $V_i^+(t, A_{jj})$ and $V_i^+(t, A_{jk})$ now the functions $dA_{ii}(t) = I_i(t) da_{ii}(t)$ and $dA_{ij}(t) = a_{ij}(t) dN_{ij}(t)$ describing the change of the stochastic payment streams $A_{jj}(t)$ and $A_{jk}(t)$ at time t are considered in more detail.

First, a theorem is formulated which afterwards allows calculating $V_i^+(t, A_{jj})$ and $V_i^+(t, A_{jk})$:

Theorem 3.12. *Let $\{X_t\}_{t \geq 0}$ be a regular Markov process and i, j, $k \in S$, $j \neq k$. Then*

$$\mathbf{E}\left[\int_{(t,\infty)} I_j(\tau) da_{jj}(\tau) | X_t = i\right] = \int_{(t,\infty)} p_{ij}(t, \tau) da_{jj}(\tau) \tag{3.18}$$

holds for a_{jj} being of bounded variation and

$$\mathbf{E}\left[\int_{(t,\infty)} a_{jk}(\tau) dN_{jk}(\tau) | X_t = i\right] = \int_{(t,\infty)} a_{jk}(\tau) p_{ij}(t, \tau) \mu_{jk}(\tau) d\tau \tag{3.19}$$

is valid for a_{jk} being a measurable function and $a_{jk}\mu_{jk}$ being integrable over $[t, \infty)$.

For the proof of (3.19) further considerations are required:
Assume a regular Markov process with intensities $\mu_{ij}(t)$, $i, j \in S$, $i \neq j$ and $\sum_{\substack{j \in S \\ j \neq i}} \mu_{ij}(t) \leq \lambda$. Furthermore assume a Poisson process having an intensity bounded by $|S|\lambda$ where $|\cdot|$ denotes the cardinality of a set. Then, such a Markov process can be constructed by means of the described Poisson process that way that the Markov process only has a point of discontinuity if the Poisson process has.

The following theorem shows an important property of a Poisson process:

Theorem 3.13. *For $N = \{N_t\}_{t \geq 0}$ being a Poisson process with intensity $\lambda > 0$,*

$$\lim_{t \searrow 0} \frac{1}{t} P[N_t \geq 2] = 0 \tag{3.20}$$

i.e. $P[N_t \geq 2] = o(t)$ for $t \searrow 0$.

For $t \geq 0$, N_t denotes the number of events that have occurred up to time t, and $\mathbf{P}[(N_{s+t} - N_s) = k] = \frac{e^{-\lambda t}(\lambda t)^k}{k!}$. As for a Poisson process the probability distribution function of $N_{s+t} - N_s$ is equal to that of N_t, Theorem 3.13 shows that there cannot be 2 or more events ("transitions") at any time t. When proving Theorem 3.12, this property will be used for a Markov process which is regarded as constructed by means of a Poisson process.

Proof.

$$P[N_t \geq 2] = 1 - P[N_t = 0] - P[N_t = 1] = 1 - e^{-\lambda t} - \lambda t e^{-\lambda t}$$

Then adding the two parts

$$\lim_{t \searrow 0} \frac{1 - e^{-\lambda t}}{t} = -\lim_{t \searrow 0} \frac{e^{-\lambda t} - 1}{t} = -\frac{d}{dt} e^{-\lambda t}\Big|_{t=0} = \lambda$$

and

$$\lim_{t \searrow 0} \frac{-\lambda t e^{-\lambda t}}{t} = -\lambda$$

gives the desired result. □

Now, the validity of (3.18) and (3.19) in Theorem 3.12 can be shown:

Proof.

- To prove equation (3.18), Fubini's theorem allows interchanging the integral over (t,∞) and the expectation value. So

$$\begin{aligned}\mathbf{E}\left[\int_{(t,\infty)} I_j(\tau)da_{jj}(\tau)|X_t=i\right] &= \int_{(t,\infty)} \mathbf{E}\left[I_j(\tau)da_{jj}(\tau)|X_t=i\right] \\ &= \int_{(t,\infty)} \mathbf{P}\left[X_\tau=j|X_t=i\right]da_{jj}(\tau) \\ &= \int_{(t,\infty)} p_{ij}(t,\tau)da_{jj}(\tau)\end{aligned}$$

- The proof of (3.19) is more complex:
  First, the proposition is proved for a simple function $a_{jk}=I_{(u,v]}$. Then it is explained how this result can be generalised for any integrable a_{jk} being of bounded variation. In a first step, $a_{jk}(\tau)=I_{(u,v]}(\tau)$ and $t\le u<v<\infty$. Because $\{X_\tau\}_{\tau\ge 0}$ is a regular Markov process, $\mu_{jk}:[0,\infty)\to[0,\infty)$ is continuous and therefore bounded on $[u,v]$ and $a_{jk}\mu_{jk}$ is integrable. By using $h(\tau):=\mathbf{E}[N_{jk}(\tau)|X_t=i]$,

$$\begin{aligned}\mathbf{E}\left[\int_{(t,\infty)} a_{jk}(\tau)dN_{jk}(\tau)|X_t=i\right] &= \mathbf{E}\left[\int_u^v 1dN_{jk}(\tau)|X_t=i\right] \\ &= \mathbf{E}\left[N_{jk}(v)-N_{jk}(u)|X_t=i\right] \\ &= h(v)-h(u)=\int_u^v h'(\tau)d\tau\end{aligned}$$

  So to prove that

$$\int_u^v h'(\tau)d\tau=\int_{(t,\infty)} a_{jk}(\tau)p_{ij}(t,\tau)\mu_{jk}(\tau)d\tau$$

  for $a_{jk}=I_{(u,v]}$ it has to be shown that

$$h'(\tau)=p_{ij}(t,\tau)\mu_{jk(\tau)}$$

  For this purpose, the difference quotient is used and then $\lim_{\Delta\tau\searrow 0}$ is applied:

$$\begin{aligned}\frac{h(\tau+\Delta\tau)-h(\tau)}{\Delta\tau} &= \frac{\mathbf{E}\left[\left(N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau)\right)|X_t=i\right]}{\Delta\tau} \\ &= \frac{1}{\Delta\tau}\sum_{l\in S}\mathbf{E}\left[I_{\{X_t=l\}}N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau)|X_t=i\right] \quad (3.21)\end{aligned}$$

  holds because the process is in any state $l\in S$ at time t.

$$\begin{aligned}(3.21) &= \frac{1}{\Delta\tau}\sum_{l\in S}\mathbf{E}\left[N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau)|X_t=i,X_\tau=l\right]\mathbf{P}\left[X_\tau=l|X_t=i\right] \\ &= \frac{1}{\Delta\tau}\sum_{l\in S}\mathbf{E}\left[N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau)|X_\tau=l\right]p_{il}(t,\tau) \quad (3.22)\end{aligned}$$

holds because of the Markov property.
In the expectation value of (3.22), $N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau)$ denote the number of transitions from state j to state k over $(\tau,\tau+\Delta\tau]$. When applying $\lim_{\Delta\tau\searrow 0}$, for every addend in (3.22)

$$\begin{aligned}
&\lim_{\Delta\tau\searrow 0}\frac{1}{\Delta\tau}\mathbf{E}\left[N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau)|X_\tau=l\right]\\
&\quad=\lim_{\Delta\tau\searrow 0}\frac{1}{\Delta\tau}\sum_{m=0}^{\infty}m\mathbf{P}\left[(N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau))=m|X_\tau=l\right]\\
&\quad=\lim_{\Delta\tau\searrow 0}\frac{1}{\Delta\tau}1\,\mathbf{P}\left[(N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau))=1|X_\tau=l\right]
\end{aligned}$$

because according to Theorem 3.13 more than one transition over $(\tau,\tau+\Delta\tau]$ is not possible for $\lim_{\Delta\tau\searrow 0}$. For the same reason, the sum in (3.22) reduces to the addend for which $l=j$ and so

$$\begin{aligned}
&\lim_{\Delta\tau\searrow 0}\frac{1}{\Delta\tau}\sum_{l\in S}\mathbf{P}\left[(N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau))=1|X_\tau=l\right]p_{il}(t,\tau)\\
&\quad=\lim_{\Delta\tau\searrow 0}\frac{1}{\Delta\tau}\mathbf{P}\left[(N_{jk}(\tau+\Delta\tau)-N_{jk}(\tau))=1|X_\tau=j\right]p_{ij}(t,\tau)\\
&\quad=\lim_{\Delta\tau\searrow 0}\frac{1}{\Delta\tau}\mathbf{P}\left[X_{\tau+\Delta\tau}=k|X_\tau=j\right]p_{ij}(t,\tau)\\
&\quad=p_{ij}(t,\tau)\mu_{jk}(\tau)
\end{aligned}$$

So for $a_{jk}=I_{(u,v]}$, the desired result has been shown. For step functions with half-open intervals $(u,v]$, the proposition holds as well because of linearity. All $B\in\mathcal{B}([t,\infty))$ for which the function $a_{jk}=I_B$ fulfills the proposition, constitute a monotone class. This class contains all finite unions of $(u,v]$, $t\le u<v<\infty$ and therefore (3.19) is valid for all $B\in\mathcal{B}([t,\infty))$. Linearity leads to the desired result for any step function. According to the monotone convergence theorem, the proposition holds for non-negative functions a_{jk}. $a_{jk}=a_{jk}^{+}-a_{jk}^{-}$ provides the result for any integrable function a_{jk}.

□

Due to Theorem 3.12, now $V_i^{+}(t,A_{jj})$ and $V_i^{+}(t,A_{jk})$ can be calculated:

Theorem 3.14. *Assume a regular insurance model with deterministic functions modelling the force of interest and $t\in[0,\infty)$. Then the prospective reserve for remaining in one state j over (t,∞) given that $X_t=i$ is*

$$V_i^{+}(t,A_{jj})=\frac{1}{v(t)}\int_{(t,\infty)}v(\tau)p_{ij}(t,\tau)da_{jj}(\tau) \tag{3.23}$$

and the prospective reserve for transitions from state j to state k over (t,∞) given that $X_t = i$ is

$$V_i^+(t, A_{jk}) = \frac{1}{v(t)} \int_{(t,\infty)} v(\tau) a_{jk}(\tau) p_{ij}(t,\tau) \mu_{jk}(\tau) d\tau \tag{3.24}$$

Proof. After having used Definition 3.11 as well as (3.8) and (3.9), (3.23) and (3.24) directly result from Theorem 3.12:

$$\begin{aligned} V_i^+(t, A_{jj}) &= \mathbf{E}\left[\frac{1}{v(t)} \int_{(t,\infty)} v(\tau) dA_{jj}(\tau) | X_t = i\right] \\ &= \mathbf{E}\left[\frac{1}{v(t)} \int_{(t,\infty)} v(\tau) I_j(\tau) da_{jj}(\tau) | X_t = i\right] \\ &= \frac{1}{v(t)} \int_{(t,\infty)} v(\tau) p_{ij}(t,\tau) da_{jj}(\tau) \end{aligned}$$

$$\begin{aligned} V_i^+(t, A_{jk}) &= \mathbf{E}\left[\frac{1}{v(t)} \int_{(t,\infty)} v(\tau) dA_{jk}(\tau) | X_t = i\right] \\ &= \mathbf{E}\left[\frac{1}{v(t)} \int_{(t,\infty)} v(\tau) a_{jk}(\tau) dN_{jk}(\tau) | X_t = i\right] \\ &= \frac{1}{v(t)} \int_{(t,\infty)} v(\tau) a_{jk}(\tau) p_{ij}(t,\tau) \mu_{jk}(\tau) d\tau \end{aligned}$$

□

(3.23) and (3.24) now allow calculating the total prospective reserve for an insurance contract which is a formula composed of the prospective reserves for the different partial commitments of the insurance company:

Theorem 3.15. *For a regular insurance model with deterministic functions modelling the force of interest and $t \in [0,\infty)$*

$$V_i^+(t) = \frac{1}{v(t)} \int_{(t,\infty)} v(\tau) \sum_{j \in S} p_{ij}(t,\tau) \left(da_{jj}(\tau) + \sum_{\substack{k \in S \\ k \neq j}} a_{jk}(\tau) \mu_{jk}(\tau) d\tau \right) \tag{3.25}$$

holds $\forall\, i \in S$.

Proof. Formula (3.10) and Theorem 3.14 are applied:

$$V_i^+(t) = \frac{1}{v(t)} \mathbf{E}\left[\int_{(t,\infty)} v(\tau) dA(\tau) | X_t = i\right]$$

$$
\begin{aligned}
&= \frac{1}{v(t)}\mathbf{E}\left[\int_{(t,\infty)} v(\tau)\sum_{j\in S}\left(dA_{jj}(\tau)+\sum_{\substack{k\in S\\k\neq j}} dA_{jk}(\tau)\right)|X_t=i\right]\\
&= \frac{1}{v(t)}\mathbf{E}\left[\int_{(t,\infty)} v(\tau)\sum_{j\in S}\left(I_j(\tau)da_{jj}(\tau)+\sum_{\substack{k\in S\\k\neq j}} a_{jk}(\tau)dN_{jk}(\tau)\right)|X_t=i\right]\\
&= \frac{1}{v(t)}\int_{(t,\infty)} v(\tau)\sum_{j\in S} p_{ij}(t,\tau)\left(da_{jj}(\tau)+\sum_{\substack{k\in S\\k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right)
\end{aligned}
$$

□

3.5 Thiele's differential equation

Starting from equation (3.25), a recurrence relation is derived in this section in order to simplify calculating $V_i^+(t)$. The recurrence relation allows using already calculated values $V_i^+(u)$, $u > t$ to determine $V_i^+(t)$. In a further step, based on the recurrence relation, Thiele's differential equation is introduced as a useful formula for practical application.

To simplify the proofs for the next two theorems, the total prospective reserve discounted to 0 will be used instead of $V_i^+(t)$:

Definition 3.16. *For a regular insurance model with deterministic functions modelling the force of interest,*

$$W_i^+(t) := v(t)V_i^+(t), \quad t \geq 0,\ i \in S \tag{3.26}$$

is defined as the total prospective reserve discounted to 0*.*

$V_i^+(t)$ is calculated with respect to the present value at time t of the stochastic payment stream whereas $W_i^+(t)$ refers to the present value at time 0. Because the interest is deterministic, it does not make any difference whether $W_i^+(t)$ is considered or $V_i^+(t)$. Dividing $W_i^+(t)$ by $v(t)$ leads back to $V_i^+(t)$ in any case and vice versa. For reasons of simplification, $W_i^+(t)$ will be referred to as the "total prospective reserve" as well. But when regarding the corresponding formulas, it will be clearly evident which of the two expressions $V_i^+(t)$ and $W_i^+(t)$ is considered.

Theorem 3.17. *For a regular insurance model with deterministic interest, the recurrence relation*

$$\begin{aligned} W_i^+(t) &= \sum_{j\in S} p_{ij}(t,u)W_j^+(u) \\ &\quad + \int_{(t,u]} v(\tau)\sum_{j\in S} p_{ij}(t,\tau)\left(da_{jj}(\tau) + \sum_{\substack{k\in S\\ k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right) \end{aligned} \tag{3.27}$$

In (3.27), the total prospective reserve $W_i^+(t)$ is split up into the prospective reserve for the two intervals $(t,u]$ and (u,∞). The second term on the right-hand side of (3.27) shows the expected payments discounted to 0 over $(t,u]$ whereas in the first term the expected payments for (u,∞) are described by the sum of the total prospective reserves $W_i^+(u)$ at time u multiplied by the corresponding transition probabilities. So $W_i^+(t)$ can be read as the expected discounted payments for a first interval $(t,u]$ plus the expected discounted reserve required for time u.

Proof. The central idea of this proof is to split up $W_i^+(t)$ into two integrals over $(t,u]$ and (u,∞) and to apply the Chapman-Kolmogorov equation (2.4) to the second integral:

$$\begin{aligned} W_i^+(t) &= \int_{(t,\infty)} v(\tau)\sum_{j\in S} p_{ij}(t,\tau)\left(da_{jj}(\tau) + \sum_{\substack{k\in S\\ k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right) \\ &= \int_{(t,u]} v(\tau)\sum_{j\in S} p_{ij}(t,\tau)\left(da_{jj}(\tau) + \sum_{\substack{k\in S\\ k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right) \\ &\quad + \int_{(u,\infty)} v(\tau)\sum_{j\in S} p_{ij}(t,\tau)\left(da_{jj}(\tau) + \sum_{\substack{k\in S\\ k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right) \\ &= \int_{(t,u]} v(\tau)\sum_{j\in S} p_{ij}(t,\tau)\left(da_{jj}(\tau) + \sum_{\substack{k\in S\\ k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right) \\ &\quad + \int_{(u,\infty)} v(\tau)\sum_{j\in S}\sum_{jl\in S} p_{il}(t,u)p_{lj}(u,\tau)\left(da_{jj}(\tau) + \sum_{\substack{k\in S\\ k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right) \\ &= \int_{(t,u]} v(\tau)\sum_{j\in S} p_{ij}(t,\tau)\left(da_{jj}(\tau) + \sum_{\substack{k\in S\\ k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right) \end{aligned}$$

$$+\sum_{l\in S} p_{il}(t,u)\underbrace{\int_{(u,\infty)} v(\tau)\sum_{j\in S} p_{lj}(u,\tau)\left(da_{jj}(\tau)+\sum_{\substack{k\in S\\ k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right)}_{W_l^+(u)}$$

□

The introduction of Thiele's differential equation is split up into two steps: First, the important formula is given with restriction to a prospective reserve without discontinuities having a simple proof based on the Chapman-Kolmogorov equation and the already considered formulas. Afterwards a more general version of Thiele's differential equation is described which allows a finite number of discontinuities such as all the formulas which have already been considered in this chapter. Proving the general formula is much more complex so that the full proof will not be given here.

Theorem 3.18 (**Thiele's differential equation I**). *Let* $\{X_t\}_{t\geq 0}$, a_{ii}, a_{ij} *and* $\{\delta_t\}_{t\geq 0}$ *constitute a regular insurance model with* a_{ij} *being continuous and* a_{ii} *being continuously differentiable* $\forall\, i,\, j\in S,\, i\neq j$.
Then, $\forall\, i\in S$ *the total prospective reserve* $W_i^+(t)$ *is continuous and its change for* $t\geq 0$ *is described by* Thiele's differential equation

$$\begin{aligned}\frac{d}{dt}W_i^+(t) &= -v(t)\left(a'_{ii}(t)-\sum_{\substack{j\in S\\ j\neq i}}\mu_{ij}(t)a_{ij}(t)\right)\\ &\quad+\left(\sum_{\substack{j\in S\\ j\neq i}}\mu_{ij}(t)\right)W_i^+(t)-\sum_{\substack{j\in S\\ j\neq i}}\mu_{ij}(t)W_j^+(t)\end{aligned}\tag{3.28}$$

with $a'_{ii}(t)=\frac{d}{dt}a_{ii}(t)$.

Proof. Starting from Theorem 3.17, the difference quotient for $W_i^+(t)$ is calculated and then $\lim_{\Delta t\searrow 0}$ is applied:

$$\begin{aligned}W_i^+(t) &= \underbrace{\sum_{j\in S} p_{ij}(t,t+\Delta t)W_j^+(t+\Delta t)}_{I}\\ &\quad+\underbrace{\int_{(t,t+\Delta t]} v(\tau)\sum_{j\in S} p_{ij}(t,\tau)\left(da_{jj}(\tau)+\sum_{\substack{k\in S\\ k\neq j}} a_{jk}(\tau)\mu_{jk}(\tau)d\tau\right)}_{II}\end{aligned}$$

holds according to Theorem 3.17 for $u = t + \Delta t$, $\Delta t > 0$. Splitting up the first term I into $i = j$ and $\sum_{j \in S,\, j \neq i}$ and using

$$p_{ij}(t, t + \Delta t) = \mu_{ij}(t)\Delta t + o(\Delta t), \quad i \neq j$$

and

$$p_{ii}(t, t + \Delta t) = 1 - \left(\sum_{\substack{j \in S \\ i \neq i}} \mu_{ij}(t) \right) \Delta t + o(\Delta t)$$

leads to

$$I = \left(1 - \left(\sum_{\substack{j \in S \\ i \neq i}} \mu_{ij}(t) \right) \Delta t \right) W_i^+(t + \Delta t) + \sum_{\substack{j \in S \\ i \neq i}} \mu_{ij}(t)\, \Delta t\, W_j^+(t + \Delta t) + o(\Delta t)$$

Because v, a_{jj}, a_{jk} and μ_{jk} are continuous functions, the integral in the second term II can be replaced by the value of the integrand at time t multiplied by Δt:

$$\begin{aligned} II &= \int_{(t,t+\Delta t]} v(\tau) \sum_{j \in S} p_{ij}(t, \tau) \left(a'_{jj}(\tau) + \sum_{\substack{k \in S \\ k \neq j}} a_{jk}(\tau)\mu_{jk}(\tau) d\tau \right) \\ &= v(t) \sum_{j \in S} p_{ij}(t, t) a'_{jj}(t)\Delta t + v(t) \sum_{j \in S} p_{ij}(t, t) \sum_{\substack{k \in S \\ k \neq j}} a_{jk}(t)\mu_{jk}(t)\Delta t + o(\Delta t) \\ &= v(t) a'_{ii}(t)\Delta t + v(t) \sum_{\substack{k \in S \\ k \neq i}} a_{ik}(t)\mu_{ik}(t)\Delta t + o(\Delta t) \end{aligned}$$

Inserting $W_i^+(t) = I + II$ into $\frac{W_i^+(t+\Delta t) - W_i^+(t)}{\Delta t}$ gives

$$\begin{aligned} \frac{W_i^+(t + \Delta t) - W_i^+(t)}{\Delta t} &= \left(\sum_{\substack{j \in S \\ i \neq i}} \mu_{ij}(t) \right) W_i^+(t + \Delta t) - \sum_{\substack{j \in S \\ i \neq i}} \mu_{ij}(t) W_j^+(t + \Delta t) \quad (3.29) \\ &\quad - v(t) \left(a'_{ii}(t) + \sum_{\substack{j \in S \\ k \neq i}} a_{ij}(t)\mu_{ij}(t) \right) + \frac{o(\Delta t)}{\Delta t} \end{aligned}$$

Applying $\lim_{\Delta t \searrow 0}$ leads to the desired result. □

With the objective of a combined treatment of contracts with continuous and discrete payment streams, a second version of Thiele's differential equation shall allow discontinuities

of the total prospective reserve. [Koller, 2000] provides a formula for a differential equation describing the change of the p-th moment

$$V_i^{(p)}(t) := \mathbf{E}[(V^+(t))^p | X_t = i],\ p \geq 1 \tag{3.30}$$

of $V^+(t)$. $V^+(t)$ there denotes the present value of the stochastic payment stream for a life insurance contract allowing discontinuities. By contrast to Thiele's differential equation I, [Koller, 2000] uses $V_i^p(t)$ instead of $W_i^p(t)$.
As for the remaining part of this diploma thesis only the (total) prospective reserve $V_i^+(t)$ or $W_i^+(t)$ is of interest, I just give the special case for $p = 1$ of the theorem describing the formula $\frac{d}{dt}V_i^{(p)}(t)$ in [Koller, 2000]:

Theorem 3.19 (**Thiele's differential equation II**). *Let $\{X_t\}_{t\geq 0}$ be a Markov process with μ_{ij} being piece-wise continuous functions $\forall\, i, j \in S, i \neq j$. Let a_{ii} be piece-wise continuously differentiable and a_{ij} and δ_i be piece-wise continuous $\forall\, i, j \in S, i \neq j$. Let $[0, T] \subset \mathbf{R}$ signify the insurance period of a life insurance contract and $D \subset (0, T)$ denote the finite set of discontinuities appearing in a_{ii}, a_{ij}, μ_{ij} or δ_i. Then Thiele's differential equation for the total prospective reserve $V_i^+(t)$ allowing discontinuities*

$$\begin{aligned} \frac{d}{dt}V_i^+(t) &= -a'_{ii}(t) - \sum_{\substack{j\in S\\ j\neq i}} \mu_{ij}(t)a_{ij}(t) \\ &\quad + \left(\delta_i(t) + \sum_{\substack{j\in S\\ j\neq i}} \mu_{ij}(t)\right) V_i^+(t) - \sum_{\substack{j\in S\\ j\neq i}} \mu_{ij}(t)V_j^+(t), \quad i \in S \end{aligned} \tag{3.31}$$

holds $\forall\, t \in (0, T) \setminus D$ and is completed by the conditions

$$V_i^+(t^-) = V_i^+(t) + da_{ii}(t), \quad i \in S \tag{3.32}$$

$\forall\, t \in D$.

Equation (3.31) is also valid for all points of discontinuity $t \in D$ if the differential notation $dV_i^+(t)$ is considered.

Proof. In [Koller, 2000], the proof of the complete theorem describing $\frac{d}{dt}V_i^{(p)}(t)$ for $p \geq 1$ can be found on p.71-75. □

The second version of Thiele's differential equation given through (3.31) and (3.32) provides a formula which allows calculating the total prospective reserve for a life insurance contract which very individually can be defined through the functions a_{ii}, a_{ij}, μ_{ij} and δ_i.

In order to prepare practical applications, the following chapter deals with the question which of the different numerical discretisation methods are qualified to calculate a solution of Thiele's differential equation.

Chapter 4

Numerical solutions for initial value problems in ordinary differential equations

The subject of this chapter is the numerical solution of an initial value problem (IVP) consisting of an ordinary first order differential equation and a corresponding initial value. Concerning the introduced numerical methods, only those are presented in detail which are relevant for solving *Thiele's differential equation* (cf. Section 5.1, equation (5.3)). In section 4.5 and 5.2, the reasons for choosing those methods are pointed out.

Principal sources of this chapter are [Auzinger, 2007], [Kloeden and Platen, 1992] and [Stetter, 1985]. A good description of the important methods is given in [Press et al., 2007] where great importance is attached to providing relevant information for the practical application of those methods. Further particular references are quoted at the corresponding positions in the text.

4.1 The initial value problem and its solvability

For the remaining part of this chapter, every considered IVP will be an *IVP of standard form* which is defined as follows:

Definition 4.1 (**Initial value problem of standard form**).

$$y' = f(t,y) \;\; and \;\; y(0) = y_0 \quad with \qquad f : G \subset \mathbf{R^{n+1}} \to \mathbf{R^n} \tag{4.1}$$

$$y, y', y_0 \in \mathbf{R^n} \tag{4.2}$$

$$n \in \mathbf{N}, \;\; \mathbf{t} \in [\mathbf{0}, \mathbf{T}] \subset \mathbf{R} \tag{4.3}$$

$$(0, y_0) \in G \tag{4.4}$$

In (4.1)-(4.4), the field line for $(t,y) \in G$ with $y' = f(t,y)$ representing the vectors in the field line is considered. A function $y(t)$ for which the field line holds shall be found. This solution function $y(t)$ is determined through the initial condition $y(0) = y_0$. So $f(t,y)$ represents the right-hand side of the differential equation with t being the independent variable and $y = y(t) = (y_1(t), \ldots, y_n(t))$, $n \in \mathbf{N}$ being the dependent variable and unknown solution. Although any desired integration interval $[a,b] \subset \mathbf{R}$ can be chosen, w.l.o.g. it is sufficient here to consider the integration interval $[0,T] \subset \mathbf{R}$.

The next paragraphs deal with two theorems that specify under which conditions an existing or a unique solution of an IVP of standard form can be ensured. Before phrasing the theorems it is useful to presume the following characteristics for the region G introduced in Definition 4.1:

- $G := \{(t,y) : \; |t| < \rho_1, \; ||y - y_0||_2 < \rho_2\}$ with $\|.\|_2$ symbolising the Euclidean norm in $\mathbf{R^s}$
- The closed envelope $\overline{G}$ of G is defined through $\overline{G} := \{(t,y) : \; |t| \leq \rho_1, \; ||y - y_0||_2 \leq \rho_2\}$.

Then, the *Peano Existence Theorem* says:

Theorem 4.2 (**Peano Existence Theorem**). *An IVP as defined in (4.1)-(4.4) with*

- *f being continuous in $\overline{G}$ and therefore being bounded there and*
- *$||f(t,y)||_2 \leq A(t)$ with $A(t)$ being an integrable function supplying $\int_0^t A(\tau)d\tau \leq \rho_2$ for all t with $|t| \leq \alpha$, $0 < \alpha \leq \rho_1$*

has at least one solution which exists for $|t| \leq \alpha$.

The *Picard-Lindelöf Theorem* even ensures the existence of a unique solution under the following conditions:

Theorem 4.3 (**Picard-Lindelöf Theorem**). *For an IVP as introduced in (4.1)-(4.4) with*

- f *being continuous in* G,
- $||f(t,y)||_2 \leq M$ *in* G *and*
- $\|f(t,y_1) - f(t,y_2)\|_2 \leq L\,\|y_1 - y_2\|$
  with Lipschitz constant $L \geq 0$ *and* (t,y_1), $(t,y_2) \in G$

there exists a single unique solution for $|t| < \alpha = \min\{\rho_1, \frac{\rho_2}{M}\}$.

The proofs of these two theorems for instance can be found in [Hartman, 1964], chapter II.
In section 5.1.4, the required conditions for the two theorems are discussed for Thiele's differential equation posed as an IVP which can facilitate understanding the meaning of the different requirements.

For the remaining part of this chapter, by using the expression *IVP* I will always refer to an IVP of standard form as specified in Definition 4.1 for which the existence of a unique solution over the considered integration interval is guaranteed.

4.2 The idea of approximating a solution of an IVP by using the explicit Euler method

The easiest way of numerically approximating a solution of an IVP is to split up the integration interval into equidistant sub-intervals and to replace the real solution curve by a linear function on each sub-interval. The whole exact solution curve then is approximated by a polygonal chain.

For this purpose, a constant step size h is chosen and the considered integration interval $[0,T]$ is split up into $m = \frac{T}{h}$ equidistant sub-intervals $[t_{k-1}, t_k]$, $k = 1, \ldots, m$, $m \in \mathbf{N}$ with $t_0 = 0$, $t_m = T$ and $t_k = t_0 + kh$.

In the following, $y(t_k)$ will always represent the exact solution at time t_k while y_k will signify the approximated solution at t_k. In general, only at the beginning of the considered integration interval $[0, T]$, the exact solution $y(0)$ and the approximated one y_0 are identically equal.

The right-hand side $f(t, y)$ of the differential equation describes the field line in G and $\overline{G}$ in which, according to the initial condition, the real solution curve $y(t)$ is embedded. In other words, $f(t, y)$ represents the slope of each solution curve $y(t)$ of the differential equation for $t \in [0, T]$.

The idea of the numerical calculation is to start in $(0, y_0)$ and to act as if the slope of the solution curve at $(0, y_0)$, i.e. $f(0, y_0)$, held for the whole sub-interval $[0, t_1]$. By doing this, the line segment in $[0, t_1]$ is calculated through $y_1 = y_0 + hf(0, y_0)$ with y_1 representing the approximation of $y(t)$ at $t = t_1$. Provided that $y(t)$ is not linear in $[0, t_1]$, (t_1, y_1) is not a point on the real solution curve $y(t)$ of the IVP but on another solution curve $\widetilde{y}(t)$ of the regarded differential equation having another initial value.
Having calculated y_1, (t_1, y_1) then is the initial value for the next step of numerically solving the IVP. Again, $f(t_1, y_1)$ is "frozen" for the sub-interval $[t_1, t_2]$ in order to obtain the line segment for this interval.
The procedure described above is exactly what the explicit Euler scheme does. So the formula for the *explicit Euler method* is:

$$y_0 = y(0) \tag{4.5}$$

$$y_k = y_{k-1} + hf(t_{k-1}, y_{k-1}), \quad k = 1, \ldots, m \tag{4.6}$$

The explicit Euler scheme is the simplest one of the one-step methods for solving ordinary differential equations. A *one-step method* calculates the approximated solution for each sub-interval by only using information of this sub-interval. In the case of the explicit Euler method, this information consists of the initial value for the regarded sub-interval $[t_{k-1}, t_k]$ and the right-hand side of the differential equation at this point.
By contrast, a *multistep method* refers to several previous function values y_{k-1}, y_{k-2}, ... when calculating y_k (cf. section 4.5).

When numerically approximating a solution of an IVP by means of the explicit Euler method or any other discretisation method in each step a certain error occurs. To describe these errors I introduce two terms:

Definition 4.4 (**Local and global discretisation error**).

- *The* local discretisation error l_k *indicates the error that occurs when calculating the k-th step of the discretisation:*

$$l_k \;:=\; y(t_k; t_{k-1}, y_{k-1}) - y_k, \quad k = 1, \ldots, m \tag{4.7}$$

  with $y(t_k; t_{k-1}, y_{k-1})$ *representing the exact solution at* t_k *of the differential equation starting from* (t_{k-1}, y_{k-1}).

- *The* global discretisation error e_k *signifies how much the approximated solution after the k-th step differs from the exact solution of the original IVP:*

$$e_k := y(t_k; t_0, y_0) - y_k, \quad k = 1, \ldots, m \tag{4.8}$$

Remark 4.5. *In literature, the local discretisation error sometimes is defined different from (4.7):*

$$\widetilde{l}_k := \frac{y(t_k; t_{k-1}, y_{k-1}) - y(t_{k-1})}{h} - f(t_{k-1}, y_{t_{k-1}}), \quad k = 1, \ldots, m \tag{4.9}$$

This definition refers to the difference between the real difference quotient of the interval $[t_{k-1}, t_k]$ *and the slope in* $(t_{k-1}, y_{t_{k-1}})$. *The relation between* l_k *and* $\widetilde{l}_k$ *simply is*

$$l_k = h\,\widetilde{l}_k \tag{4.10}$$

Notice that different definitions of the local discretisation error also have effect on further properties of a discretisation method introduced in the following. For example, a consistency order p (cf. (4.11)) calculated for l_k *is equal to a consistency order of* $p - 1$ *for* $\widetilde{l}_k$.
The definition of the local error l_k *as in (4.7) can be found in [Kloeden and Platen, 1992] whereas, for instance, in [Auzinger, 2007] the used definition is the one for* $\widetilde{l}_k$ *as in (4.9).*

Figure 4.1 shows the first two steps of the explicit Euler method as well as the local and global errors of those two steps for an IVP of dimension 1. The function values of the real solution curve $y(t)$ at time t_1 and t_2 in the figure are denoted by $y(t_1)$ and $y(t_2)$ as an abbreviation of the expressions $y(t_1;\, t_0, y_0)$ and $y(t_2;\, t_0, y_0)$ in Definition 4.4. Furthermore, $\widetilde{y}(t_2)$ is used instead of $y(t_2;\, t_1, y_1)$

When applying a numerical method it is preferable that its accuracy can be augmented by increasing the computational effort. In other words, calculating with a smaller step size

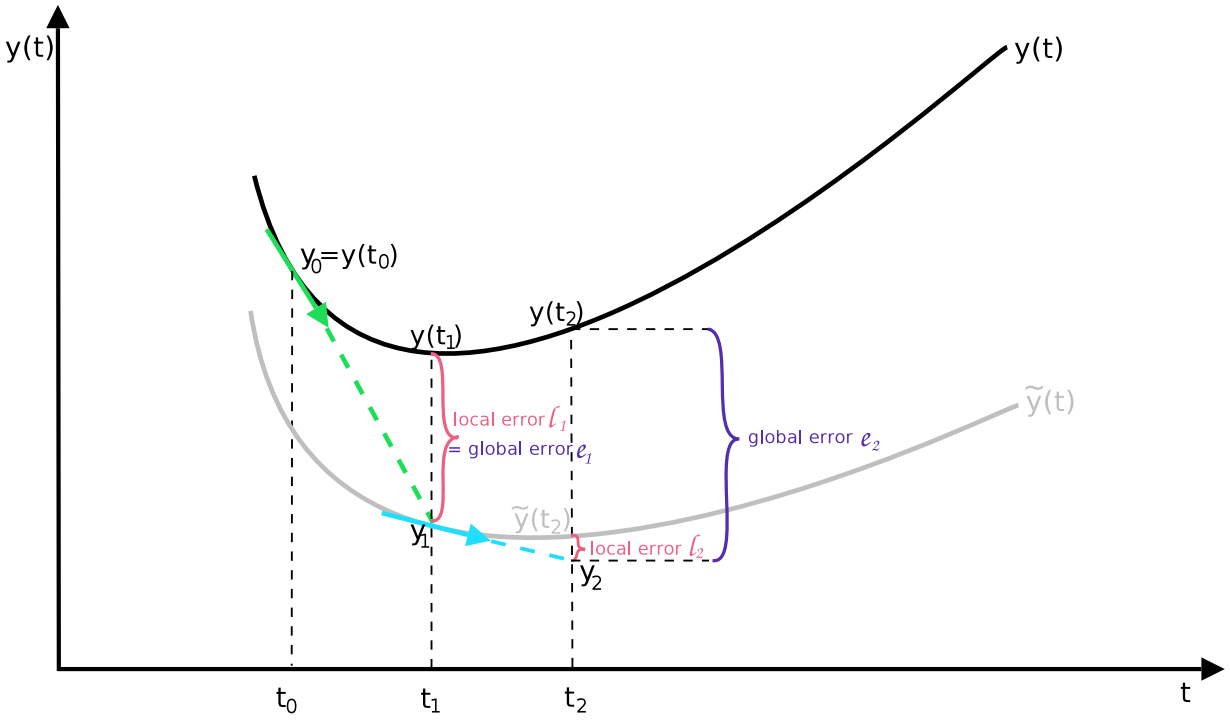

Figure 4.1: The first two steps of the explicit Euler method

leads to a better approximation if a "useful" discretisation scheme is applied. To determine whether this quality is fulfilled for the explicit Euler method, the convergence of the Euler approximation to the exact solution for $h \to 0$ has to be analysed.
There are two characteristics of a discretisation scheme which are inevitable for its convergence:

1. A discretisation method fulfills the quality of being *consistent* if the local error l_k becomes close to 0 for $h \to 0$, i.e.

$$\|l_k\| \to 0 \quad \text{as} \quad h \to 0 \tag{4.11}$$

   The method is called *consistent of order* p if $\|l_k\| = O(h^p)$.

2. The *stability* of a discretisation method signifies that the global effect of local errors remains bounded uniformly for $h \to 0$. A one-step method has to fulfill the "step-by-step" stability in order to be a *stable* discretisation scheme:
   For two "parallel" steps

$$(t_{k-1}, y_{k-1}) \mapsto (t_k, y_k) \tag{4.12}$$

$$(t_{k-1}, \widetilde{y}_{k-1}) \mapsto (t_k, \widetilde{y}_k) \tag{4.13}$$

of a one-step method with step size h applied to an IVP, the method is called *stable* if

$$\|y_k - \widetilde{y}_k\| \leq (1 + Sh)\|y_{k-1} - \widetilde{y}_{k-1}\| \tag{4.14}$$

holds uniformly for $h \leq h_0$ with some h-independent constant S and $h_0 > 0$.

Now, the explicit Euler method is analysed with regard to its properties of consistency and stability. The order of consistency for the explicit Euler method is derived as follows: For a twice continuously differentiable function $y(t)$, the local discretisation error l_k as defined in (4.7) can be rewritten by using the Taylor formula with remainder

$$y(t_k) = y(t_{k-1}) + y'(t_{k-1})h + \frac{1}{2}y''(\theta)h^2, \quad \theta \in (t_{k-1}, t_k) \tag{4.15}$$

and taking into account $y(t_{k-1}) = y_{k-1}$ for $y(t_k; t_{k-1}, y_{k-1})$ in (4.15):

$$\begin{aligned} l_k &= y(t_k; t_{k-1}, y_{k-1}) - y_k && (4.16)\\ &= y_{k-1} + y'(t_{k-1})h + \frac{1}{2}y''(\theta)h^2 - y_k && (4.17)\\ &= y_{k-1} + y'(t_{k-1})h + \frac{1}{2}y''(\theta)h^2 - (y_{k-1} + h\underbrace{f(t_{k-1}, y_{k-1})}_{y'(t_{k-1})}) && (4.18)\\ &= \frac{1}{2}y''(\theta)h^2 && (4.19) \end{aligned}$$

If $y''(t)$ is bounded on $[0, T]$, i.e. if there exists $M \geq 0$ with $\sup_{t\in[0,T]} \|y''(t)\| = M$,

$$\|l_k\| = \frac{1}{2}h^2\|y''(\theta)\| \leq \frac{1}{2}h^2 M \tag{4.20}$$

and therefore

$$\|l_k\| = O(h^2) = O(h^p) \quad \text{with} \quad p = 2 \tag{4.21}$$

This means that the local discretisation error for the explicit Euler method is of order $p = 2$. Because of

$$\|l_k\| \leq \frac{1}{2}h^2 M \xrightarrow{h\to 0} 0 \tag{4.22}$$

also the order of consistency of the explicit Euler method is $p = 2$.

The proof of the explicit Euler method being a stable discretisation scheme is elaborated in the following lines by considering two parallel Euler steps $y_k = y_{k-1} + hf(t_{k-1}, y_{k-1})$ and $\widetilde{y}_k = \widetilde{y}_{k-1} + hf(t_{k-1}, \widetilde{y}_{k-1})$:

$$\begin{aligned} \|y_k - \widetilde{y}_k\| &= \|y_{k-1} - \widetilde{y}_{k-1} + h\left(f(t_{k-1}, y_{k-1}) - f(t_{k-1}, \widetilde{y}_{k-1})\right)\| && (4.23)\\ &\leq \|y_{k-1} - \widetilde{y}_{k-1}\| + h\|f(t_{k-1}, y_{k-1}) - f(t_{k-1}, \widetilde{y}_{k-1})\| && (4.24)\\ &\leq \|y_{k-1} - \widetilde{y}_{k-1}\| + hL\|y_{k-1} - \widetilde{y}_{k-1}\| && (4.25)\\ &= (1 + Lh)\|y_{k-1} - \widetilde{y}_{k-1}\| && (4.26) \end{aligned}$$

for a Lipschitz constant $L \geq 0$ with respect to y for the right-hand side f of the differential equation supplying $\|f(t, y) - f(t, \widetilde{y})\| \leq L\|y - \widetilde{y}\|$.

After having analysed the qualities of consistency and stability now the theorem concerning the convergence of the explicit Euler scheme can be formulated:

Theorem 4.6 (**Convergence of the explicit Euler method**). *Assume $y(t)$ being twice continuously differentiable and $y''(t)$ bounded on $[0, T]$, i.e. $\sup_{t\in[0,T]} \|y''(t)\| = M$. Then the global error $e_k = y(t_k; t_0, y_0) - y_k$ (cf. (4.8)) of the explicit Euler method with step size h satisfies*

$$\|e_k\| \leq \frac{1}{2}(e^{Lt_k} - 1)\frac{M}{L}h \tag{4.27}$$

provided that $e_0 = 0$.

$$\|e_k\| = O(h) = O(h^1) \tag{4.28}$$

implies that the explicit Euler method is convergent of order $p = 1$.

Proof. To prove Theorem 4.6, an Euler step

$$y_{k-1} \rightarrow y_k := y_{k-1} + hf(t_{k-1}, y_{k-1}) \tag{4.29}$$

and a "parallel" step starting from the exact solution $y(t_{k-1};\, t_0, y_0)$

$$y(t_{k-1};\, t_0, y_0) \rightarrow \widetilde{y}_k := y(t_{k-1};\, t_0, y_0) + hf(t_{k-1}, y(t_{k-1};\, t_0, y_0)) \tag{4.30}$$

are considered. For these two parallel steps (4.29) and (4.30)

$$\|y_k - \widetilde{y}_k\| \leq (1 + Lh)\|y_{k-1} - y(t_{k-1};\, t_0, y_0)\| \tag{4.31}$$

holds because of the explicit Euler method's property of stability (4.26).
With the aim of determining the size of the global error e_k,

$$||e_k|| = \|y_k - y(t_k;\, t_0, y_0)\| \leq \|y_k - \widetilde{y}_k\| + \|\widetilde{y}_k - y(t_k;\, t_0, y_0)\| \tag{4.32}$$

is used. By taking into account that $\widetilde{y}_k - y(t_k;\, t_0, y_0) = l_k$ and by inserting (4.31) into (4.32) this leads to

$$\begin{aligned} ||e_k|| &\leq (1 + Lh)\| \underbrace{y_{k-1} - y(t_{k-1};\, t_0, y_0)}_{e_{k-1}} \| + ||l_k|| && (4.33) \\ &\leq (1 + Lh)||e_{k-1}|| + \frac{1}{2}h^2 M && (4.34) \end{aligned}$$

because of the explicit Euler method's property of consistency (4.26).
By using the *discrete Gronwall lemma* (cf. [Auzinger, 2007], section 2.1), this recursively formulated upper estimate of e_k can be rewritten as an estimate which only depends on e_0:

$$||e_k|| \leq e^{Lt_k}||e_0|| + \frac{e^{Lt_k} - 1}{Lh}\frac{Mh^2}{2} \tag{4.35}$$

Having supposed $e_0 = 0$, this leads to

$$||e_k|| \leq \frac{1}{2}(e^{Lt_k} - 1)\frac{M}{L}h \tag{4.36}$$

□

The complete proof including the calculation of the discrete Gronwall lemma can be found in [Auzinger, 2007].

Remark 4.7. *Assume that $e_0 \neq 0$ and furthermore assume that e_0 remains constant or bounded for $h \searrow 0$, i.e. $e_0 = O(h)$. Then, according to Theorem 4.6, an upper estimate for the global error can be found through*

$$||e_k|| \leq e^{Lt_k}||e_0|| + \frac{1}{2}(e^{Lt_k} - 1)\frac{M}{L}h \tag{4.37}$$

and the explicit Euler method is convergent of order $p = 1$.

Remark 4.8. *The convergence theorem refers to the case of a constant step size h and therefore of an equidistant grid. If h is variably chosen for each step, a conclusion concerning the convergence as in Theorem 4.6 cannot be drawn. For further considerations see section 4.4.*

4.3 Improved explicit one-step methods

Achieving higher accuracy with the explicit Euler method by decreasing the step size increases the numerical effort. This is the reason why it is useful to deal with alternatives to the explicit Euler method for solving IVPs. Furthermore, it shall be mentioned that for an extremely small step size another problem occurs: There is a minimum step size h_{min} for each IVP below which the accuracy of the approximations cannot be improved because the roundoff error is dominating.

The objective is to construct discretisation methods which are more efficient than the explicit Euler scheme which means that better approximations can be achieved with less computational effort. Such a desired method has a higher order of convergence than $p = 1$ then.

4.3.1 How constructing an improved explicit one-step method

It is easy to imagine that better approximations can be obtained if more information is taken into account: Instead of only using the field line of the initial point of the regarded sub-interval $[t_{k-1}, t_k]$, the slope at several points in this interval can be included into the calculation.
Of course, these points are unknown. But it is sufficient to approximate one or more solution values in the interval $[t_{k-1}, t_k]$ which can then be used to construct a better approximation of the exact solution at t_k.

In contrast to the explicit Euler method which has been described in (4.5)-(4.6) through

$$\begin{aligned} y_0 &= y(0) \\ y_k &= y_{k-1} + hf(t_{k-1}, y_{k-1}), \quad k = 1, \ldots, m \end{aligned}$$

a general explicit one-step method for the solution of an IVP is a discretisation scheme of the form

$$y_0 = y(0) \tag{4.38}$$

$$y_k = y_{k-1} + h\varphi(t_{k-1}, y_{k-1}; h), \quad k = 1, \ldots, m \tag{4.39}$$

The function φ depending on the step size h and on the initial point (t_{k-1}, y_{k-1}) for the actual step is called *increment function*.

A simple example of an explicit one-step method which uses more information than the explicit Euler method is the so-called *improved Euler method*:

$$Y_2 = y_{k-1} + \frac{h}{2} f(t_{k-1}, y_{k-1}) \tag{4.40}$$

$$y_k = y_{k-1} + h f(t_{k-1} + \frac{h}{2}, Y_2) \tag{4.41}$$

So an intermediate value Y_2 is calculated by an explicit Euler step with step size $\frac{h}{2}$ in order to use the field line at $(t_{k-1} + \frac{h}{2}, Y_2)$ to predict the value y_k. Inserting (4.40) into (4.41) shows that

$$\varphi(t_{k-1}, y_{k-1};\, h) = f\left(t_{k-1} + \frac{h}{2}, y_{k-1} + \frac{h}{2} f(t_{k-1}, y_{k-1})\right) \tag{4.42}$$

The improved Euler method is the simplest case of the *explicit Runge-Kutta methods* which are described in the following subsection.
As well as for the explicit Euler scheme, consistency and stability are necessary for the convergence of the approximations calculated by means of an improved explicit one-step method, too.

Remark 4.9. *For higher order one-step methods, the local error typically is an expression involving higher order derivatives of $y(t)$ with respect to t. So to guarantee consistence or convergence of a certain order the corresponding smoothness of $y(t)$ is required.*

It can be shown that the improved Euler method is consistent with order $p = 3$ and convergent with order $p = 2$ (see [Auzinger, 2007]).

4.3.2 Runge-Kutta methods

In the previous subsection, the improved Euler method has already been designated as a very simple explicit Runge-Kutta method.
As a generalisation, an *explicit s-stage Runge-Kutta method* is defined as follows:

Definition 4.10 (**Explicit s-stage Runge-Kutta method**). *An explicit one-step method which calculates y_k by starting from y_{k-1} and using s recursively generated intermediate values ("stages") Y_i at the points $\tau_i := t_{k-1} + \gamma_i h,\ i = 1, \ldots, s,\ \gamma_1 = 0$*

$$\begin{aligned} Y_1 &= y_{k-1} \\ Y_2 &= y_{k-1} + h\alpha_{21} f(\tau_1, Y_1) \end{aligned}$$

$$
\begin{aligned}
Y_3 &= y_{k-1} + h\left(\alpha_{31} f(\tau_1, Y_1) + \alpha_{32} f(\tau_2, Y_2)\right) \\
&\vdots \\
Y_s &= y_{k-1} + h\left(\alpha_{s1} f(\tau_1, Y_1) + \alpha_{s2} f(\tau_2, Y_2) + \ldots + \alpha_{s,s-1} f(\tau_{s-1}, Y_{s-1})\right)
\end{aligned} \tag{4.43}
$$

$$
y_k = y_{k-1} + h \underbrace{\left(\beta_1 f(\tau_1, Y_1) + \beta_2 f(\tau_2, Y_2) + \ldots + \beta_{s-1} f(\tau_{s-1}, Y_{s-1}) + \beta_s f(\tau_s, Y_s)\right)}_{\varphi(t_{k-1}, y_{k-1}; h)}
$$

is called explicit s-stage Runge-Kutta method.

So there are s intermediate points $\tau_i \in [t_{k-1}, t_k]$ for which the corresponding approximations Y_i are calculated by using the information of the already available approximations Y_j, $j = 1, \ldots, i-1$, weighted by α_{ij}.
Finally, weights β_i allow combining the field lines for each calculated point (τ_i, Y_i), $i = 1, \ldots, s$ in order to obtain the corresponding increment function φ.

An explicit s-stage Runge-Kutta method is usually symbolised by the coefficient tableau

$$
\begin{array}{c|ccccc}
0 & & & & & \\
\gamma_2 & \alpha_{21} & & & & \\
\gamma_3 & \alpha_{31} & \alpha_{32} & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
\gamma_s & \alpha_{s1} & \alpha_{s2} & \ldots & \alpha_{s,s-1} & \\
\hline
 & \beta_1 & \beta_2 & \ldots & \beta_{s-1} & \beta_s
\end{array}
$$

For example, the corresponding tableau of the improved Euler method is

$$
\begin{array}{c|cc}
0 & & \\
\frac{1}{2} & \frac{1}{2} & \\
\hline
 & 0 & 1
\end{array}
$$

It can be proved that every Runge-Kutta method is stable so that an order $p+1$ of consistency implies an order p of convergence. However, to achieve a certain order of convergence, the coefficients γ_i, α_{ij} and β_i must satisfy the corresponding order conditions which can be derived by Taylor expansion.
The coefficients γ_i, α_{ij} and β_i determined for some examples can be found in [Stetter, 1985], [Auzinger, 2007], and [Kloeden and Platen, 1992]. For a more detailed background the

reader is referred to the literature dealing more thoroughly with this topic.

So the objective is to find γ_i, α_{ij} and β_i, $i = 1, \ldots, s$, $j = 1, \ldots, i$ which allow achieving a very high order p of convergence by using only a few stages s.
The following table explains which values of s lead to which optimal order p:

$s \leq 4$	$p = s$
$5 \leq s \leq 7$	$p = s - 1$
$8 \leq s \leq 9$	$p = s - 2$

The fourth order Runge Kutta methods are the most commonly used, representing a good compromise between accuracy and computational effort.

4.4 Adaptive step size

At the beginning of section 4.3, it has already been explained why it is recommendable to use improved solving methods having higher orders of convergence: Higher accuracy is achieved without making the step size that small that there were heavy losses of efficiency as well as comparatively elevated influence of roundoff errors.
But also with higher order schemes it can be necessary to choose a small step size, at least for parts where the slope changes very fast.

In figure 4.2, the local error of the j-th step is very small whereas the one of the k-th step is not. Decreasing the step size over the whole regarded interval would lead to a better approximation in the right-hand part of the shown function. But at the same time, this would cause needless computational effort in the region of the left-hand side.
By contrast, a variable step size would allow adjusting the actual step size to the function's properties in each step which would lead to more accurate approximations without loss of efficiency.

Adaptive step size works as follows:

- For each new calculated approximation value y_k, the local error, i.e. the quantity (4.7) has to be determined. Because the real solution $y(t_k;\, t_{k-1}, y_{k-1})$ is usually unknown the local error is estimated. Even so, the estimated local error is also called *observed local error*.

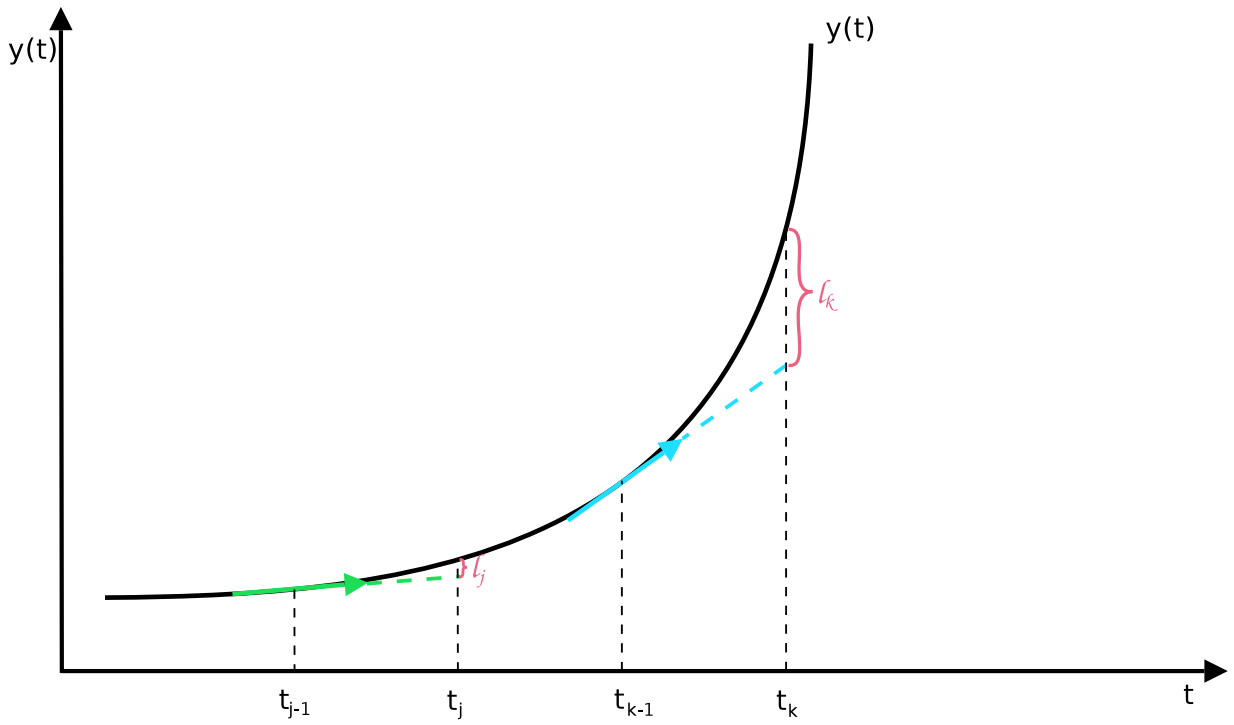

Figure 4.2: Why it is useful to have variable step size

- Then, the estimated local error is compared to a chosen error level. If the observed error is small enough the step is accepted. If not, y_k is rejected and the step has to be repeated for some smaller step size.
- Whether the step is accepted or not, a new step size has to be determined. The new step size is used to repeat the step which has been rejected or to execute the following step if y_k has been accepted. The aim is to calculate a new step size for which the accuracy requirements are fulfilled as narrowly as possible (so that not too much effort is caused).

To estimate the local error there are two established methods:

1. The first possibility is to use two discretisation schemes having different orders of convergence. The difference between the results then is an estimate of the local error.
   Some Runge-Kutta methods make a very precious contribution here: The so-called *embedded Runge-Kutta formulas* are pairs of Runge-Kutta schemes having the following properties:

- They have different numbers of stages s, $\overline{s}$ and are of different orders p, $\overline{p}$ (usually, $\overline{p} = p + 1$).
- The coefficient tableau of the more accurate scheme contains all coefficients γ_i and α_{ij}, $i = 1, \ldots, s$, $j = 1, \ldots, i$ of the lower order scheme. As it has $\overline{s}$ intermediate values ($\overline{s} > s$), it is completed by the additional coefficients having as indices $i = s + 1, \ldots, \overline{s}$ and $j = 1, \ldots, i$.
- The weights β_i, $\overline{\beta}_i$ are different for two such schemes.

The following coefficient tableaus correspond to a Runge-Kutta method with $s = 4$, $p = 4$ and a second one with $\overline{s} = 6$, $\overline{p} = 5$ which together form an embedded Runge-Kutta formula.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$		
1	0	-1	2	
	$\frac{1}{6}$	0	$\frac{2}{3}$	$\frac{1}{6}$

0						
$\frac{1}{2}$	$\frac{1}{2}$					
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$				
1	0	-1	2			
$\frac{2}{3}$	$\frac{7}{27}$	$\frac{10}{27}$	0	$\frac{1}{27}$		
$\frac{1}{5}$	$\frac{28}{625}$	$-\frac{1}{5}$	$\frac{645}{625}$	$\frac{54}{625}$	$-\frac{378}{625}$	
	$\frac{14}{336}$	0	0	$\frac{35}{336}$	$\frac{162}{336}$	$\frac{125}{336}$

So almost without additional effort, every step can be executed with both schemes in order to obtain two approximations of different accuracy for every step. The difference then is a useful estimate of the local error.

2. The other way is the so-called procedure of *step-doubling*: Each step is taken twice, once as a full step with step size $h = t_k - t_{k-1}$

$$(t_{k-1}, y_{k-1}) \xrightarrow{h} (t_k, y_k) \tag{4.44}$$

then, independently, as two half steps with step size $\frac{h}{2} = \frac{t_k - t_{k-1}}{2}$

$$(t_{k-1}, y_{k-1}) \xrightarrow{\frac{h}{2}} (t_{k-1} + \frac{h}{2}, \overline{y}_{k-\frac{1}{2}}) \xrightarrow{\frac{h}{2}} (t_k, \overline{y}_k) \tag{4.45}$$

The difference between y_k and $\overline{y}_k$, again, serves as an estimate of the local error.

The desired error level $\epsilon > 0$ can be chosen to be an absolute or a relative error bound or a mixed version:

$$\epsilon = \epsilon_{abs} + \epsilon_{rel} \left(a_y ||y_k|| + a_{y'} ||f(t_k, y_k)|| \right) \tag{4.46}$$

with ϵ_{abs} and ϵ_{rel} being the absolute and relative component of the bound, respectively, and a_y and $a_{y'}$ representing the scaling factors regulating the influence of y_k and $f(t_k, y_k)$. How can now, starting from the actual step size h, the observed local error l_k and a desired error level ϵ, the step size for the next step be determined?
For a discretisation scheme with local error of order p,

$$||l_k|| = ||y(t_k;\, t_{k-1}, y_{k-1}) - y_k|| = O(h^p) \approx h^p c \tag{4.47}$$

with $c > 0$ depending on the considered IVP and on the position t_{k-1}. It is assumed that c changes slowly with reference to t. Then, the following way of proceeding seems natural: The estimated local error of the actual step (cf. (4.44) and (4.45))

$$||y_k - \overline{y}_k|| =: s \geq 0 \tag{4.48}$$

can be used to calculate c:

$$c \approx \frac{s}{h^p} \tag{4.49}$$

With the aim of achieving the tolerance ϵ with the new step size h_{new}

$$h_{new}^p c \approx \epsilon \tag{4.50}$$

h_{new} is determined as follows:

$$h_{new} = \sqrt[p]{\frac{\epsilon}{c}} = \sqrt[p]{\frac{\epsilon}{s} h^p} = \sqrt[p]{\frac{\epsilon}{s}} h \tag{4.51}$$

The formula (4.51) is relevant for both repeating one step because it has been rejected and determining the next step size after an accepted step. In the first case, h_{new} is the step size the actual step is repeated with, in the second case the next step has length h_{new}. Usually, some safety factor $S < 1$ (e.g. $S = 0.8$) is included in the formula

$$h'_{new} = S \sqrt[p]{\frac{\epsilon}{s}} h \tag{4.52}$$

to avoid coming too close to the tolerance level ϵ.

As it has already been mentioned in Remark 4.8, when using adaptive step size it is not possible to formulate a convergence declaration like it is done in Theorem 4.6 for an equidistant grid. Regarding things from a different point of view, the theoretical statements and conclusions in section 4.2 and 4.3 have been derived by supposing simplified conditions which are not true in practical implemention. So the convergence conclusions have been drawn with respect to a constant step size over the whole integration interval whereas in practical applications adaptive step size leads to a non-equidistant grid which locally is adapted to the properties of the IVP.
Nevertheless, having a vague idea of the global error's size in the considered calculation is of great interest also in this case. The user keeping in mind a certain global error level has the following opportunity:
Taking into account the length T of the integration interval as well as the behaviour of the field line, he can choose the local error tolerance ϵ that way that most likely his desired global error level is achieved. Usually, the local error level will then be chosen to have some decimal powers less than the global one (e.g. for a desired global error of about 10^{-5}, the local error will be $\epsilon = 10^{-7}$).
The user also has the possibility to call the routine twice with two different tolerance levels (e.g. ϵ_1 and $\epsilon_2 = \frac{\epsilon_1}{10}$) and by calculating the difference of the two results to gain an estimate of the global error size.
In any case, there is no possibility to have an a priori estimate of the global error when using adaptive step size. But analysing the differential equation and its field line as well as realising the just explained proceedings at least allows the user a posteriori forming an opinion concerning the size of the occurred global error.

4.5 Further methods for solving IVPs

In the preceding parts of this chapter, only explicit one-step methods have been discussed. But these methods are just one type of numerical solving schemes for IVPs. According to a differential equation's properties it makes sense to use different types of discretisation methods for different IVPs.
In contrast to the explicit methods which all of the already introduced solving schemes belong to, there is the important class of *implicit solving methods*. Furthermore, as opposed to the one-step schemes, also *multi-step methods (explicit and implicit)* can be used to solve IVPs. In addition to those two big classes of discretisation schemes, there also exist further

methods which, for example, represent a combination of multi-step methods.
The first two of the now mentioned classes of solving methods are briefly explained in the following two subsections. But as it turns out that the choice of an explicit Runge-Kutta method for solving Thiele's differential equation is justified under certain conditions (see Section 5.2), I will not go into detail with those additional methods.

4.5.1 Implicit solving methods

As described in section 4.3, the information used for advancing an approximated solution from (t_{k-1}, y_{k-1}) to (t_k, y_k) need not necessarily come from the initial point of the considered interval. Also intermediate function values of $(t_{k-1}, t_k]$ can be included into the calculation which, of course, are unknown at this moment.
Instead of approximating an intermediate value as shown in section 4.3, it is also possible to solve an equation which contains unknown values.
The *implicit Euler scheme*

$$y_0 = y(0) \tag{4.53}$$

$$y_k = y_{k-1} + hf(t_k, y_k), \quad k \in \{1, \ldots, m\} \tag{4.54}$$

is an example of a simple implicit discretisation method.
Also a Runge-Kutta method can be implicit: To determine the intermediate values Y_i, $i = \{1, \ldots, s\}$, not only already calculated values Y_j, $j = \{1, \ldots, i-1\}$ but also unknown ones ($j = \{i, \ldots, s\}$) are used then.
So for an implicit solving method, the approximated solution is calculated by solving a non-linear system in each step.

Implicit schemes are indispensable methods for solving so-called stiff IVPs. A system of ordinary differential equations is called *stiff* if its Jacobian matrix $\frac{\partial f_i(t,y)}{\partial y_j}$ has eigenvalues λ_i with $Re\ \lambda_i \ll 0$ besides eigenvalues of moderate size. As it will turn out that this is not the case for Thiele's differential equation provided that certain conditions are fulfilled (see Section 5.2) it is not necessary to go more into detail here. The reader whose interest for implicit solving methods after all has been aroused is referred to [Auzinger, 2007] and [Stetter, 1985].

4.5.2 Multi-step methods

In section 4.2, I have already explained the difference between a one-step method and a multi-step method. When calculating y_k, a multi-step method does not only refer to y_{k-1}

but also includes y_{k-2}, y_{k-3}, ... into the calculation in order to achieve better approximated values.
Like the other solving methods which have already been mentioned also multi-step methods can either be explicit or implicit schemes. Implicit multi-step schemes belong to the class of the implicit solving methods discussed in the previous subsection and therefore are left out here. So in the next lines, only some properties of the explicit multi-step methods are briefly described.

The decided advantage of an explicit multi-step method is its efficiency: For each new step $(t_{k-1}, y_{k-1}) \rightarrow (t_k, y_k)$ only one new evaluation is required. Then the field line in (t_{k-1}, y_{k-1}) completes the already available values $f(t, y)$ for (t_{k-2}, y_{k-2}), (t_{k-3}, y_{k-3}), ... in order to calculate y_k.

By contrast, adaptive step size is very difficult or even impossible to apply on multi-step methods. Firstly, the fact that information of previous function values y_{k-2}, y_{k-3}, ... is used complicates the procedure of changing the step size: Previous function values for a completely new grid defined through the new step size are indispensable after having changed the step size. Secondly, too abruptly changing the step size in multi-step methods can cause unstable behaviour.
So if the evaluation of the right-hand side of the differential equation is not too expensive and if the properties of the differential equation do not enforce very small steps, a higher order one-step method combined with adaptive step size is a good alternative to a multi-step method.

Chapter 5

The practical application of Thiele's differential equation

The aim of this chapter is to discuss how the theoretical considerations in Chapter 3 can be put into practice. More precisely, Thiele's differential equation will be used to calculate the total prospective reserve of life insurance contracts defined through the Markovian model for a finite number $n \in \mathbf{N}$ of states and over a finite insurance period $[0, tEnd]$. Whereas in literature $[0, T]$ is a commonly used interval denoting the insurance period, I have chosen $tEnd$ instead of T to signify the end of the insurance period. As in Section 5.3.1 the introduced variable T constitutes the chosen numerical algorithm for solving Thiele's differential equation, the choice of $tEnd$ here will avoid confusion.

In contrast to Chapter 1 where the state space has been introduced as $S = \{1, \ldots, n\}$, as of now every insurance contract is assumed to start in an initial state 0. Therefore the state space

$$S = \{0, \ldots, n-1\}, \quad n \in \mathbf{N}$$

will be used for the remaining part of this diploma thesis. For reasons of simplification, I will sometimes write about state j and k, always meaning that j and k are elements of the state space S even if it is not explicitly mentioned. Using t will in any case imply that t is element of the insurance period $[0, tEnd]$, in the following also referred to as "integration interval". In Chapter 3, I have throughout used the expression "total prospective reserve" to avoid mixing it up with the prospective reserves for the different partial commitments. As in this chapter there is no risk of confusion, I will simply use the expression "reserve" when talking about the total prospective reserve.

5.1 Thiele's differential equation posed as an initial value problem

5.1.1 Thiele's differential equation

For a life insurance contract defined through the Markovian model, the course of the reserve $\{V^+(t)\}_{t\in[0,tEnd]}$ over the insurance period $[0, tEnd]$ can be calculated by using Thiele's differential equation which describes how the reserve $V^+(t)$ changes over the time t. In order to guarantee a combined treatment of continuous and discrete payment streams, discontinuities of the reserve have to be allowed and, as in (3.31) and (3.32), TDE consists of a differential equation for the continuous parts and an additional formula corresponding to the points of the finite set of discontinuities $D \subset [0, tEnd]$ of the reserve. So the component-wise formula of *Thiele's differential equation (TDE)* which holds for all $j \in S$ is

$$\begin{aligned}\frac{d}{dt}V_j^+(t) &= -a'_{jj}(t) - \sum_{\substack{k\in S\\ k\neq j}} \mu_{jk}(t)a_{jk}(t) \\ &\quad + \left(\delta'_j(t) + \sum_{\substack{k\in S\\ k\neq j}} \mu_{jk}(t)\right) V_j^+(t) - \sum_{\substack{k\in S\\ k\neq j}} \mu_{jk}(t)V_k^+(t) \quad \forall t \in [0, tEnd] \setminus D \end{aligned} \tag{5.1}$$

$$\begin{aligned}\text{with}& \\ V_j^+(t^-) &= V_j^+(t) + \Delta a_{jj}^{(d)}(t) + \Delta\delta_j^{(d)}(t) \\ \text{or}& \qquad\qquad \forall t \in D \\ V_j^+(t) &= V_j^+(t^+) + \Delta a_{jj}^{(d)}(t) + \Delta\delta_j^{(d)}(t)\end{aligned} \tag{5.2}$$

Corresponding to an n state insurance contract, (5.1) is an n dimension system of ordinary differential equations in which t is the independent variable and $V^+(t) = \left(V_0^+(t), \ldots, V_{n-1}^+(t)\right)$ is the dependent variable and unknown solution – a vector of the reserves at time t in state $0, \ldots, n-1$.
In the remaining part of my diploma thesis, I will frequently refer to TDE by using the general expressions $f_j(t, V^+)$ for $\frac{d}{dt}V_j^+(t)$ and $f(t, V^+)$ for $\frac{d}{dt}V^+(t)$, respectively.

In Chapter 3, TDE for the continuous case (cf. equation (3.28)) has been deduced and also a formula for the reserve allowing discontinuities (cf. equations (3.31) and (3.32)) has

been presented. (5.1) and (5.2) slightly differ from (3.31) and (3.32). By contrast to the formulas of Chapter 3 where $V_j^+(t)$ is a càdlàg function, here it can have both càdlàg parts and parts where it is continuous on the left with limit on the right. The other modification concerning the functions δ'_j and $\Delta\delta_j^{(d)}$ ensures more flexibility with respect to the interest. So by using the modified formulas (5.1) and (5.2) very individually created contracts can be considered.

Now the different symbols in (5.1) and (5.2) will be briefly described just to get an idea of which functions cause changes of the reserve. In Section 5.1.3 the components of TDE and their properties will be explained in detail. For better understanding, also the functions a_{jj} and δ_j now are described in a few words although they do not appear in (5.1) and (5.2).

- $a_{jj}(t)$:
  the accumulated payment stream for state j describing the "sum" of the payments from time 0 until time t, provided that the contract has always been in state j

- $a'_{jj}(t)$:
  the value by which the continuous component (cf. Section 5.1.3) of $a_{jj}(t)$ is increased at time t, i.e. the continuously effected payment at time t paid for remaining in state j

- $\Delta a_{jj}^{(d)}(t)$:
  the discrete payment at time t for remaining in state j

- $a_{jk}(t)$:
  the payment paid for the transition from state j to state k at time t

- $\mu_{jk}(t)$:
  the transition intensity for the transition from state j to state k at time t

- $\delta_j(t)$:
  the accumulated interest from time 0 until time t provided that the contract has always been in state j

- $\delta'_j(t)$:
  the value by which the continuous component of $\delta_j(t)$ is increased at time t, i.e. the force of interest at time t for remaining in state j

- $\Delta\delta_j^{(d)}(t)$:
  the interest added at the discrete point in time t for remaining in state j

The continuous change (cf. (5.1)) of the reserve $V_j^+(t)$ at time t consists of four parts:

1. the continuously effected payment at time t for remaining in state j

2. the sum of the expected payments at time t for transitions from state j to state $k,\ k \neq j$. Intuitively spoken, the addends of this term represent the values of the payments the insurer has to provide if the corresponding transitions occur. In actual fact, the reserve is an expected value and it increases or decreases by the expected payments whether the corresponding transition occurs or not.

3. The third term indicates that on the one hand $V_j^+(t)$ increases by the continuously realised interest of the already existing value. On the other hand, it increases by the expected profit due to its own value becoming available because of transitions $j \to k,\ k \neq j$.

4. The fourth term describes the amounts which are subtracted because they are, based on the expected transitions, transferred from $V_j^+(t)$ to $V_k^+(t)$, $k \neq j$.

An intuitive idea of 3 and 4, again, is that $V_j^+(t)$ becomes available while $V_k^+(t)$ has to be provided for the corresponding transition. In reality, independently from the transitions, the reserve changes by the expected amounts.

At the points of discontinuity, i.e. at $t \in D$, discrete payments and interest cause changes of the reserve (cf. (5.2)).

Notice that the payment functions a'_{jj}, $\Delta a_{jj}^{(d)}$ and a_{jk} can have positive or negative values. If these payments are benefits or costs and therefore positive the reserve (regarded in the common positive direction of time) decreases by effecting the payments. Negative payments (contributions) let the reserve increase.

5.1.2 The initial value problem

Starting from TDE, an initial condition is required to form an initial value problem (IVP). By means of the *equivalence principle* an appropriate initial condition can be found:
For every insurance contract having as insurance period the interval $[0, tEnd]$, the specified insurance benefits determine the given insurance coverage. The price for buying this insurance coverage is paid in terms of contributions which obviously depend on the scale of the defined benefits. The equivalence principle as premium principle is based on the

law of large numbers and works as follows: When considering a large number $N \in \mathbf{N}$ of identical insurance contracts which are all issued at time 0, according to the corresponding transition probabilities the sum of all expected discounted benefits is equal to the sum of all expected discounted contributions at time 0 for $N \to \infty$. So for every single insurance contract it is justified to regard the expected discounted benefits as equal to the expected discounted contributions at time 0, provided that a sufficiently large number of insurance contracts is issued and that safety margins are taken into account.

Coming from classical life insurance mathematics, the equivalence principle usually refers to the net reserve because only benefits and contributions but not costs are included in the calculation. But as for the Markovian model positive values of $a'_{jj}(t)$, $\Delta a^{(d)}_{jj}(t)$ and $a_{jk}(t)$ can also signify arising costs that the insurer has to pay, for this life insurance model the equivalence principle allows determining the gross contributions.

Because the reserve introduced in Chapter 3 is defined as the difference between the expected discounted benefits and costs and the expected discounted contributions with condition to the actual state, according to the equivalence principle the reserve of the initial state 0 at time 0 is equal to 0, i.e. $V_0^+(0) = 0$. This important result is used when calculating the contribution heights (see Section 5.1.5) but does not lead to an appropriate initial condition here. By contrast, the value of the reserve for all states at time $tEnd$ can be determined analogously and provides the desired result: When the insurance period is over, i.e. at time $tEnd$, there are no more future benefits or costs and no more future contributions. Following the equivalence principle, $V_j^+(tEnd) = 0$ for all states $j \in S$ which constitutes the required initial condition.

So for the special case where the reserve does not have any discontinuities, i.e. $D = \emptyset$, the course of the reserve can be described through the following IVP consisting of TDE and the corresponding initial condition (component-wise formula, $\forall j \in S$):

$$\begin{aligned}
\frac{d}{dt}V_j^+(t) &= -a'_{jj}(t) - \sum_{\substack{k\in S\\ k\neq j}} \mu_{jk}(t)a_{jk}(t) \\
&\quad + \left(\delta_j(t) + \sum_{\substack{k\in S\\ k\neq j}} \mu_{jk}(t)\right) V_j^+(t) - \sum_{\substack{k\in S\\ k\neq j}} \mu_{jk}(t)V_k^+(t), \quad t \in [0, tEnd] \\
V_j^+(tEnd) &= 0 \qquad (5.3)
\end{aligned}$$

So having an initial condition for $tEnd$ means that instead of calculating the reserve in the common positive direction it is computed following the opposite direction.

If $D \neq \emptyset$ the course of the reserve is given through a sequence of contiguous IVPs

which have to be solved successively starting at $tEnd$ and following the negative direction of t: The whole integration interval $[0, tEnd]$ is split up into a sequence of m sub-intervals $[t_{i-1}, t_i],\ i = 1, \ldots, m,\ m \in \mathbf{N}$ with $t_0 = 0$, $t_m = tEnd$ and t_i in between signifying the remaining points of discontinuity. So starting at time $tEnd$, the reserve is calculated from one point of discontinuity until the next one in order to arrive at time 0. If required, $V_j^+(t^-)$ and $V_j^+(t^+)$, respectively, are calculated at the beginning and at the end of the sub-intervals.

Two different types of discontinuities of the reserve can be distinguished:

- additional additive values which occur at discrete times t, i.e. $\Delta a_{jj}^{(d)}(t)$ and $\Delta \delta_j^{(d)}(t)$
- discontinuities of the right-hand side of the differential equation caused by discontinuities of at least one of the components a'_{jj}, a_{jk}, μ_{jk} and δ'_j at those points

For the fist type of discontinuity, the proceeding of calculating the reserve in the way it has been described above is evidently clear: The differential equation is well-defined also at those points for which additional discrete values are defined. So the integration over $[0, tEnd]$ is simply interrupted at those points and the discrete values are added before continuing with the integration.
By contrast, it has not been described in detail which properties of the functions a'_{jj}, a_{jk}, μ_{jk} and δ'_j let occur discontinuities of the second type. These details as well as an exact description of the proceeding for calculating the reserve for this type of discontinuity will be given in the following subsection.

5.1.3 Describing the continuous and discrete components of TDE

As it can be seen from (5.1) and (5.2), the components of TDE consist of both continuous and discrete functions in order to allow considering a broad range of individually created contracts. Of course, it is presumed that these functions exist and have finite values over the whole insurance period.

1. Payments for remaining in state $j \in S$:
   - $a_{jj}(t) = \int_0^t da_{jj}(s)$ is a function representing the accumulated payment stream from time 0 until time t, $t \in [0, tEnd]$ provided that the contract has always been in state j. It contains both continuously effected payments and payments

realised at discrete points in time. $a_{jj}(t)$ can be split up into a continuous and a discrete component

$$a_{jj}(t) = a_{jj}^{(c)}(t) + a_{jj}^{(d)}(t)$$

with

$$a_{jj}^{(d)}(t) = \sum_{0 \le s \le t} \Delta a_{jj}^{(d)}(s) = \sum_{0 \le s \le t} \left(a_{jj}^{(d)}(s^+) - a_{jj}^{(d)}(s^-) \right)$$

and

$$a_{jj}^{(c)}(t) = a_{jj}(t) - a_{jj}^{(d)}(t)$$

- $\Delta a_{jj}^{(d)}(t)$ is the discrete payment at time t for remaining in state j. As it has been described in the previous subsection, the integration has to be interrupted at this point and $\Delta a_{jj}^{(d)}(t)$ has to be added. Then the integration can be continued.
- $a'_{jj}(t) = \frac{da_{jj}^{(c)}(t)}{dt}$ describes the change of the continuous component $a_{jj}^{(c)}(t)$ of the accumulated payment stream $a_{jj}(t)$, i.e. the continuously effected payment at time t paid for remaining in state j.

  In contrast to the type of discontinuity where an additional discrete value has to be taken into account, discontinuities of $a'_{jj}(t)$ cause discontinuities of the right-hand side of the differential equation. For $a'_{jj}(t)$ two cases of discontinuity can be distinguished:

  - If $a_{jj}^{(c)}$ is differentiable but not continuously differentiable at time t, a'_{jj} is either càdlàg or continuous on the left with limit on the right at t.
  - If $a_{jj}^{(c)}$ is not differentiable at time t, a'_{jj} does not exist at t. Figure 5.1 shows this case of discontinuity of a'_{jj}: The function a_{jj} illustrates that contributions are paid continuously over the interval $[0, 4)$ in state j. In the same state, annuity benefits are paid continuously over the interval $[4, 10)$. So $a_{jj}^{(c)}(t)$ is not continuously differentiable at $t = 4$ and $t = 10$.

  In both of those two cases there is no additional discrete value which has to be added. Nevertheless, the integration interval has to be split up at those points of discontinuity. All that has to be done then is to continuously extend a'_{jj} for each sub-interval at those points where it is not defined. Interrupting the integration at the points of discontinuity and continuing with the next sub-interval leads to the desired result.

2. The function a_{jk}:
   Whereas a'_{jj} and $\Delta a_{jj}^{(d)}$ help defining continuous and discrete payments for remaining

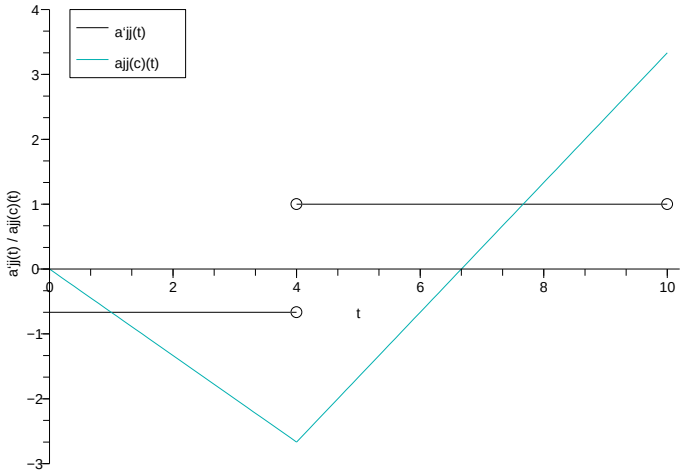

Figure 5.1: $a'_{jj}(t)$ and $a^{(c)}_{jj}(t)$

in state j, different payment types and times for the transition payments are all described by the function a_{jk}.

- There can be variable finite heights of the payments, depending on the time of the transition and, of course, on the states j and k.
- For each payment it can be chosen whether to effect the payment immediately at the point of time the transition occurs at or to realise it at an individually specified later date.

Depending on whether the payment is immediately effected or at a later time, a_{jk} has different meanings:

- If the payment is defined to be effected immediately at time t, $a_{jk}(t)$ is equal to the value fixed for the transition from state j to state k at time t.
- If the payment is defined to be realised at a later time than time t, nevertheless the reserve is adjusted immediately at time t. Because the full value fixed for the transition is not paid immediately but at a later time, not the full value has effect on the reserve but only the discounted one. So in this case, $a_{jk}(t)$ is the discounted value of the full value specified for the transition. That is why there

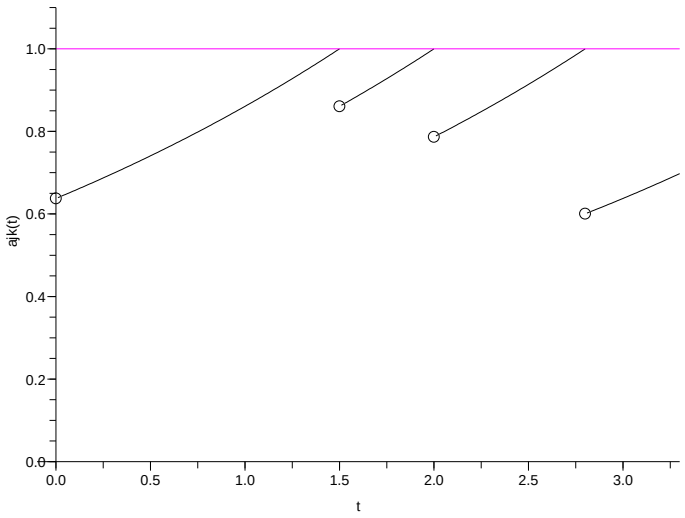

Figure 5.2: a_{jk} effected immediately vs. a_{jk} effected at later paying times

occur discontinuities of a_{jk} in t even if the value fixed for the transition remains constant over the insurance period.

Figure 5.2 shows a payment function a_{jk} for payments of amount 1 effected at the same time t the transition occurs at, as well as a payment function a_{jk} for payments of the same height effected at individually chosen paying times. The two functions are symbolized by the red and the black curve, respectively. The black curve indicates that if a transition occurs in the interval $(0, 1.5]$, the amount of 1 is paid at the end of this interval. The remaining part of the black curve can analogously be interpreted. Notice that Figure 5.2 just shows an example! a_{jk} need not necessarily be a function which is continuous on the left with limit on the right over $[0, tEnd]$ – it can also contain càdlàg parts.

For the points of discontinuity of a_{jk} the proceeding is exactly the same as it has been described for a'_{jj}: The integration interval is split up at those points, a_{jk} is continuously extended and the sequence of IVPs is solved successively.

3. μ_{jk}:
   Also the functions μ_{jk} modelling the transition intensities from state j to state k

can be individually defined. One can easily imagine μ_{jk} having discontinuities in t, when thinking of transitions that are only possible up to a certain time t^* but not afterwards. As μ_{jk} also is a function describing values for transitions from one state to another one, it has similar properties as a_{jk} and therefore the points of discontinuities are treated the same way as for a_{jk}.

4. The functions describing the interest for state $j \in S$:
   As the functions δ_j, δ'_j and $\Delta\delta_j^{(d)}$ are functions which describe values for remaining in state j they show the same properties as the functions a_{jj}, a'_{jj} and $\Delta a_{jj}^{(d)}$ and can be treated in the same way. The only restriction for the functions modelling the interest is to be deterministic functions in order to guarantee the validity of Thiele's differential equation.

5.1.4 Solvability of the initial value problem

Before starting to calculate a solution it has to be clarified whether the IVP (5.3) for $D = \emptyset$ respectively each IVP of the sequence of IVPs for $D \neq \emptyset$ has a solution and whether this solution is the unique one. In other words, I carefully consider which conditions the IVPs have to fulfill or in which cases I have to be attentive in order to effectively obtain an existing and unique solution.

The IVP (5.3) for $D = \emptyset$ respectively each IVP of the sequence of IVPs for $D \neq \emptyset$ represent initial value problems as described in Definition 4.1. For each of those IVPs a solution has to be calculated in the negative direction of t. This fact does not change anything for the following paragraphs in which the application of the *Peano Existence Theorem* and the *Picard-Lindelöf Theorem* is discussed. In both theorems, solutions are considered in an interval which is symmetric in t with respect to the initial value. So it does not make any difference whether to regard a solution on the right-hand side or on the left-hand side of the initial point of time.

I will write about several estimates taken from the two theorems. Instead of the 2-norm I will use the 1-norm which is easier to handle for the IVP (5.3). That does not make any difference concerning the existence of the calculated upper bounds because the different p-norms are equivalent to each other in a finite space. Of course, I will take into account the conversion constants.

As described in section 4.1, I start by considering a region

$$G = \{(t, V^+) : \ |t| < \rho_1, \ ||V^+(t) - V^+(tEnd)||_1 < \sqrt{n}\rho_2\}$$

and its closed envelope

$$\overline{G} = \{(t, V^+) : \ |t| \leq \rho_1, \ ||V^+(t) - V^+(tEnd)||_1 \leq \sqrt{n}\rho_2\}$$

For the IVP (5.3), I consider the integration interval $[0, tEnd]$ with finite value $tEnd$. Consequently, it is possible to find a value ρ_1 for which $|t| < \rho_1$ as well as $|t| \leq \rho_1$ holds $\forall t \in [0, tEnd]$.
For the values of V^+ an upper bound $\sqrt{n}\rho_2$ is prescribed so that together with the choice of the initial value $V^+(tEnd)$ only values of $V^+(t)$ lying in the region defined through $V^+(tEnd)$ and $\sqrt{n}\rho_2$, i.e. $||V^+(t) - V^+(tEnd)||_1 < \sqrt{n}\rho_2$ and $||V^+(t) - V^+(tEnd)||_1 \leq \sqrt{n}\rho_2$, respectively, are considered. For my purposes, $\sqrt{n}\rho_2$ can be chosen to be a large finite number.
The calculations can only take place in the region defined through G and $\overline{G}$, respectively. So I have to make sure that the solution $V^+(t)$ does not get outside the region because it cannot be foreseen what happens then.

The *Peano Existence Theorem* (cf. Theorem 4.2) guarantees the existence of a solution for an IVP under certain conditions which now are analysed for the IVP (5.3).

- First, $f(t, V^+)$ has to be continuous in $\overline{G}$. After having split up the interval $[0, tEnd]$ at the points of discontinuity as it has already been described and after having continuously extended $f(t, V^+)$ at the beginning and at the end of each sub-interval, the continuity condition is fulfilled.

- The second condition to be fulfilled is that f has to be bounded in such a way that

  $$||f(t, V^+)||_1 \leq \sqrt{n}A(t) \ \text{ with } \ \int_0^t \sqrt{n}A(\tau)d\tau \leq \sqrt{n}\rho_2 \quad \forall t \ \text{ with } \ |t| \leq \alpha, \ 0 < \alpha \leq \rho_1$$

  This means that $||f(t, V^+)||_1$ has an upper bound $\sqrt{n}A(t)$ which is integrable so that the solution $V^+(t)$ remains $\leq \sqrt{n}\rho_2$ within a (small) interval t, $|t| \leq \alpha, 0 < \alpha \leq \rho_1$ and there can be calculated. The procedure restarts at t in order to obtain a solution over $[0, tEnd]$.
  For my purposes, it is sufficient to consider a very generous global upper bound $\sqrt{n}A(t) = B$. I also suppose global upper bounds B_0 for the absolute values of each of the components $a'_{jj}(t)$, $a_{jk}(t)$, $\delta_j(t)$ and $\mu_{jk}(t)$ of f. Those upper bounds shall be

valid $\forall j \in \{1,\ldots,n\}$, $\forall k \in \{1,\ldots,n\}$, $k \neq j$, and over the whole integration interval $[0, tEnd]$ for all choices of those functions corresponding to their description in the previous subsection. By using those upper bounds, I want to estimate an upper bound B for f:

$$||f(t,V^+)||_1 = \sum_{j=1}^{n} |f_j(t,V^+)| \leq ||V^+||_1(2n-1)B_0 + nB_0 + n(n-1)B_0^2 \leq B$$

with V^+ such that $||V^+(t) - V^+(tEnd)||_1 \leq \sqrt{n}\rho_2$.
Then $\int_0^t \sqrt{n}A(\tau)d\tau = \int_0^t Bd\tau = tB \leq \sqrt{n}\rho_2$ for all t with $|t| \leq \alpha, 0 < \alpha \leq \rho_1$. So a solution V^+ remaining bounded, i.e. $||V^+(t) - V^+(tEnd)||_1 \leq \sqrt{n}\rho_2$, can be calculated within an interval of length $t \leq \frac{\sqrt{n}\rho_2}{B}$. Then the procedure restarts for new initial values but for the same upper bounds ρ_1, $\sqrt{n}\rho_2$ and B_0.
Only too big values of B can cause difficulties: Then $t \leq \frac{\sqrt{n}\rho_2}{B}$ becomes close to 0 which means that the calculation almost does not proceed or is even stopped. In fact, too big values of B are caused by too big initial values $V^+(tEnd)$ because we consider values $V^+(t)$ in a region around $V^+(tEnd)$ and big values of V^+ make B increase (provided that ρ_1, $\sqrt{n}\rho_2$ and B_0 are fixed). So too big values of V^+ can also be detected through small integration steps which can be observed by the user when applying a numerical solving algorithm to the IVP.

Summarising, for f being continuous and bounded in the way it is described above, the existence of a solution in $[0, tEnd]$ is ensured.

According to the *Picard-Lindelöf Theorem* (cf. Theorem 4.3), an IVP has a unique solution if the conditions which are now discussed for the case of the IVP (5.3) are fulfilled:

- $f(t, V^+(t))$ has to be continuous in G:
  This property of f can be deduced from its property of being continuous in $\overline{G}$ which is required in the Peano Existence Theorem.

- $||f(t,V^+)||_2 \leq M$ in G
  also follows from the fact that $||f(t,V^+)||_1 \leq B$ in $\overline{G}$ for appropriate upper bounds as required in the Peano Existence Theorem (which is equivalent to $||f(t,V^+)||_2 \leq \frac{1}{\sqrt{n}}B =: M$ in $\overline{G}$).

- $f(t,V^+)$ has to be Lipschitz continuous with respect to V^+, i.e. there must be a Lipschitz constant $L \geq 0$ supplying $||f(t,V_1^+) - f(t,V_2^+)||_2 \leq L||V_1^+ - V_2^+||_2$ for (t,V_1^+) and $(t,V_2^+) \in G$.

The existence of a Lipschitz constant L is definitely ensured if $f(t, V^+)$ is differentiable and if its partial derivatives with respect to the components of V^+ are bounded. Above, I have assumed $f(t, V^+)$ to be continuous in G for this first consideration. Then a local Lipschitz constant L_L can be found through

$$L_L = \sup_{t \in [t_i, t_j]} \left\| \frac{\partial f(t, V^+(t))}{\partial V^+(t)} \right\|_2$$

for those parts $[t_i, t_j]$ where f is differentiable and has bounded partial derivatives with respect to the components of V^+.
At the points t^* of non-differentiability, a possible value for the local Lipschitz constant L_L is the maximum of the local Lipschitz constants calculated for the left-hand and the right-hand interval with respect to t^*.
A global Lipschitz constant L then can be found through

$$L = \max L_L$$

Of course, those considerations are only correct if the partial derivatives of $f(t, V^+)$ with respect to V^+ are bounded over $[0, tEnd]$. In the IVP (5.3) they all only consist of sums of δ_j and μ_{jk}. As I have supposed $|\delta_j(t)| \leq M_0$ and $|\mu_{jk}(t)| \leq M_0$ $\forall j \in \{1, \ldots, n\}$, $\forall k \in \{1, \ldots, n\}$, $k \neq j$, $t \in [0, tEnd]$ the derivatives all are bounded which completes the considerations concerning the Lipschitz condition.

To summarise the results of the preceding considerations, the validity of the Picard-Lindelöf Theorem requires

- upper bounds ρ_1 for $|t|$, $\sqrt{n}\rho_2$ for $\|V^+(t) - V^+(tEnd)\|_1$ and M_0 for each of the functions $|a'_{jj}(t)|$, $|a_{jk}(t)|$, $|\delta_j(t)|$ and $|\mu_{jk}(t)|$

as well as

- f being a continuous function in t.

Then the existence of a unique solution of the IVP (5.3) can be guaranteed.

5.1.5 Calculating the real heights of the contributions

Contributions are always paid when remaining in specified states $j \in S$. So in (5.1) and (5.2) they are represented by negative values of the functions a'_{jj} and $\Delta a^{(d)}_{jj}$ which means

that the premiums can be paid continuously or at discrete points of time or both. For the following considerations any information about the states in which contributions have to be paid is not required and therefore left out.
When considering an insurance contract for which certain benefits are defined, the heights of the contributions for this contract are determined by the scale of the benefits. Nevertheless, it is possible to prescribe sort of a course of the premium payments. For better understanding, I will introduce some symbols:

- Let $\pi'(t)$, $t \in [0, tEnd]$ denote the function describing the real heights of the continuously effected contributions and
- let $\Delta\pi^{(d)}(t)$ denote the real contribution heights paid at discrete points of time t.

This means that for a given insurance contract the scale of the contributions paid through $\pi'(t)$ and $\Delta\pi^{(d)}(t)$ corresponds to the scale of the defined benefits. By contrast,

- let $b'(t)$, $t \in [0, tEnd]$ stand for the function describing the course of the continuously effected contribution payments given apart from some proportional factor pF and
- let $\Delta b^{(d)}(t)$ denote the contribution heights paid at discrete points of time t given apart from pF.

Then for an insurance contract having uniquely defined benefits, the course of the contribution payments can be given through $b'(t)$, $t \in [0, tEnd]$ and $\Delta b^{(d)}(t)$ for all discrete points t. Obviously,

$$\pi'(t) = b'(t)\, pF$$

and

$$\Delta\pi^{(d)}(t) = \Delta b^{(d)}(t)\, pF.$$

So for a course of the contribution payments given apart from pF, this factor can be calculated by using the equivalence principle: At time 0, the expected discounted benefits of a contract are equal to the expected discounted contributions, in symbols

$$\mathbf{E}_0[\text{discounted benefits}] = \mathbf{E}_0[\text{discounted real contributions}].$$

This is equivalent to

$$\underbrace{\mathbf{E}_0[\text{discounted benefits}]}_{(1)} = \underbrace{\mathbf{E}_0[\text{discounted contributions apart from } pF]}_{(2)}\, pF.$$

(1) can be determined by calculating the reserve at time 0 for the given contract by means of (5.1) and (5.2) when leaving out any contribution payments ($b'(t)$ and $\Delta b^{(d)}(t)$). Analogously, (2) can be calculated by leaving out all benefits and multiplying by (-1) (because in contrast to the positive benefit payments the contribution payments have negative values). Finally, pF can be determined by calculating $\frac{(1)}{(2)}$.

5.2 Solving Thiele's differential equation by means of a discretisation method

To calculate the course of the reserve $V^+(t)$ for a given insurance contract over the insurance period $[0, tEnd]$, Thiele's differential equation ((5.1) and (5.2)) respectively the corresponding sequence of IVPs has to be solved by using a discretisation method.

5.2.1 Choosing a suitable numerical method for solving TDE

The numerical methods for solving ordinary differential equations have already been described in Chapter 4. To decide which discretisation schemes are qualified for solving an IVP it is useful to distinguish between a stiff and a non-stiff differential equation: A system of ordinary differential equations $y' = f(t, y)$ is called stiff if its derivative with respect to y, i.e. the function $\frac{df(t,y)}{dy}$ has eigenvalues λ_i with real part $Re\ \lambda_i \ll 0$, possibly besides eigenvalues of moderate size. For TDE (5.1) in matrix notation

$$\frac{d}{dt}V^+(t) = A(t) + B(t)\,V^+(t)$$

with

$$A(t) = -\begin{pmatrix} a'_{00}(t) \\ a'_{11}(t) \\ \vdots \\ a'_{(n-1)(n-1)}(t) \end{pmatrix} - \begin{pmatrix} \sum_{k\neq 0} \mu_{0k}(t)a_{0k}(t) \\ \sum_{k\neq 1} \mu_{1k}(t)a_{1k}(t) \\ \vdots \\ \sum_{k\neq (n-1)} \mu_{(n-1)k}(t)a_{(n-1)k}(t) \end{pmatrix}$$

and

$$B(t) = \begin{pmatrix} c_0 & -\mu_{01}(t) & \dots & -\mu_{0(n-1)}(t) \\ -\mu_{10}(t) & c_1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -\mu_{(n-2)(n-1)}(t) \\ -\mu_{(n-1)0}(t) & \dots & -\mu_{(n-1)(n-2)}(t) & c_{n-1} \end{pmatrix}$$

where

$$c_j = \delta_j'(t) + \sum_{k \neq j} \mu_{jk}(t), \quad j \in 0, \ldots, n-1$$

as well as

$$V^+(t) = \begin{pmatrix} V_0^+(t) \\ V_1^+(t) \\ \vdots \\ V_{n-1}^+(t) \end{pmatrix}$$

the derivative with respect to V^+ is the matrix $B(t)$.

Provided that the functions μ_{jk}, $j \neq k$ are bounded that way that do not cause eigenvalues λ_i with $Re\ \lambda_i \ll 0$, TDE constitutes a non-stiff differential equation and therefore it is not necessary to use an implicit discretisation method. As adaptive step size is very difficult or even impossible to apply on multi-step methods, the explicit Runge-Kutta schemes constituting the most important class of the explicit one-step methods seem to be a good choice for solving TDE provided that the described conditions are fulfilled. An additional possibility to determine whether the chosen method is qualified or not is to observe the course of the step sizes: If the step size becomes very small (which can also be detected by observing that the calculation takes a comparatively long time) the applied method does not seem to be the best choice.

One of the requirements guaranteeing an existing and unique solution is the continuity of the right-hand side of (5.1) over each sub-interval. When applying a discretisation method (5.1) has to fulfill further conditions concerning the smoothness of the solution function $V^+(t)$ and depending on the order of convergence of this method.
The improved Euler method (cf. (4.40) and (4.41)) as a simple example of the Runge-Kutta methods is convergent of order 2 and requires $V^+(t)$ to be three times continuously differentiable. So what are the conditions the right-hand side of (5.1) has to fulfill in order to guarantee three times continuous differentiability of $V^+(t)$? The first three derivatives of $V^+(t)$ are:

$$\frac{dV^+(t)}{dt} = f(t, V^+(t)) \tag{5.4}$$

$$\frac{d^2V^+(t)}{dt^2} = \underbrace{\frac{\partial f(t, V^+)}{\partial t}}_{I} + \underbrace{\frac{\partial f(t, V^+)}{\partial V^+}}_{II} \underbrace{\frac{dV^+(t)}{dt}}_{III} \tag{5.5}$$

$$\frac{d^3V^+(t)}{dt^3} = \underbrace{\frac{\partial^2 f(t,V^+)}{\partial t^2}}_{IV} + 2\,\underbrace{\frac{\partial^2 f(t,V^+)}{\partial V^+\,\partial t}}_{V}\,\underbrace{\frac{dV^+(t)}{dt}}_{=III} + \underbrace{\frac{\partial^2 f(t,V^+)}{\partial (V^+)^2}}_{VI}\left(\underbrace{\frac{dV^+(t)}{dt}}_{=III}\right)^2 + \underbrace{\frac{\partial f(t,V^+)}{\partial V^+}}_{=II}\,\underbrace{\frac{d^2V^+(t)}{dt^2}}_{=I+II\cdot III} \tag{5.6}$$

In (5.5) and (5.6),

- I is continuous if the derivatives of the functions $a'_{jj}(t)$, $a_{jk}(t)$, $\mu_{jk}(t)$ and $\delta'_j(t)$ with respect to t exist and if they are continuous.
- II is equal to the matrix $B(t)$ described before. As all elements of $B(t)$ consist of the functions $\mu_{jk}(t)$ and $\delta'_j(t)$ and as those functions are assumed to be continuous in each sub-interval, the continuity of II is fulfilled.
- III is equal to the right-hand side of TDE (5.1) and therefore assumed to be continuous in each sub-interval.
- IV is continuous if the second derivatives of the functions $a'_{jj}(t)$, $a_{jk}(t)$, $\mu_{jk}(t)$ and $\delta'_j(t)$ with respect to t exist and if they are continuous.
- V is continuous if the derivatives of the functions $\mu_{jk}(t)$ and $\delta'_j(t)$ with respect to t exist and if they are continuous.
- VI is equal to 0 as the right-hand side of TDE is linear in V^+.

So to guarantee three times continuous differentiability of $V^+(t)$, the components $a'_{jj}(t)$, $a_{jk}(t)$, $\mu_{jk}(t)$ and $\delta'_j(t)$ at least have to be twice continuously differentiable. If there are points where this condition is not fulfilled, the integration interval has to be split up at these points, again (cf. Sections 5.1.2 and 5.1.3).

The generalised statement for an explicit Runge-Kutta method of order p is that $V^+(t)$ has to be $p+1$ times continuously differentiable, i.e. that the functions $a'_{jj}(t)$, $a_{jk}(t)$, $\mu_{jk}(t)$ and $\delta'_j(t)$ have to be p times continuously differentiable to guarantee the convergence when applying this method.

5.2.2 Settings for solving TDE by means of a numerical method

When using a discretisation method for solving an ordinary differential equation, certain settings can be specified:

- First, a numerical algorithm has to be chosen. For the case of TDE, this will be one of the Runge-Kutta methods.

- Secondly, if adaptive step size shall be used, a desired error level consisting of an absolute and a relative error bound can be chosen (cf. Section 4.4, (4.46)). Adaptive step size only allows specifying a desired error level for the local error but not for the global one. Nevertheless, an a-posteriori estimate of the global error can be obtained as described in Section 4.4.

- Every algorithm starts with an initial step size. In order to avoid numerous rejections of unsuitable initial step sizes, the user has to specify a value for the initial step size which, according to the properties of the differential equation, he regards as more or less suitable.

5.3 The realisation in the C programming language

Besides the detailed theoretical considerations concerning the calculation of the reserve for the Markovian model, an important constituent part of this diploma thesis is the practical application of TDE: I have written a programme to calculate the course of the reserve $V^+(t)$ over $t \in [0, tEnd]$ for given life insurance contracts by using the C programming language. In this chapter, I will describe the structure of this routine and explain all important components.

Concerning the applied discretisation methods, I have used the GSL ODEIV package of the GNU Scientific Library (GSL), a free software library written in C for numerical calculations in applied mathematics and science. In the following subsection, I will briefly describe the functions of this package and how they are used.

5.3.1 The GSL functions for solving ordinary differential equations

The GSL ODEIV package provides several functions for solving ordinary differential equation (ODE) initial value problems. The user can choose between a couple of different solving methods on the one hand and whether to leave the initial step size unchanged over the integration interval or to use adaptive step size control on the other hand.

The ODE system

The routines are able to solve a first order system of n dimensions

$$\frac{dy_i}{dt} = f_i(t, y_1(t), \ldots, y_n(t))$$

together with some initial value $(t_0, y(t_0))$.
All information about the system that the routines need is stored in a variable of type `gsl_odeiv_system` having four components:

- `int (*function) (double t, const double y[], double dydt[], void *params);`
  is a pointer to a function describing the right-hand side of the differential equation system depending on the arguments t and y and the parameters $params$, and storing it in the array $dydt$. The function returns an integer value in order to inform whether the calculation has been successful or not.

- `int (*jacobian) (double t, const double y[], double *dfdy, double dfdt[], void *params);`
  is a pointer to a function calculating

  $$\frac{\partial f_i(t, y, params)}{\partial t}, \quad i \in \{1 \ldots dim\}$$

  and the Jacobian matrix

  $$J_{ij} = \frac{\partial f_i(t, y, params)}{\partial y_j}, \quad i \in \{1 \ldots dim\}, \quad j \in \{1 \ldots dim\}$$

  and storing them in the arrays $dfdt$ and $dfdy$ respectively. Also this function returns an integer value indicating whether the calculation has been successful or not. Only some of the implicit algorithms solving the system use the Jacobian matrix.

- `size_t dimension;`
  is the dimension of the differential equation system.

- `void *params;`
  is a pointer to data of an unspecified type so that any differential equation system can be described using the required parameters stored in an individual data structure.

The stepping functions

The lowest level functions are the stepping functions which, starting from initial values $y[]$ at time t, calculate the solution of the ODE at time $t+h$ as well as a value estimating the resulting local error.
After having defined the ODE system the user has to choose an algorithm for solving the system, e.g. `const gsl_odeiv_step_type *T = gsl_odeiv_step_rk8pd` for choosing the embedded Runge-Kutta Prince-Dormand (8,9) method. The GSL ODEIV package provides several explicit and two implicit Runge-Kutta methods as well as three other implicit methods.
`gsl_odeiv_step *s = gsl_odeiv_step_alloc(T,dim)` creates a new stepper object which contains all required information concerning the chosen algorithm for the calculation of one step.
By using the command `gsl_odeiv_step_apply` one step from t to $t+h$ is calculated for given initial values $y[]$, the ODE system information $\&sys$ and the created stepper object s. The called routine first calculates the result of the step and then a value estimating the local error occurred during this step. In most methods, error estimation is either done by step doubling or by using two algorithms of different order. Then the difference between the two results gives an approximation of the local error occurred during this step.
When calculating without adaptive step size control, the values of the estimation errors don't have any consequences. In the following paragraph, it is described how error estimation can be used to adjust the step size in order to keep the local error on each step within a desired error level.

Adaptive step size control

By using adaptive step size control it is possible to obtain an approximation of the solution of the ODE having a certain accuracy on each step. The proceeding is exactly the same as described in Section 4.4.
The GSL ODEIV package provides two control types to determine the desired error level ϵ_i:

- The standard control type:
  When using the standard control type, the desired error level ϵ_i for each component is calculated by taking into account the absolute and relative errors eps_abs and eps_rel and the factors a_y and a_dydt for scaling $y(t)$ and $y'(t)$:

$$\epsilon_i = \epsilon_{abs} + \epsilon_{rel}(a_y|y_i| + a_{dydt}h|y_i'|)$$

`gsl_odeiv_control *c = gsl_odeiv_control_standard_new(double eps_abs, double eps_rel, double a_y, double a_dydt);` is the command for creating a new step size control object for the standard control type.
There are two special types of the standard control object: The commands `gsl_odeiv_control *c = gsl_odeiv_control_y_new(double eps_abs, double eps_rel);` and `gsl_odeiv_control *c = gsl_odeiv_control_yp_new(double eps_abs, double eps_rel);` are equivalent to creating a new step size control object for the standard control type with $a\_y = 1$, $a\_dydt = 0$ and $a\_y = 0$, $a\_dydt = 1$ respectively. So the first one will keep the local error on each step within an absolute error of *eps_abs* and a relative error of *eps_rel* with respect to the solution $y_i(t)$ and the second one within an absolute error of *eps_abs* and a relative error of *eps_rel* with respect to the solution $y_i'(t)$.

- The scaled control type:
  `gsl_odeiv_control *c = gsl_odeiv_control_scaled_new(double eps_abs, double eps_rel, double a_y, double a_dydt, const double scale_abs[], size_t dim);` creates a new step size control object for the scaled control type. In this case the formula for ϵ_i is

$$\epsilon_i = \epsilon_{abs} scale\_abs_i + \epsilon_{rel}(a_y|y_i| + a_{dydt}h|y_i'|)$$

  So for the scaled control type the desired error level is calculated by including an absolute error which is scaled for each component.

Using the evolution object

The highest level component is the evolution function which combines the information returned by the stepping function and the control function and - according to the results - continues with the next step or by repeating the last step for some smaller step size in order to calculate the solution of the ODE over the considered interval.

`gsl_odeiv_evolve *e = gsl_odeiv_evolve_alloc(dim);` creates a new evolution object which contains all required information for using the evolution function for the calculation.
To calculate the solution of the ODE system in the interval $[t, tEnd]$ by using the evolution object the user has to call the function `gsl_odeiv_evolve_apply` in a loop:

```
while (t<tEnd)
{
```

```
    int status = gsl_odeiv_evolve_apply(e, c, s, &sys, &t, tEnd, &
        h, y);
    if (status != GSL_SUCCESS)
        break;
}
```

The arguments are

- pointers e to the evolution object, c to the step size control object, s to the stepper object and $\&sys$ to the ODE system information
- the end of the considered integration interval $tEnd$ and a pointer $\&t$ to the actual point of the integration interval
- a pointer $\&h$ to the initial step size
- and initial values y[].

If any function called by the evolution function returns a value indicating that the calculation has not been successful the step will be aborted and the return value *status* will inform the user about the error.

5.3.2 The structure of the programme

1. As I have decided to use the GSL ODEIV package to calculate a solution of TDE, the following settings are required to be specified in the function main.c:

   - A discretisation method has to be chosen (one of the explicit Runge-Kutta methods).
   - A control type together with the required values eps_abs, eps_rel, a_y and a_dydt has to be chosen to specify the desired error level.
   - The initial step size has to be specified.

   Furthermore, the commands for creating the corresponding stepper object, control object and evolution object have to be executed.

2. All information of the right-hand side of TDE is contained in the variable sys:

   - The function $function$ describes the right-hand side of TDE, i.e. the formula (5.1). For given $t \in [0, tEnd]$, function value $y[]$ (corresponding to the vector

$V^+(t)$) and the required parameters specifying the right-hand side of TDE it calculates the corresponding value of the differential equation

- As the function *jacobian* is not used for the explicit Runge-Kutta methods the pointer to this function is set to 0 in my programme.
- the dimension of TDE, i.e. the number of states
- *params* is a pointer to a structure which contains all parameters specifying the right-hand side of TDE. So $function$ represents sort of a frame which is filled by using the elements of the parameter structure in order to determine the values of this function

3. The connection between the functions of the GSL ODEIV package and the information of TDE contained in *sys* is established by means of the function `gsl_odeiv_evolve_apply`. In a loop until the end of the considered integration interval (or sub-interval), this function induces one step of the calculation by using the GSL ODEIV functions. For this purpose, the information concerning the integration interval, the initial values $y[]$, the right-hand side of TDE in *sys* as well as the chosen settings are delivered to the GSL ODEIV functions.

5.3.3 The parameters

The component *params* in *sys* is a pointer to the data structure `allParameters_t`. This data structure consists of all parameters which are required to calculate TDE. In other words, the individual properties of any insurance contract are given through the elements of the structure `allParameters_t`. `allParameters_t` contains the following information:

- the number of insured persons belonging to the contract
- the number of states per person, in the following also referred to as "single states"
- the number of states in the contract, also referred to as "contract states". If there is only one insured person the number of states per person and the number of contract states are the same.
- the age of the insured persons at the beginning of the insurance period, i.e. at $t = 0$ (the "initial ages")
- the insurance period

- the parameters required for calculating the transition intensities as well as the function calculating the transition intensities $\mu_{jk}(t)$, $j \neq k$

- the function calculating the value of the payments $a'_{jj}(t)$ and $a_{jk}(t)$ as specified for the contract as well as a second function of this type which always returns the value 0 (the second function is required for calculating the real heights of the contributions). A third function of this type is either set to the first or the second function (more details will be given afterwards)

- the upper bound *upper* of the actual integration sub-interval which is needed for calculating $a_{jk}(t)$

- the proportional factor pF (contributions are given apart from this factor)

- a function describing all points of discontinuity of TDE as well as the value of $\Delta a_{jj}^{(d)}(t)$ and $\Delta \delta_j^{(d)}(t)$. Furthermore a variable $jumpLists$ of data structure `threeJumpLists_t` containing three lists: The first list $j1$ contains the same information as the function describing the points of discontinuity. The second list $j2$ only contains the points of discontinuity having negative values of $\Delta a_{jj}^{(d)}(t)$. The third list $j3$ only consists of one entry having a negative value of $\Delta a_{jj}^{(d)}(t)$. $j3$ is only needed if the character of the insurance contract is taken into account and will be described in Chapter 6.

- the function calculating the values of the force of interest $\delta'_j(t)$ as well as the function representing the discounting function which is needed for calculating $a_{jk}(t)$

- a variable that can have two values indicating whether the character of the insurance contract is taken into account or not. All details concerning the character of an insurance contract are left out in this chapter but are described in Chapter 6.

So when starting to execute the programme a variable of type `allParameters_t` is created and every element of it is given a special value or function characterising the chosen insurance contract. The user can modify the command assigning a certain insurance contract to this variable by choosing one of the available insurance contracts that I have already implemented. Furthermore, it is possible to write new functions and choose appropriate values for the parameters in order to create a new insurance contract that afterwards can be chosen in the main programme.

5.3.4 The insured persons and the states of the contract

If only one insured person is part of an insurance contract, the states of the contract describe situations for this single person. So in this case the single states are identically equal to the contract states.
By contrast, when having more than one insured person the contract states represent situations of the whole group of insured persons. Obviously, these contract states depend on the single states of all persons: For example, the single states $*$ ("alive") and † ("dead") lead to the contract states $**$, $*\dagger$, $\dagger*$ and $\dagger\dagger$ for a contract having two insured persons.
Of course, in TDE only the contract states are considered. But as the transition intensities for the contract states will be composed of the transition intensities for the single states, the introduction of the number of insured persons and of the single states is indispensable: The function calculating the transition intensities for the contract states requires the information concerning the insured persons and the single states.

5.3.5 The payment functions

The payment functions in TDE (5.1) and (5.2) consist of the continuously effected payments $a'_{jj}(t)$ for remaining in state j, the transition payments $a_{jk}(t)$ for the transitions $j \to k$, $j \neq k$ and the discrete payments $\Delta a^{(d)}_{jj}(t)$ for remaining in state j.
As the discrete payments $\Delta a^{(d)}_{jj}(t)$ cause discontinuities they are described below in the paragraph which explains how to deal with the points of discontinuity listed in $jumpLists$.

In my programme, the payments $a'_{jj}(t)$ and $a_{jk}(t)$ are together described in one function: According to the values of j and k, the function distinguishes between $j = k$ and $j \neq k$ and calculates the required value depending on j, k and t. The different functions calculating $a'_{jj}(t)$ and $a_{jk}(t)$ for the different implemented examples can be found in $ajk.c$.
If the transition payment $a_{jk}(t)$ is defined not to be paid immediately at t but at a later time, instead of the full value only the discounted value of $a_{jk}(t)$ has effect on the reserve. The function `calculatePaymentValue` in `payments.c` checks whether a discounted value is defined or not and calculates the desired result by using the discounting function. Of course, the fact that a transition payment is effected at a later time causes a point of discontinuity at this later date (cf. Section 5.1.3). So the integration has to be interrupted at this point which has to be indicated in the function describing all points of discontinuity as well as the discrete values $\Delta a^{(d)}_{jj}(t)$ and $\Delta \delta^{(d)}_{j}(t)$.

This function is used to create the list $j1$ of discontinuities which is arranged in descending order with respect to t. $j1$ consists of three different types of entries:

- points of discontinuity where a discrete payment $\Delta a_{jj}^{(d)}(t)$ is effected - then the entry also contains the value of this payment.
- points of discontinuity where a discrete interest $\Delta\delta_j^{(d)}(t)$ is realised - then the entry also contains the value $\Delta\delta_j^{(d)}(t)$
- points of discontinuity caused by discontinuities of one of the functions $a'_{jj}(t)$, $a_{jk}(t)$, $\delta'_j(t)$ and $\mu_{jk}(t)$ or of one of their k-th derivatives (as far as they are required for the convergence of the applied discretisation method). In this case, the entry signifies that there is no discrete value which has to be added.

5.3.6 The interest functions

The functions modelling the interest in TDE (5.1) and (5.2) consist of the continuously realised interest $\delta'_j(t)$ and the interest $\Delta\delta_j^{(d)}(t)$ realised at discrete points of time. Furthermore, there is a discounting function $v(t_1, t_2)$ calculating the value at time $t1$ of one unit paid at time t_2. The discounting function is only used to determine the discounted values of $a_{jk}(t)$ which are defined to be paid at a later time. The parameter *upper* in $allParameters\_t$ is required here to indicate the point of time the transition payment $a_{jk}(t)$ is paid at. $a_{jk}(t)\, v(t, upper)$ then is the corresponding discounted value.

For reasons of simplification, for all the implemented examples modelling different insurance contracts, $\delta'_j(t)$ is a constant function $\delta'_j(t) = 0.03$ and there is no interest realised at discrete points of time.

5.3.7 Modelling the transition intensities

The chosen model

Assigning priorities while working on my diploma thesis, I did not choose the question of how modelling the transition intensities as a problem of prime importance. Having properly integrated the functions calculating the transition intensities into the programme is much more essential than greatly fitted values produced by those functions.
Starting from a classical insurance contract with one insured person and two states per person, i.e. “alive” and “dead”, I just needed mortality intensities as transition intensities. I chose a simple model together with estimated parameters found in Section 3.4

of [Møller and Steffensen, 2007]: Møller and Steffenson describe the Gompertz-Makeham model in which the mortality intensities are calculated through

$$\mu(x+t) = \alpha + \beta c^{x+t} \tag{5.7}$$

where x denotes the age of the insured person, t the time since the beginning of the insurance period and α, β and c are the parameters estimated for the Danish population in 2003.
The estimated values for males and females are:

	α	β	c
male	0.000134	0.0000353	1.1020
female	0.000080	0.0000163	1.1074

By calculating the one-year mortality probabilities

$${}_1p_x = exp\left(-\int_0^1 \mu(x+\tau)d\tau\right) = exp\left(-\int_0^1 (\alpha + \beta c^{x+\tau})d\tau\right) = exp\left(-\alpha - \beta\frac{c^{x+1}-c^x}{ln(c)}\right)$$

I could compare the Danish values to the Austrian ones using the life table ÖStt2000_ 2002 and realise that they conform very well for $x \in [30, 80]$. Nevertheless, I use them as well for ages below 30 and above 80.
For insurance contracts having more than one possible transition I also chose a very simple way of calculating the transition intensities: Every intensity is deduced from the mortality intensity and just multiplied with some proportional factor.
In contracts having more than one insured persons the probabilities and therefore the intensities are assumed to be independent.

One possibility to obtain better fitted transition intensities is to fit a model to Austrian biometric data and to estimate the parameters for this model. Furthermore, the effects of dependence because of having more than one insured person could be taken into account more carefully. But as it has already been mentioned this has not been the central point of this diploma thesis.

The intensity parameters

$parInt$ is an array of dimension $pers$ x $nPers$ which contains elements of the data structure `parameters_Int`. The data structure `parameters_ Int` consists of the parameters α, β and c and the array p of dimension $nPers$.

	state 0	state 1	state 2
person 0			
person 1			

Table 5.1: $parInt$ of dimension 2 x 3

Table 5.1 shows the array for $pers = 2$ and $nPers = 3$. Every element $parInt[i][j]$ of the array ($i \in \{0, 1\}$ denoting the insured person and $j \in \{0, 1, 2\}$ denoting the state) contains values α, β, c and $p = \{p_0, p_1, p_2\}$ leading to the transition intensities for person i:

$$\mu^i_{j\bullet}(x + t) = \alpha + \beta c^{x+t} \tag{5.8}$$

is the transition intensity for person i to leave state j.

$$\mu^i_{jk}(x + t) = \left(\alpha + \beta c^{x+t}\right) p_k \tag{5.9}$$

is the transition intensity for person i from state j to state k under the condition that person i leaves state j. $\mu^i_{jj}(x + t)$ is not used in the calculation and therefore $p_j = 0$ in $parInt[i][j]$.
Obviously, the parameters α, β and c determine whether the insured person is male oder female.

The functions calculating the transition intensities

Let us first consider an insurance contract with one insured person: The number of contract states is equal to the number of states per person. So a transition from contract state j to contract state k is caused by the person's transition from the person's state j to the person's state k.
For an insurance contract containing more than one insured person, a transition from contract state j to contract state k is caused by one of the persons' transition from one person's state to another one.
For this reason it was necessary to write two types of functions for calculating the transition intensities:

1. One function calculating the transition intensity for a given person, given person's initial and following state and remaining parameters t, x and $parInt$.

2. One function having as arguments the contract state j, the contract state k and the remaining required parameters t, x and $parInt$. This function chooses the person and this person's initial and following state corresponding to the transition from contract state j to contract state k and calls the function described above.

The first function is used for every insurance contract, always called by the second function which is individual for each contract and an element of the variable of type `allParameters_t`.

5.3.8 Calculating the reserve

As it has already mentioned in Section 5.3.2, the function `gsl_odeiv_evolve_apply` induces one step of the calculation by using the GSL ODEIV functions. `gsl_odeiv_evolve_apply` is called in a loop which is stopped at every point of discontinuity indicated by one of the lists $j1$, $j2$ and $j3$. Then the discrete values are added and the loop for the next sub-interval is started.

This procedure is executed three times:

1. The list $j1$ is used, but $pF = 0$ implies that the contributions are left out in order to determine the expected discounted benefits.

2. This time, the benefits are left out by using the second of the payment functions in `allParameters_t` which always returns the value 0. Using the list $j2$ in which only the contribution payments are described leads to the expected value of the discounted contributions.

3. After having determined pF by dividing the expected discounted benefits by the expected discounted contributions, the reserve is calculated for the original payment function and for the original list $j1$ which leads to the desired course of the reserve.

5.4 Examples

In this section some concrete examples calculated by using the implemented programme are discussed. First, the course of the reserve for three classical insurance contracts is examined and compared to the course of the reserve for the same examples calculated using the discrete recursion. Then, the more complex case of a long-term care insurance for two insured persons is presented.

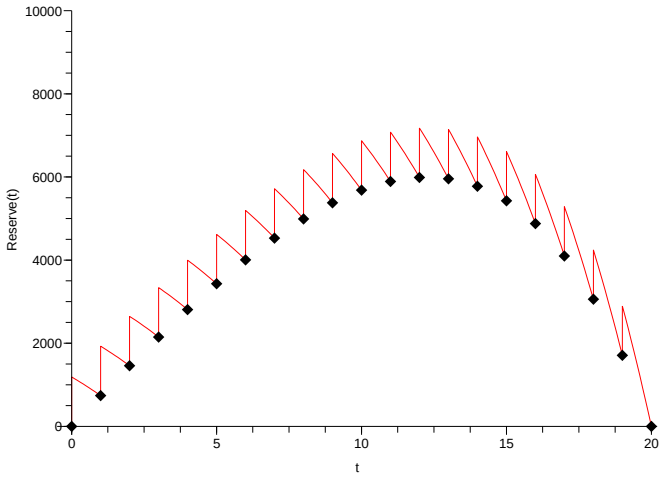

Figure 5.3: Term insurance

5.4.1 Simple classical insurance contracts

This subsection is about three simple classical insurance contracts: The course of their reserve in state 0 calculated by using the time-continuous markov model is compared to the corresponding reserve calculated by using the time-discrete markov recursion. By calculating the one year mortality probabilities based on the mortality intensities used in the time-continuous calculation (cf. formula (5.8)), the corresponding time-discrete calculation can be made.

Term insurance

This example considers a term insurance for a male insured person of initial age 50. There are two contract states "alive"$(*)$ and "dead"$(\dagger)$. The insurance period is 20 years. The defined death benefit is 100000, paid at the end of the year the insured person dies in. Contributions are of constant height and paid at the beginning of each year $k = 1, \ldots, 20$. The force of interest is constant with $\delta'_j(t) = 0.03$ over the insurance period.
As shown in Figure 5.3, the values of the reserve in state 0 calculated in the time-continuous model are equal to the values of the corresponding reserve in the time-discrete model at the end of each year k, $k = \{0, \ldots, 20\}$. At times k, i.e. at integer values $\lceil t \rceil$, the reserve in state 0 in the time-continuous model has a discontinuity of positive height: contributions are paid. Basically, the continuous change of the reserve is influenced by the expected

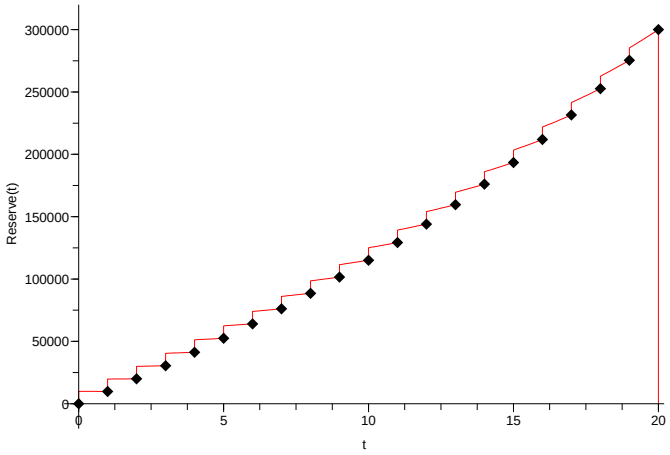

Figure 5.4: Term and endowment insurance

death benefits which are subtracted. This is the reason why the reserve decreases between the discrete contribution payments over the whole insurance period.
The function `setInsuranceModel01M` describing this example in C can be found in `parameters.c` in the appendix.

Term and endowment insurance

An endowment insurance for a male insured person of initial age 50 is considered. Again, there are two contract states "alive"$(*)$ and "dead"$(\dagger)$. The insurance period is 20 years. The defined death benefit is 100000, paid at the end of the year the insured person dies in. The defined endowment benefit is 300000, paid at the end of the insurance period. Contributions are of constant height and paid at the beginning of each year $k = 1, \ldots, 20$. The force of interest is constant with $\delta'_j(t) = 0.03$ over the insurance period.

Figure 5.4 shows the typical increasing course of the reserve in state 0 for the term and endowment insurance. Again, the values of the reserve in state 0 calculated in the time-continuous model are equal to the values of the corresponding reserve in the time-discrete model at the end of each year k, $k = \{0, \ldots, 20\}$.
This example is described by the function `setInsuranceModel02M` and can be found in `parameters.c` in the appendix.

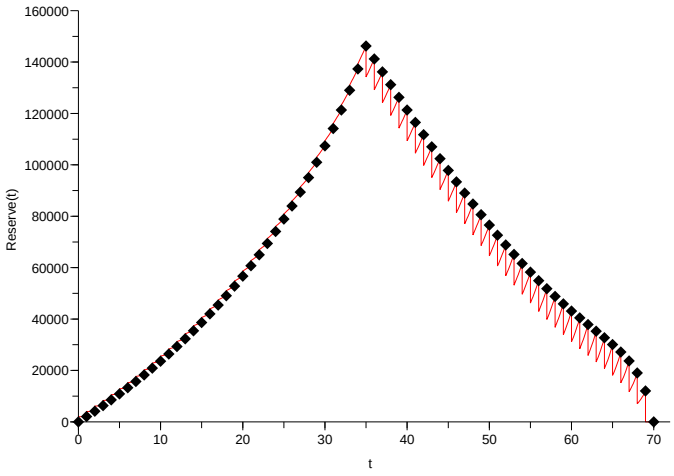

Figure 5.5: Annuity insurance

Annuity insurance

The third classical example considers an annuity insurance for a male insured person of initial age 30. There are two contract states "alive"($*$) and "dead"($\dagger$). The insurance period is 70 years. Contributions are of constant height and paid at the beginning of each year $k = 1, \ldots, 35$ whereas for $k = 36, \ldots, 70$, annuity benefits of height 12000 are paid at the beginning of each year k, always provided that the contract remains in state $*$. The force of interest is constant with $\delta'_j(t) = 0.03$ over the insurance period.

Figure 5.5 shows the typical course of the reserve in state 0 of an annuity insurance: In the interval where the contributions are paid, the course of the reserve is similar to that in Figure 5.4. From $t = 35$ on annuity benefits are paid. Between those payments, the reserve is increasing because of the interest.
Also this example can be found in `parameters.c` in the appendix: `setInsuranceModel04M`

5.4.2 Long-term care insurance

As an example of a complex insurance contract I have chosen a long-term care insurance for two insured persons. Figure 5.6 shows the different states and the possible transitions for this example. There are three single states for each insured person: "active"($*$), "in need of care"(C) and "dead"($\dagger$). 9 contract states which are shown in the figure result from these single states. The arrows symbolising the possibility of remaining in one state

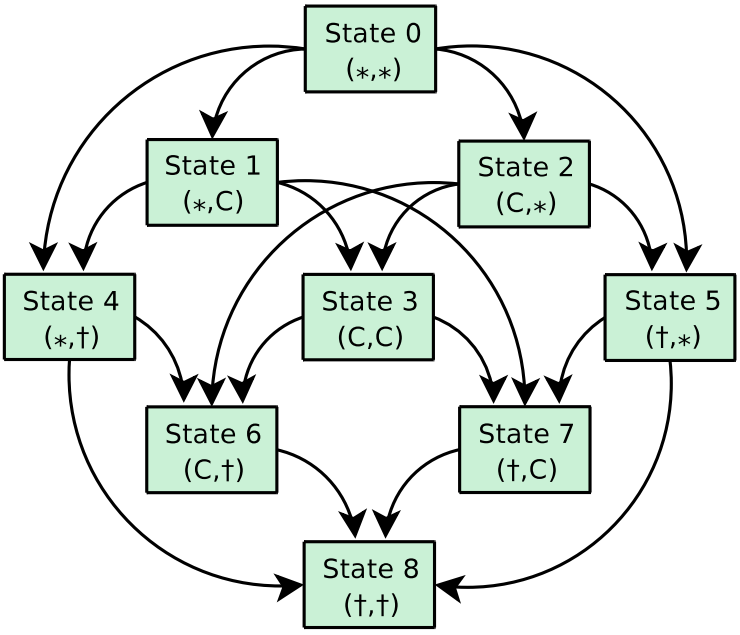

Figure 5.6: Long-term care insurance insurance

have been left out in the figure in order to keep it clear and easy to understand.
Two possible ways of specifying an insurance contract for this state space have been considered:

1. A symmetric version described in the function `setInsuranceModel2Lives02a`:
   The payments for remaining in one state are defined as follows:

   - state 0: Contributions are paid
   - states 1 and 2: One of the two insured persons is still "active", the other one is in need of care. The active one cares for the other one. So in these states, no contributions but also no benefits are paid.
   - state 3: Both insured persons are in need of care. A "full" annuity is paid.
   - states 6 and 7: One of the two insured persons has already died, the other one is in need of care. A "half" annuity is paid.
   - states 2, 5 and 8: No contributions and no benefits are paid.

   In addition to these payments also transition payments are defined: For each of the following transitions

   - $0 \to 4$

- $1 \to 4$
- $0 \to 5$
- $2 \to 5$

a transition benefit is paid. For all other transitions no benefits are paid.

2. A non-symmetric version is described in the function `setInsuranceModel2Lives02b`: In this case, the first of the insured persons has sort of a maintaining function. The payments for remaining in one state are defined as follows:

   - states 0 and 4: The "maintaining" person is active, the other one is "active", too, or has already died. So contributions are paid (by the "maintaining" person).
   - state 3: Both insured persons are in need of care. A "full" annuity is paid.
   - states 1, 6 and 7: Either one of the two insured persons is in need of care and the other one has already died. Or the "maintaining" person is in need of care whereas the other one is still "active". Then the "active" one cares for the "maintaining" person. In both cases, a "half" annuity is paid because the "maintaining" person can no longer earn money.
   - states 2, 5 and 8: No contributions and no benefits are paid.

   As in the symmetric version, additional transition payments are defined for the following transitions:

   - $0 \to 4$
   - $1 \to 4$
   - $0 \to 5$
   - $2 \to 5$

   For all other transitions no benefits are paid.

Chapter 6

Taking into account the character of an insurance contract

For every life insurance contract, a certain risk is covered by the insurance company: Benefit payments which are defined for certain events (e.g. for remaining in a state or for specified transitions) are effected if these events occur. So depending on how the benefits are defined in a contract, the risk for the insurer can consist in

- a specified transition $j \to k$ of the contract if benefits are defined for this transition (transition benefits) or for remaining in state k (annuities)
- remaining in the actual state j if benefits (annuities) are defined for this state

In classical life insurance, the so called *capital at risk (car)* at time t is defined as follows:

$$\begin{aligned} car(k, k+1) &= \text{death benefit} - \text{reserve(k)} \\ car(k, k+1) < 0 &\Leftrightarrow \text{endowment character} \\ car(k, k+1) > 0 &\Leftrightarrow \text{death character} \end{aligned}$$

as there are no states and only one-year transition probabilities and as the death benefit is a constant value over the whole insurance period.

For the time-continuous Markovian model the capital at risk can be defined analogously:

$$car(t, j, k) = a_{jk}(t) + V_k^+(t) - V_j^+(t)$$

So for this model, the expressions "endowment character" and "death character" are modified as follows:

$$\begin{aligned} car(t,j,k) < 0 &\Leftrightarrow \text{"character of remaining"} \\ car(t,j,k) > 0 &\Leftrightarrow \text{"transition character"} \end{aligned}$$

In Section 5.1.2 it has been mentioned that using the equivalence principle in order to determine the contributions is justified if a sufficiently large number of insurance contract is issued and if safety margins are included in the contributions. These safety margins are obtained by modifying the transition intensities:
If $car(t,j,k) > 0$ the risk for the insurer consists in the transition $j \to k$ of the contract. So the corresponding transition intensity $\mu_{jk}(t)$ is increased by a certain factor in order to increase the probability for this transition in the calculation. Analogously, for $car(t,j,k) < 0$ the corresponding transition intensity $\mu_{jk}(t)$ is decreased by a certain factor in order to increase the contract's probability of remaining in state j in the calculation.
In other words, the probabilities for effectively realising the benefit payments are increased which implies that the value of the expected discounted benefits is greater than it would be when using the non-modified transition intensities. Following the equivalence principle, also the value of the expected discounted contributions is greater – now safety margins are included in the contributions.

6.1 The modified model

The modification of the transition intensities as described in the previous section implies that in TDE the transition intensities $\mu_{jk}(t)$, $j \neq k$ are replaced by $\mu_{jk}(t, V_j^+(t), V_k^+(t))$, $j \neq k$ which do not only depend on t but also on V^+. The proof of Theorem 3.18 (TDE) starts with expressing the reserve $W_j^+(t)$, $j \in S$ as follows:

$$\begin{aligned} W_i^+(t) &= \left(1 - \left(\sum_{\substack{j \in S \\ i \neq i}} \mu_{ij}(t)\right) \Delta t\right) W_i^+(t+\Delta t) + \sum_{\substack{j \in S \\ i \neq i}} \mu_{ij}(t)\, \Delta t\, W_j^+(t+\Delta t) \\ &+ v(t) \left(a'_{ii}(t)\Delta t + \left(\sum_{\substack{j \in S \\ j \neq i}} \mu_{ij}(t) a_{ij}(t)\right) \Delta t\right) + o(\Delta t) \end{aligned} \tag{6.1}$$

In (6.1), Theorem 3.17 is applied for $u = t + \Delta t$, $\Delta t > 0$ and each transition probability is approximated by the corresponding transition intensity multiplied by the length Δt of the considered interval. So starting from $W^+(t + \Delta t)$ at time $t + \Delta t$, the reserve $W^+(t)$

at time t is approximated by using the values at time t of the functions a'_{ii}, a_{ij}, μ_{ij} and v, i.e. the values $a'_{ii}(t)$, $a_{ij}(t)$, $\mu_{ij}(t)$ and $v(t)$. Because in the proof $\lim_{\Delta t \searrow 0}$ is considered it is also possible to use the values $a'_{ii}(t+\Delta t)$, $a_{ij}(t+\Delta t)$, $\mu_{ij}(t+\Delta t)$ and $v(t+\Delta t)$ at time $t+\Delta t$ as it is done when numerically solving TDE in my programme.
Replacing $\mu_{ij}(t+\Delta t)$ by $\mu_{ij}(t+\Delta t, W_i^+(t+\Delta t), W_j^+(t+\Delta t))$ means that the Markovian model defined in Chapter 3 is locally modified: In each step $t+\Delta t \to t$, instead of the defined transition intensities $\mu_{ij}(t+\Delta t)$ different intensities $\mu_{ij}(t+\Delta t, W_i^+(t+\Delta t), W_j^+(t+\Delta t))$ are used. So together with Δt, the new intensities still can be used to locally approximate the transition probabilities as in (6.1). By contrast, global probability functions cannot be determined:
When calculating with the simple intensities $\mu_{ij}(t+\Delta t)$ global probability functions $p_{ij}(s,u)$ are uniquely defined by the intensities and they hold for any insurance contract defined for the Markovian model. But when using intensities $\mu_{ij}(t+\Delta t, W_i^+(t+\Delta t), W_j^+(t+\Delta t))$, the individual values of the reserve are required to calculate the values of the intensity functions and therefore also the values of the probability functions. So probability functions can only be determined for a concrete path of $W^+(t)$.

6.2 The practical application of TDE

As it has been described in the preceding section, the intensity functions

$$\mu_{jk}(t, V_j^+(t), V_k^+(t))$$

now depend on further parameters $V_j^+(t)$ and $V_k^+(t)$. This means that V^+ as a parameter appears in the functions μ_{jk} on the right-hand side of TDE. This causes additional requirements concerning the functions μ_{jk} in order to guarantee the existence and uniqueness of a solution and the convergence of an applied discretisation method. When regarding the derivatives of $V^+(t)$ with respect to t (cf. (5.4) – (5.6)), there are only important changes in those terms containing derivatives of $f(t, V^+)$ with respect to V^+.

- II: To ensure the existence and the continuity of this term, the functions μ_{jk} have to be continuously differentiable with respect to V^+. The same holds for term V.

- VI: This term is continuous if the functions μ_{jk} are twice continuously differentiable.

So the convergence for an explicit Runge-Kutta method of order p can be guaranteed if, in addition to the requirements described in Section 5.2, the functions μ_{jk} are p times

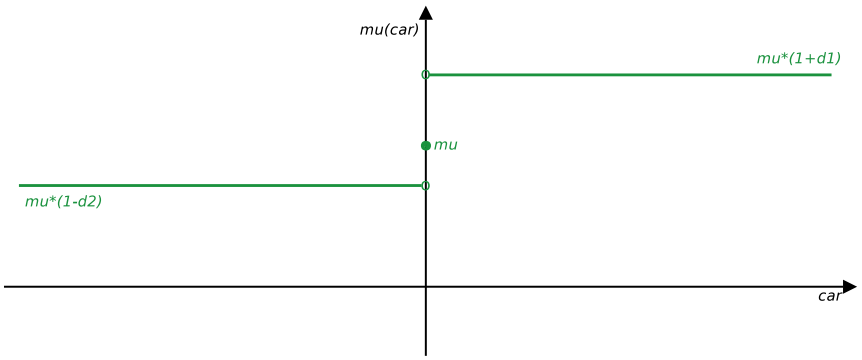

Figure 6.1: $\mu_{jk}(t, car(t,j,k))$ for fixed j, k and t - first version

continuously differentiable with respect to V^+.

The values of the functions $\mu_{jk}(t, V_j^+(t), V_k^+(t))$ can be determined as follows: For fixed states j and k and also for fixed t, the value of $\mu_{jk}(t)$ (without character!) constitutes the starting point for the calculation. Depending on the value of $car(t,j,k) = a_{jk}(t) + V_k^+(t) - V_j^+(t)$, the value of $\mu_{jk}(t, V_j^+(t), V_k^+(t))$ is lower, equal or greater than $\mu_{jk}(t)$:

$$\mu_{jk}(t, V_j^+(t), V_k^+(t)) = \begin{cases} \mu_{jk}(t)\,(1+d_1) & \text{if } car(t,j,k) > 0 \\ \mu_{jk}(t) & \text{if } car(t,j,k) = 0 \\ \mu_{jk}(t)\,(1-d_2) & \text{if } car(t,j,k) < 0 \end{cases}$$

with positive values d_1 and d_2 (see also Figure 6.1).

Obviously, according to this definition μ_{jk} is not even continuous in V^+ and therefore does not fulfill the condition of being p times continuously differentiable. In this case, the required smoothness cannot be achieved by splitting up the integration interval at these points before solving TDE because as the functions μ_{jk} depend on the concrete path of V^+, their points of discontinuity as well as the discontinuities of their derivatives are unknown. So the functions μ_{jk} have to be modified in order to fulfill the required smoothness conditions. Such a modified intensity function for fixed j, k and t is shown in Figure 6.2. The corresponding definition is:

$$\mu_{jk}(t, V_j^+(t), V_k^+(t)) = \begin{cases} \mu_{jk}(t)\,(1+d_1) & \text{if } car(t,j,k) > c \\ spline(car(t,j,k)) & \text{if } |car(t,j,k)| < c \\ \mu_{jk}(t)\,(1-d_1) & \text{if } car(t,j,k) < -c \end{cases}$$

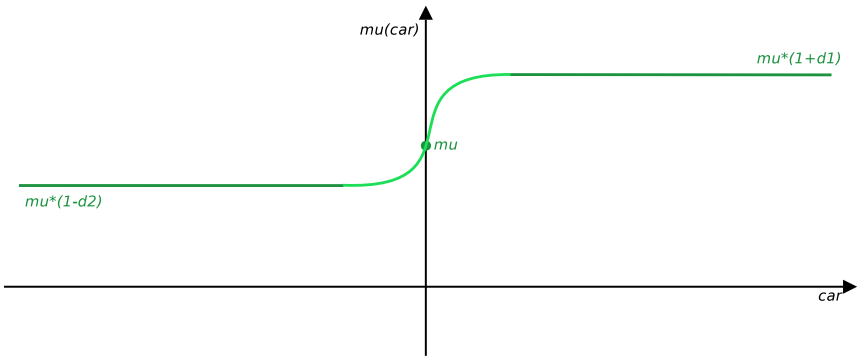

Figure 6.2: $\mu_{jk}(t, car(t, j, k))$ for fixed j, k and t - second version

with positive values d_1 and d_2.

So in the interval $(-c, c)$ there is a spline function connecting the points $(-c, \mu_{jk}(t)(1 - d_2))$ and $(c, \mu_{jk}(t)(1 + d_1))$ fulfilling the required smoothness conditions. The coefficients $a_0, \ldots, a_m$ for this function

$$s(x) = a_m x^m + a_{m-1} x^{m-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

are determined by solving a system of linear equations representing the desired function values and values of the derivatives at the points $-c$ and c. So to guarantee p times continuous differentiability of μ_{jk} with respect to V^+, $2\,(p + 1)$ conditions are required: p conditions concerning the values of the k-th derivative $k = 1, \ldots, p$ and one condition for the function value, all at $-c$, and the same number of conditions at c.

6.3 The realisation in C

6.3.1 The variable *useChar*

As it has already been mentioned in Section 5.3.3, one element of the data structure `allParameters_t` is the variable *useChar* which indicates whether the character of the insurance contract is taken into account or not. So by setting *useChar* to 1, the reserve is calculated by using intensities which depend on the capital at risk. This means that whenever an intensity is calculated, the value of the corresponding capital at risk is determined in order to calculate the value of the intensity as described in the previous section.

As in the GSL ODEIV package the highest order Runge-Kutta method is of order 8, a spline function guaranteeing 8 times continuous differentiability of μ_{jk} with respect to V^+ has been implemented. so no matter which Runge-Kutta method is used, the smoothness requirements are always fulfilled.

It shall be mentioned here that the values $-c$ and c as well as d_1 and d_2 are arbitrarily chosen. Any estimation of concrete values for safety margins has not been part of this diploma thesis. The programme just has the functionality of being able to increase or decrease the calculated intensities by chosen values.

6.3.2 Calculating the real contribution heights

The calculation of the real contribution heights by determining the proportional factor pF as

$$pF = \frac{\mathbf{E}_0[\text{discounted benefits}]}{\mathbf{E}_0[\text{discounted contributions apart from } pF]}$$

(cf. Section 5.1.5) does not necessarily lead to the desired result $V_0^+(0) = 0$ when taking into account the character of an insurance contract. The reason is that the course of the capital at risk over the time can change when leaving out the contributions or the benefits. So when taking into account the character of an insurance contract, another algorithm has to be used to calculate the real contribution heights. The *regula falsi method* has been chosen for this purpose.

The algorithm works as follows: The reserve $res_{0,0}(pF)$ in the initial state 0 at time 0 is regarded as a function of pF. By applying the regula falsi method (see Figure 6.3), the root of $res_{0,0}(pF)$, i.e. the proportional Factor pF^* supplying $res_{0,0}(pF^*) = 0$ can be calculated. Two values pF_1 and pF_2 such that $res_{0,0}(pF_1) > 0$ and $res_{0,0}(pF_2) < 0$ are necessary to start the regula falsi method.
Assuming that $res_{0,0}(pF)$ as a function of the proportional factor is continuous and monotone in pF over the interval (pF_1, pF_2), the intermediate value theorem implies the existence of one single root in the interval (pF_1, pF_2).
A first initial value pF_1 for which $res_{0,0}(pF_1) > 0$ is obtained by calculating the expected discounted benefits, i.e. by calculating the reserve for $pF_1 = 0$.
For the second initial value pF_2, $res_{0,0}(pF_2) < 0$ is required which means that there are comparatively more premiums than benefits.

$$pF_2 = \frac{\mathbf{E}_0[\text{discounted benefits}]}{\mathbf{E}_0[\text{discounted contributions apart from } pF]}$$

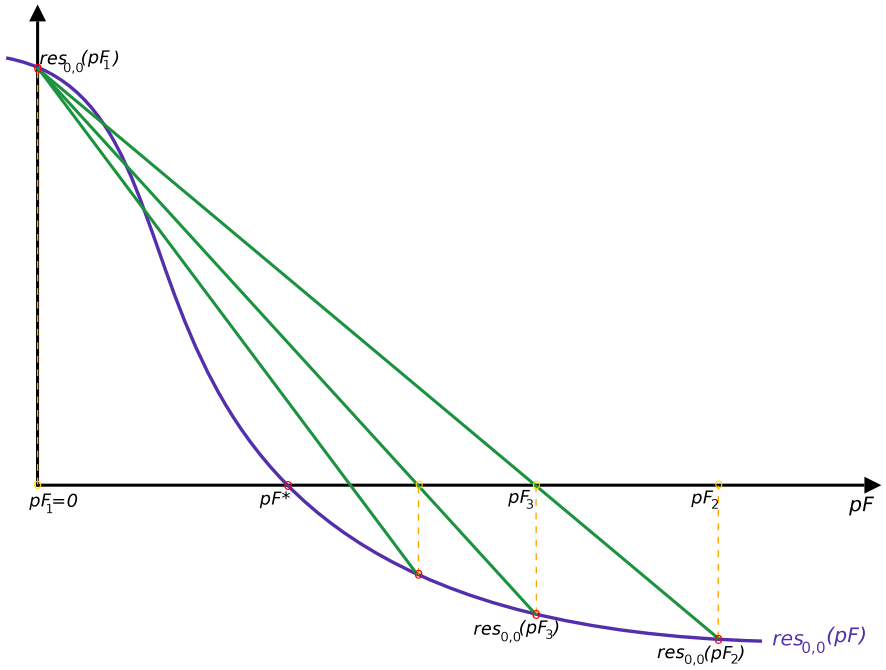

Figure 6.3: Regula falsi method

does not necessarily lead to $res_{0,0}(pF_2) < 0$. So it is assumed that the "sum" of the contributions

$$\int_0^{tEnd} b'(t)dt + \sum_t \Delta b^{(d)}(t)$$

is paid at the end of the insurance period $tEnd$. Then calculating the expected discounted value

$$\mathbf{E}_0[\text{discounted modified contributions apart from } pF]$$

and

$$pF_2 = \frac{\mathbf{E}_0[\text{discounted benefits}]}{\mathbf{E}_0[\text{discounted modified contributions apart from } pF]}$$

leads to the desired second initial value pF_2 for which $res_{0,0}(pF_2) < 0$.

The third list $j3$ in $jumpLists$ is a list which consists of one single entry: the contribution payment described above which is used for calculating pF_2.

Having pF_1 with $res_{0,0}(pF_1) > 0$ and pF_2 with $res_{0,0}(pF_2) < 0$, the regula falsi method works as follows: It computes the root of the line passing through the two points $(pF_1, res_{0,0}(pF_1))$ and $(pF_2, res_{0,0}(pF_2))$ as an approximation of the root of $res_{0,0}(pF)$. The

calculated root approximation is taken as pF_3 in order to start the program calculating the reserve of the contract again. If $|res_{0,0}(pF_3)| < eps$ for a chosen $eps > 0$, pF_3 is the searched value of pF^* and the routine can be terminated. Otherwise,

$$\begin{aligned} res_{0,0}(pF_3) \geq 0 \quad &\Rightarrow \quad pF_1 := pF_3 \\ res_{0,0}(pF_3) < 0 \quad &\Rightarrow \quad pF_2 := pF_3 \end{aligned}$$

gives the new points to start the regula falsi again.

Appendix A

Source code

The following source code written in the C programming language contains all functions which are required to calculate the course of the reserve as it has been explained in the previous parts of this diploma thesis. Concerning the different examples that I have implemented, only those are given here which have been described in Section 5.4. The complete code containing further examples is available on request.

```
/* File main.c */

#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_odeiv.h>
#include <float.h>
#include <math.h>
#include "ajk.h"
#include "boolean.h"
#include "calcReserve.h"
#include "funcJac.h"
#include "intensities.h"
#include "jump.h"
#include "more.h"
#include "parameters.h"
#include "payments.h"
#include "regulaFalsi.h"
#include "vDelta.h"

int main (void)
{  /*************** Definition of the required variables **************/

   // Index variables:
   long i = 0;
   boolean_t ok = 0;
```

```
// Data structure containing the parameters:
allParameters_t *allP = allocateAllParameters();

// For using the GSL functions:
const gsl_odeiv_step_type * T = 0;

gsl_odeiv_step    *s = 0;
gsl_odeiv_control *c = 0;
gsl_odeiv_evolve  *e = 0;

gsl_odeiv_system sys = {0, 0, 0, 0};

double tEnd   = 0.0; // the end of the integration interval
double t      = 0.0; // the beginning of the integration interval; t is index variable
double h      = 0.0; // initial step size

double *y     = 0; // solution function

boolean_t (*setModel) (allParameters_t *) = 0;

/********************** Assigning the variables **********************/
/*************** The user can modify these settings!! ***************/

// Choose a model representing the considered insurance contract:
setModel = &setInsuranceModel02M;
ok = setModel(allP);
if (ok != 1) // != 1: an error has occurred in setModel!
{
   fprintf(stderr, "Error (setModel)! \n");
   return -1;
}

// GSL: Discretisation method, control object, ...:
T = gsl_odeiv_step_rk8pd; // choose an algorithm
s = gsl_odeiv_step_alloc (T, allP->nModel);
c = gsl_odeiv_control_y_new (1e-09, 0.0);
             // choose a control object, eps_abs and eps_rel
e = gsl_odeiv_evolve_alloc (allP->nModel);

// gsl_odeiv_system components (modification not required!):
sys.function  = func;
sys.jacobian  = 0; // jac is not required for the explicit algorithms
sys.dimension = allP->nModel;
sys.params    = allP;

// The integration interval; initial step size:
tEnd = 0.0;              // beginning of the integration interval
t    = allP->insPeriod; // end of the integration interval; t is index variable
h    = -1.0;             // choose an initial step size

/************************** End of choice **************************/
/********************** Calculation and result **********************/

y = calcReserveAndPremFactor(allP, tEnd, &t, &h, s, c, e, &sys);
```

```
    printf("\n\n");
    printf("Result: \n");
    for (i = 0; i <= allP->nModel - 1; i++)
        printf("y[%ld] = %f  ", i, y[i]);
    printf("\n");
    printf("premFactor = %f \n", allP->premFactor);

    /********************* Freeing allocated memory *********************/
    if (y)
    {
        free(y);
        y = 0;
    }
    freeAllParameters(allP);

    gsl_odeiv_evolve_free (e);
    gsl_odeiv_control_free (c);
    gsl_odeiv_step_free (s);

    return 0;
}
```

```
/* File ajk.c
 * contains the definition of all functions concerning the continuous payments ajj'(t) and
      ajk(t)
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "ajk.h"

// function calculateConstValueA
double calculateConstValueA(double t, long j, long k)
{  // j and k denote the contract states
    if ((j == 0) && (k == 1))
        return 100000.0;
    else
        return 0.0;
} // end calculateConstValueA


// function calculateConstValueAZero
double calculateConstValueAZero(double t, long j, long k)
{  // j and k denote the contract states
    return 0.0; // no benefits
} // end calculateConstValueAZero
```

```
// function calculateConstValueA2Lives02
double calculateConstValueA2Lives02(double t, long j, long k)
{  // j and k denote the contract states
   if (((j == 0) || (j == 1)) && (k == 4))
      return 100000.0;
   else if (((j == 0)||(j == 2)) && (k == 5))
      return 100000.0;
   // These payments are death benefits: One person remains active while the second one
   // dies after having been active (j==0)or in need of care (j==1 or j==2)
   else
      return 0.0; // no benefits for any other transition
} // end calculateConstValueA2Lives02
```

```
/* File ajk.h
 * contains the declaration of all functions concerning the continuous payments ajj'(t)
     and ajk(t)
 */

double calculateConstValueA(double t, long j, long k);
double calculateConstValueAZero(double t, long j, long k);
double calculateConstValueA2Lives02(double t, long j, long k);
```

```
/* File boolean.h
 * contains the definition of the data structure boolean_e and its definition as boolean_t
 */

#ifndef BOOLEAN_H
#define BOOLEAN_H

typedef enum boolean_e
{
   false = 0,
   true  = 1
} boolean_t;

#endif // BOOLEAN_H
```

```
/* File calcReserve.c
```

```
 * contains the definition of the functions calcReserveAndPremFactor and calculateReserve
 */

#include <stdio.h>
#include <float.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_odeiv.h>
#include "ajk.h"
#include "boolean.h"
#include "calcReserve.h"
#include "funcJac.h"
#include "jump.h"
#include "more.h"
#include "regulaFalsi.h"
#include "vDelta.h"

// function calcReserveAndPremFactor
double *calcReserveAndPremFactor(allParameters_t *allP, double tEnd, double *time, double
    *h,
                        gsl_odeiv_step *s, gsl_odeiv_control *c, gsl_odeiv_evolve *e,
                        gsl_odeiv_system *sys)
{
   long j = 0;
   double *y1 = 0;
   double *y2 = 0;
   double *y3 = 0;
   double *yH = 0;
   double singleP = 0.0;
   double premFactor  = 0.0; // required for useChar == 0
   double premFactor1 = 0.0;
   double premFactor2 = 0.0;
   double eps_prem = 1e-07;


   // Allocating memory for the vector y of dimension nModel
   // and initialising by 0 provides the right initial values y(tEnd).
   y1 = calloc(allP->nModel, sizeof(double));
   if (y1 == 0)
   {
      fprintf(stderr, "Error (allocating memory for y1)!\n");
      exit(-1);
   }
   y2 = calloc(allP->nModel, sizeof(double));
   if (y2 == 0)
   {
      fprintf(stderr, "Error (allocating memory for y2)!\n");
      free(y1);
      exit(-1);
   }
   y3 = calloc(allP->nModel, sizeof(double));
   if (y3 == 0)
   {
      fprintf(stderr, "Error (allocating memory for y3)!\n");
```

```
    free(y1);
    free(y2);
    exit(-1);
}
yH = calloc(allP->nModel, sizeof(double));
if (yH == 0)
{
    fprintf(stderr, "Error (allocating memory for yH)!\n");
    free(y1);
    free(y2);
    free(y3);
    exit(-1);
}

// Calculating the expected discounted benefits:
premFactor1 = allP->premFactor;
y1 = calculateReserve(y1, allP, tEnd, time, h, s, c, e, sys);
if (y1[0] <= DBL_EPSILON)
{
    fprintf(stderr, "No contributions => V0(0) must not be <= 0!\n");
    freeThreePointers(y1, y2, y3);
    exit(-1);
}

/* Distinguishing between two cases:
 * useChar == 0 => premFactor = E[benefits]/E[contributions]
 * useChar == 1 => calculating premFactor by means of the regula falsi method
 */

if (allP->useChar == 0)
{
    singleP = singlePremiumTimeZero(allP->jumpLists->j1);
    if (singleP < 0) // single premium payment at time 0
        premFactor = y1[0]/fabs(singleP); // because singleP is negative
    else // no single premium payment at time 0
    {
        allP->premFactor = 1.0;
        allP->jumpLists->whichList = 2;
        allP->calcValA = allP->calcValA2;
        y2 = calculateReserve(y2, allP, tEnd, time, h, s, c, e, sys);
        if (y2[0] >= DBL_EPSILON)
        {
            fprintf(stderr, "Contributions != 0, no benefits => W0(0) must not be >= 0!\n"
                );
            exit(-1);
        }
         premFactor = y1[0]/fabs(y2[0]);
    }

    allP->premFactor = premFactor;
    allP->jumpLists->whichList = 1;
    allP->calcValA = allP->calcValA1;
    y3 = calculateReserve(y3, allP, tEnd, time, h, s, c, e, sys);
```

```
    printf("useChar == 0: \n");
    for (j = 0; j <= allP->nModel - 1; j++)
       printf("y1[%ld] = %.3f  ", j, y1[j]);
    printf("\n\n");
    for (j = 0; j <= allP->nModel - 1; j++)
       printf("y2[%ld] = %.3f  ", j, y2[j]);
    printf("\n\n");
    printf("premFactor = %.3f \n", premFactor);
    for (j = 0; j <= allP->nModel - 1; j++)
       printf("y3[%ld] = %.3f  ", j, y3[j]);
    printf("\n");

    if (y1)
    {
       free(y1);
       y1 = 0;
    }
    if (y2)
    {
       free(y2);
       y2 = 0;
    }
    return y3;
}
else // useChar == 1
{
    allP->premFactor = 1.0;
    allP->jumpLists->whichList = 3;
    allP->calcValA = allP->calcValA2;
    yH = calculateReserve(yH, allP, tEnd, time, h, s, c, e, sys);
    if (yH[0] >= DBL_EPSILON)
    {
       fprintf(stderr, "Contributions != 0, no benefits => W0(0) must not be >= 0!\n");
       exit(-1);
    }
        premFactor2 = y1[0]/fabs(yH[0]);

        allP->premFactor = premFactor2;
    allP->jumpLists->whichList = 1;
    allP->calcValA = allP->calcValA1;
    y2 = calculateReserve(y2, allP, tEnd, time, h, s, c, e, sys);

    printf("useChar == 1: \n");
        printf("premFactor1 = %.3f \n", premFactor1);
    for (j = 0; j <= allP->nModel - 1; j++)
       printf("y1[%ld] = %.3f  ", j, y1[j]);
    printf("\n\n");
        printf("E[discounted single payment at t=insPeriod]: \n");
    for (j = 0; j <= allP->nModel - 1; j++)
       printf("yH[%ld] = %.3f  ", j, yH[j]);
    printf("\n\n");
        printf("premFactor2 = %.3f \n", premFactor2);
    for (j = 0; j <= allP->nModel - 1; j++)
       printf("y2[%ld] = %.3f  ", j, y2[j]);
```

```
      printf("\n\n");

          if (y2[0] > eps_prem)
      {
         fprintf(stderr, "More contributions than benefits => W0(0) must not be > 0! \n");
         exit(-1);
      }
      else if (fabs(y2[0]) < eps_prem) // y2[0] is close enough to 0 => calculation
          terminated!
      {
         if (y1)
         {
            free(y1);
            y1 = 0;
         }
             if (y3)
         {
            free(y3);
            y3 = 0;
         }
         return y2;
      }
      else
      {
         y3 = regulaFalsi(premFactor1, y1, premFactor2, y2, y3, allP, tEnd,
                          time, h, s, c, e, sys, eps_prem);
         return y3;
      }
   }
} // end calcReserveAndPremFactor


// function calculateReserve
double *calculateReserve(double *y, allParameters_t *allP, double tEnd, double *time,
    double *h,
                         gsl_odeiv_step *s, gsl_odeiv_control *c, gsl_odeiv_evolve *e,
                         gsl_odeiv_system *sys)
{
   long j = 0;
   double t = *time;
   jump_t *ju = 0;
   double tStop = 0.0; // upper bound of the actual sub-interval
                  // The calculation of the differential equation is interrupted at time
                      tStop.

   if (allP->jumpLists->whichList == 1)
      ju = allP->jumpLists->j1;
   else if (allP->jumpLists->whichList == 2)
      ju = allP->jumpLists->j2;
   else if (allP->jumpLists->whichList == 3)
      ju = allP->jumpLists->j3;
   else
   {
      fprintf(stderr, "Invalid jumpList! \n");
```

```
    exit(-1);
}

if ((ju != 0) && (ju->time > t)) // The jumps are arranged in descending order with
     respect to t.
{
    fprintf(stderr, "Error: Jump at t>insPeriod! \n");
    exit (-1);
}

while (1)
{
    if (ju == 0)
        tStop = tEnd;
    else
        tStop = ju->time; // If tStop == insPeriod == t, the following while loop is not
            executed
                          // now. The jump (see below) is effected before.
    while (t > tStop)
    {
        int status = gsl_odeiv_evolve_apply(e, c, s, sys, &t, tStop, h, y);
        if (status != GSL_SUCCESS)
            break;
    }

         printf("before jump:  t=%.2f, ", t);
         for (j = 0; j <= allP->nModel-1; j++)
        printf("y[%ld]=%.3f  ", j, y[j]);
    printf("\n");

    if (ju != 0)
    {
        for (j = 0; j <= allP->nModel-1; j++)
        {
            if (ju->height[j] < 0.0) // contribution payment
                y[j] += ju->height[j] * allP->premFactor;
            else
                y[j] += ju->height[j];
        }
    }

         printf("after jump: t=%.2f, ", t);
         for (j = 0; j <= allP->nModel-1; j++)
        printf("y[%ld]=%.3f  ", j, y[j]);
    printf("\n");

    if ((tStop == tEnd) && (ju == 0))
    {
        printf("\n\n");
        break;
    }

         if (ju->trans == 1)          // Only if tStop == point of time a time-displaced
        allP->upper = ju->time; // transition payment is effected at => allP->upper is
```

```
                                    // set to this point of time time.
        ju = ju->next;
    }

    return y;
} // end calculateReserve
```

```
/* File calcReserve.h
 * contains the declaration of the functions calcReserveAndPremFactor and calculateReserve
 */

#include "parameters.h"

double *calcReserveAndPremFactor(allParameters_t *allP, double tEnd, double *time, double
    *h,
                        gsl_odeiv_step *s, gsl_odeiv_control *c, gsl_odeiv_evolve *e,
                        gsl_odeiv_system *sys);
double *calculateReserve(double *y, allParameters_t *allP, double tEnd, double *time,
    double *h,
                        gsl_odeiv_step *s, gsl_odeiv_control *c, gsl_odeiv_evolve *e,
                        gsl_odeiv_system *sys);
```

```
/* File funcJac.c
 * contains the definition of the function func which represents Thiele's differential
 * equation. The function jac is not required for the explicit discretisation methods
 * which are used for Thiele's differential equation and it need not be defined here.
 */

#include <gsl/gsl_errno.h>
#include <gsl/gsl_matrix.h>
#include "ajk.h"
#include "funcJac.h"
#include "parameters.h"
#include "payments.h"
#include "vDelta.h"

// function func
int func (double t, const double y[], double f[], void *params)
{
    allParameters_t *allP = (allParameters_t *)params;
    long j = 0;
    long k = 0;
    double sum1 = 0.0;
    double sum2 = 0.0;
```

```
double sum3 = 0.0;
double payment = 0.0;
double *intensityJ = 0; // for an array of length nModel

intensityJ = (double *)calloc(allP->nModel, sizeof(double));
if (intensityJ == 0)
{
   fprintf(stderr, "Error (allocating memory for intensityJ)!\n");
   exit(-1);
}
// For index j in the following loop, intensityJ[k] will be the value of the intensity
// for the transition from contract state j to contract state k.

// Forming nModel equations:
for (j = 0; j <= allP->nModel - 1; j++)
{
   f[j] = 0.0; // Initialising the j-th equation:
   // For each of the nModel equations first three sums (sum1, sum2, sum3) are
       calculated
       // and then equation f[j] is composed of these sums and the remaining factors.

   // 1st sum:
   sum1 = calculatePaymentValue(t, j, j, allP);
   for (k = 0; k <= allP->nModel - 1; k++)
   {
      if (k != j)
      {
         payment = calculatePaymentValue(t, j, k, allP);
         intensityJ[k] = allP->calcInt(t, j, k, allP->parInt, allP->x, allP->useChar, y
             , payment);
         sum1 += intensityJ[k] * payment;
      }
   }

   // 2nd sum:
   for (k = 0; k <= allP->nModel - 1; k++)
      if (k != j)
         sum2 += intensityJ[k];

   // 3rd sum:
   for (k = 0; k <= allP->nModel - 1; k++)
      if (k != j)
         sum3 += intensityJ[k] * y[k];

       // Calculating f[j]:
   f[j] = (-1)*sum1 + (allP->delta + sum2)*y[j] - sum3;

   // Reset:
   sum1 = 0.0;
   sum2 = 0.0;
   sum3 = 0.0;
   payment = 0.0;
   for (k = 0; k <= allP->nModel-1; k++)
      intensityJ[k] = 0.0;
```

```
    }

    if (intensityJ != 0)
    {
        free(intensityJ);
            intensityJ = 0;
    }

    return GSL_SUCCESS;
}
```

```
/* File funcJac.h
 * contains the declaration of the function func
 */

int func (double t, const double y[], double f[], void *params);
```

```
/* File intensities.c
 * contains the definition of all functions concerning the transition intensities
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "intensities.h"
#include "vDelta.h"

/* Function allocateParametersInt:
 * allocates memory for a two dimension array (pers x nPers) of variables of the data
 * structure parametersInt_t. Furthermore allocates memory for an array of length nPers
 * inside each of those variables.
 */
parametersInt_t **allocateParametersInt(long pers, long nPers)
{
    long i = 0;
    long j = 0;

    parametersInt_t **parInt = calloc(pers, sizeof(parametersInt_t *));
    if (parInt == 0)
    {
        fprintf(stderr, "Error (allocating memory for parInt)!\n");
        exit(-1);
    }
```

```
   for (i = 0; i <= pers-1; i++)
   {
      parInt[i] = calloc(nPers, sizeof(parametersInt_t));
      if (parInt[i] == 0)
      {
         fprintf(stderr, "Error (allocating memory for parInt[%ld])!\n", i);
         exit(-1);
      }
       for (j = 0; j <= nPers-1; j++)
      {
         parInt[i][j].p = calloc(nPers, sizeof(double));
         if (parInt[i][j].p == 0)
         {
            fprintf(stderr, "Error (allocating memory for parInt[%ld][%ld].p)!\n", i, j);
            exit(-1);
         }
      }
   }

   return parInt;
} // end allocateParametersInt


// Function setParametersInt
boolean_t setParametersInt(parametersInt_t **parInt, long i, long j, long nPers, double
    alpha,
                           double beta, double c, double *p)
{
   // Parameters of the person no. i for the transition from single state j to  single
       state k
   long k = 0;

   parInt[i][j].alpha = alpha;
   parInt[i][j].beta  = beta;
   parInt[i][j].c     = c;
   for (k = 0; k <= nPers-1; k++)
      parInt[i][j].p[k] = p[k];

   return true;
} // end setParametersInt


/* Function calculateIntensity1Person
 * for the transition from single state j to single state k
 * of the person no. i having initial age x[i]
 */
double calculateIntensity1Person(double t, long i, long j, long k, parametersInt_t **
    parInt,
                                 long *x, boolean_t useChar, double capAtRisk)
{
   double c = 1.0;
   double d = 0.1;
   double intensity = (parInt[i][j].alpha + parInt[i][j].beta
                       * pow(parInt[i][j].c, (x[i]+t))) * parInt[i][j].p[k];
```

```
   if (useChar == 0)
      return intensity;
   else
   {
      if (capAtRisk < (-1)*c)
         return intensity * (1-d); // "Character of remaining"
      else if (capAtRisk > c)
         return intensity * (1+d); // Transition charakter
      else
         return splineOfDegree17(intensity, c, d, capAtRisk);
   }
} // end calculateIntensity1Person


/* Function calculateIntensityModel01M
 * for the transition from contract state j to contract state k
 * This function can be used for all insurance contracts having only one insured person.
 */
double calculateIntensityModel01M(double t, long j, long k, parametersInt_t **parInt, long
    *x,
                                  boolean_t useChar, const double *y, double payment)
{
   double capAtRisk = 0.0; // capital at risk

   if (useChar == 1)
      capAtRisk = payment + y[k] - y[j];
   return calculateIntensity1Person(t, 0, j, k, parInt, x, useChar, capAtRisk);
} // end calculateIntensityModel01M


/* Function calculateIntensityModel2Lives02
 * for the transition from contract state j to contract state k
 * [0 = (*,*), 1 = (*,P), 2 = (P,*), 3 = (P,P),  4 = (*,+), 5 = (+,*),
 *  6 = (P,+), 7 = (+,P), 8 = (+,+)]
 */
double calculateIntensityModel2Lives02(double t, long j, long k, parametersInt_t **parInt,
                           long *x, boolean_t useChar, const double *y, double payment)
{
   double capAtRisk = 0.0; // capital at risk

   if (useChar == 1)
      capAtRisk = payment + y[k] - y[j];

   if (((j==0) && (k==5)) || ((j==1) && (k==7)) || ((j==4) && (k==8)))
      // person no.0 dies after having been in single state "active", i.e. has a
      // transition from single state 0 to single state 2:
      return calculateIntensity1Person(t, 0, 0, 2, parInt, x, useChar, capAtRisk);

   else if (((j==0) && (k==4)) || ((j==2) && (k==6)) || ((j==5) && (k==8)))
      // person no.1 dies after having been in single state "active", i.e. has a
      // transition from single state 0 to single state 2:
      return calculateIntensity1Person(t, 1, 0, 2, parInt, x, useChar, capAtRisk);
```

```
   else if (((j==0) && (k==2)) || ((j==1) && (k==3)) || ((j==4) && (k==6)))
      // person no.0 becomes in need of care, i.e. has a
      // transition from single state 0 to single state 1:
      return calculateIntensity1Person(t, 0, 0, 1, parInt, x, useChar, capAtRisk);

   else if (((j==0) && (k==1)) || ((j==2) && (k==3)) || ((j==5) && (k==7)))
      // person no.1 becomes in need of care, i.e. has a
      // transition from single state 0 to single state 1:
      return calculateIntensity1Person(t, 1, 0, 1, parInt, x, useChar, capAtRisk);

   else if (((j==2) && (k==5)) || ((j==3) && (k==7)) || ((j==6) && (k==8)))
      // person no.0 dies after having been in need of care, i.e. has a transition
      // from single state 1 to single state 2:
      return calculateIntensity1Person(t, 0, 1, 2, parInt, x, useChar, capAtRisk);

   else if (((j==1) && (k==4)) || ((j==3) && (k==6)) || ((j==7) && (k==8)))
      // person no.1 dies after having been in need of care, i.e. has a transition
      // from single state 1 to single state 2:
      return calculateIntensity1Person(t, 1, 1, 2, parInt, x, useChar, capAtRisk);

   else
      return 0.0; // for all other transitions: intensity == 0
} // end calculateIntensityModel2Lives02


// Function freeParametersInt
void freeParametersInt(long pers, long nPers, parametersInt_t **parInt)
{
   long i = 0;
   long j = 0;
   if (parInt)
   {
      for (i = 0; i <= pers-1; i++)
      {
         if (parInt[i])
         {
            for (j = 0; j <= nPers-1; j++)
            {
               if (parInt[i][j].p)
               {
                  free(parInt[i][j].p);
                  parInt[i][j].p = 0;
               }
                    }
            free(parInt[i]);
            parInt[i] = 0;
         }
          }
      free(parInt);
      parInt= 0;
   }
} // end freeParametersInt
```

```
/* Function splineOfDegree17
 * This spline function
 * s(x) = a17*x^17+a16*x^16+a15*x^15+a14*x^14+...+a3*x^3+a2*x^2+a1*x+a0
 * fulfills the following conditions:
 *
 * s(c)   = intensity*(1+d) =: d2;
 * s(-c)  = intensity*(1-d) =: d1;
 * s'(c)  = 0; (1st derivative)
 * s'(-c) = 0; (1st derivative)
 * 2nd deriv (c)  = 0;
 * 2nd deriv (-c) = 0;
 * 3rd deriv (c)  = 0;
 * 3rd deriv (-c) = 0;
 * 4th deriv (c)  = 0;
 * 4th deriv (-c) = 0;
 * 5th deriv (c)  = 0;
 * 5th deriv (-c) = 0;
 * 6th deriv (c)  = 0;
 * 6th deriv (-c) = 0;
 * 7th deriv (c)  = 0;
 * 7th deriv (-c) = 0;
 * 8th deriv (c)  = 0;
 * 8th deriv (-c) = 0;
 * Solving the system of linear equations leads to the coefficients which are
 * described in the function.
 * splineOfDegree17 calculates s(x) at x = capAtRisk.
 */
double splineOfDegree17(double intensity, double c, double d, double capAtRisk)
{
   double spline = 0.0;
   double a0 = intensity; // = (d1+d2)/2
   double a1 = 109395.0/65536.0 * 2*d/c; // = 109395.0/65536.0 * (d2-d1)/c
   double a2 = 0.0;
   double a3 = -36465.0/8192.0 *2*d/pow(c,3);
   double a4 = 0.0;
   double a5 = 153153.0/16384.0 *2*d/pow(c,5);
   double a6 = 0.0;
   double a7 = -109395.0/8192.0 *2*d/pow(c,7);
   double a8 = 0.0;
   double a9 = 425425.0/32768.0 *2*d/pow(c,9);
   double a10 = 0.0;
   double a11 = -69615.0/8192.0 *2*d/pow(c,11);
   double a12 = 0.0;
   double a13 = 58905.0/16384.0 *2*d/pow(c,13);
   double a14 = 0.0;
   double a15 = -7293.0/8192.0 *2*d/pow(c,15);
   double a16 = 0.0;
   double a17 = 6435.0/65536.0 *2*d/pow(c,17);

   spline = a17*pow(capAtRisk,17) + a16*pow(capAtRisk,16) + a15*pow(capAtRisk,15)
          + a14*pow(capAtRisk,14) + a13*pow(capAtRisk,13) + a12*pow(capAtRisk,12)
          + a11*pow(capAtRisk,11) + a10*pow(capAtRisk,10) + a9*pow(capAtRisk,9)
          + a8*pow(capAtRisk,8) + a7*pow(capAtRisk,7) + a6*pow(capAtRisk,6)
          + a5*pow(capAtRisk,5) + a4*pow(capAtRisk,4)
```

```
          + a3*pow(capAtRisk,3) + a2*pow(capAtRisk,2) + a1*capAtRisk + a0;

   return spline;
} // end splineOfDegree17
```

```
/* File intensities.h
 * contains the declaration of all functions concerning the transition intensities as well
      as
 * the definition of the data structure parametersInt_s and its definition as
     parametersInt_t
 */

#ifndef INTENSITIES_H
#define INTENSITIES_H

#include "boolean.h"

// Definition of the data structure parametersInt_s
typedef struct parametersInt_s
{
   double alpha;
   double beta;
   double c;
   double *p;
} parametersInt_t;

parametersInt_t **allocateParametersInt(long pers, long nPers);

boolean_t setParametersInt(parametersInt_t **parInt, long i, long j, long nPers, double
    alpha,
                           double beta, double c, double *p);

double calculateIntensity1Person(double t, long i, long j, long k, parametersInt_t **
    parInt,
                                 long *x, boolean_t useChar, double capAtRisk);
double calculateIntensityModel01M(double t, long j, long k, parametersInt_t **parInt, long
     *x,
                                  boolean_t useChar, const double *y, double payment);
double calculateIntensityModel2Lives02(double t, long j, long k, parametersInt_t **parInt,
                                       long *x, boolean_t useChar, const double *y, double
                                           payment);

void freeParametersInt(long pers, long nPers, parametersInt_t **parInt);

double splineOfDegree17(double intensity, double c, double d, double capAtRisk);

#endif // INTENSITIES_H
```

```
/* File jump.c
 * contains the definition of all functions concerning the
 * discontinuities ("jumps") and the discrete payments
 */

#include <stdio.h>
#include <stdlib.h>
#include "jump.h"
#include "boolean.h"
#include "more.h"

/* Function allocateJumpEntry:
 * allocates memory fo a variable of type jump_t
 */
jump_t *allocateJumpEntry(long nModel)
{
   jump_t *entry = (jump_t *) calloc(1, sizeof(jump_t));
   if (entry == 0)
   {
      fprintf(stderr, "Error (allocateJumpEntry)!\n");
      exit(-1);
   }
   entry->height = (double *) calloc(nModel, sizeof(double));
   if (entry->height == 0)
   {
      fprintf(stderr, "Error (allocateJumpEntry)! \n");
      free(entry);
      entry = 0;
   }

   return entry;
} // end allocateJumpEntry


/* Function allocateThreeJumpLists
 * allocates memory for a variable of type threeJumpLists_t
 */
threeJumpLists_t *allocateThreeJumpLists(threeJumpLists_t *jumpLists)
{
   jumpLists = (threeJumpLists_t *) calloc(1, sizeof(threeJumpLists_t));
   if (jumpLists == 0)
   {
      fprintf(stderr, "Error(allocateThreeJumpLists)! \n");
      exit(-1);
   }

   return jumpLists;
} // end allocateThreeJumpLists


/* Function allocJumpAndSetComponents
 * Memory is allocated for a new entry of a jump list j, the
 * delivered values are assigned to the components of this entry
 * and the entry is correctly inserted into the jump list j.
```

```
 */
jump_t *allocJumpAndSetComponents(jump_t *j, double t, boolean_t trans,
                                  long nModel, double *height)
{
   long k = 0;

   jump_t *jumpEntry = allocateJumpEntry(nModel);
   jumpEntry->time  = t;
   jumpEntry->trans = trans;
   for (k = 0; k <= nModel-1; k++)
      jumpEntry->height[k] = height[k];

   // Inserting the entry into the jump list j:
   return insertJumpIdenticalTimesPossible(j, jumpEntry); // Return value: the list's
       first element
} // end allocJumpAndSetComponents


/* Function jumpsConstTermPremAnnually
 *
 * This function creates the list j which contains the discontinuities ("jumps") as well
     as
 * the discrete payments for all contract states. Each entry is created by calling the
 * function allocJumpAndSetComponents and by using the point of time t, the array height
 * containing the heights of the discrete payments and the additional information trans
 * (==1: time-displaced transition payment; ==0: discrete payment).
 *
 * In the first part of the function, the entries caused by time-displaced transition
 * payments are considered (trans==1), afterwards the entries for the discrete payments
 * (trans== 0).
 *
 * So for the time-displaced transition payments, the jump height is always equal to 0.
 * These entries only make the integration stop at these points of discontinuity in order
 * to guarantee the convergence of the applied discretisation method.
 * After having created the entries for the time-displaced transition payments, the
     variable
 * tr is poining to is set to 1. If there are no time-displaced transition payments, it is
 * not modified.
 *
 * Concerning the discrete payments, benefits (positive jump heights) and contributions
 * (negative jump heights) can be distinguished. The contributions are given apart from a
 * proportional factor premFactor which is element of allP. When adding the values of the
 * discrete payments, the jump heights are multiplied by the actual value of premFactor
 * if they are negative.
 *
 * return value: pointer to the list j
 *
 * Discrete interest and jumps of height 0 caused by discontinuities of other functions
 * than ajk(t) or by discontinuities of derivatives of the right-hand side of TDE could
 * also lead to entries in the jump list. But these possibilities have not been considered
       yet.
 */
jump_t *jumpsConstTermPremAnnually(jump_t *j, double insPeriod, long nModel, boolean_t *tr
    )
```

```
{
   double *height = 0;
   long k = 0;
   boolean_t trans = 0;

   height = allocateArrayOfDouble(nModel);

   // TIME-DISPLACED TRANSITION PAYMENTS:
   trans = 1;
   // paid at the end of the year the transition occurs in, jump heights == 0;
   for (k = 1; k <= insPeriod; k++)
      j = allocJumpAndSetComponents(j, k, trans, nModel, height);
   *tr = 1; // i.e. there are time-displaced transition payments

   // DISCRETE PAYMENTS FOR REMAINING IN A STATE:
   trans = 0;
   // Contract state 0: Time 0 until time insPeriod-1:
   // contribution payments (at the beginning of each year)
   height[0] = -1.0;
   for (k = 0; k <= insPeriod - 1; k++)
      j = allocJumpAndSetComponents(j, k, trans, nModel, height);

   freeArrayOfDouble(height);

   return j;
} // end jumpsConstTermPremAnnually


/* Function jumpsConstTermAndEndowPremAnnually
 *
 * Structure and contents of the function: cf. jumpsConstTermPremAnnually
 */
jump_t *jumpsConstTermAndEndowPremAnnually(jump_t *j, double insPeriod, long nModel,
    boolean_t *tr)
{
   double *height  = 0;
   long k = 0;
   boolean_t trans = 0;

   height = allocateArrayOfDouble(nModel);

   // TIME-DISPLACED TRANSITION PAYMENTS:
   trans = 1;
   // paid at the end of the year the transition occurs in, jump heights == 0;
   for (k = 1; k <= insPeriod; k++)
      j = allocJumpAndSetComponents(j, k, trans, nModel, height);
   *tr = 1; // i.e. there are time-displaced transition payments

   // DISCRETE PAYMENTS FOR REMAINING IN A STATE:
   trans = 0;
   // contract state 0: time 0 until insPeriod-1:
   // Contribution payments (at the beginning of each year)
   height[0] = -1.0;
   for (k = 0; k <= insPeriod - 1; k++)
```

```
      j = allocJumpAndSetComponents(j, k, trans, nModel, height);

   // contract state 0: time insPeriod: endowment benefit
   height[0] = 300000.0;
   j = allocJumpAndSetComponents(j, insPeriod, trans, nModel, height);

   freeArrayOfDouble(height);

   return j;
} // end jumpsConstTermAndEndowPremAnnually


/* Function jumpsAnnuityAnnuallyPremAnnually
 *
 * Structure and contents of the function: cf. jumpsConstTermPremAnnually
 */
jump_t *jumpsAnnuityAnnuallyPremAnnually(jump_t *j, double insPeriod, long nModel,
    boolean_t *tr)
{
   double *height  = 0;
   long k = 0;
   boolean_t trans = 0;

   height = allocateArrayOfDouble(nModel);

   // DISCRETE PAYMENTS FOR REMAINING IN A STATE:
   // Contract state 0: time 0 until 34: contribution payments (at the beginning of each
       year)
   height[0] = -1.0;
   for (k = 0; k <= 35-1; k++)
      j = allocJumpAndSetComponents(j, k, trans, nModel, height);
   // contract state 0: time 35 until insPeriod-1:
   // annuity benefits (paid at the beginning of each year)
   height[0] = 12000.0;
   for (k = 35; k <= insPeriod - 1; k++)
      j = allocJumpAndSetComponents(j, k, trans, nModel, height);

   // There are NO TIME-DISPLACED TRANSITION PAYMENTS: <=> *tr == 0
   *tr = 0;

   freeArrayOfDouble(height);

   return j;
} // end jumpsAnnuityAnnuallyPremAnnually


// Function jumps2Lives02a
jump_t *jumps2Lives02a(jump_t *j, double insPeriod, long nModel, boolean_t *tr)
{
   double *height = 0;
   long k = 0;
   boolean_t trans = 0;

   height = allocateArrayOfDouble(nModel);
```

```
   // TIME-DISPLACED TRANSITION PAYMENTS:
   trans = 1;
   // paid at the end of the year the transition occurs in, jump heights == 0;
   for (k = 1; k <= insPeriod; k++)
      j = allocJumpAndSetComponents(j, k, trans, nModel, height);
   *tr = 1; // i.e. there are time-displaced transition payments

   // DISCRETE PAYMENTS FOR REMAINING IN A STATE:
   trans = 0;
   // contract state 0 = (*,*): time 0 until insPeriod-1:
   // contribution payments (at the beginning of each year)
   height[0] = -1.0; // height[1], ..., height[nModel-1]: == 0
   for (k = 0; k <= insPeriod-1; k++)
      j = allocJumpAndSetComponents(j, k, trans, nModel, height);

   // contract state 6 = (P,+) and contract state 7 = (+,P):
   // time 0 until insPeriod-1/12: annuity benefits (paid at the beginning of each month)
   // half annuity
   height[0] = 0.0;
   height[6] = 1200.0;
   height[7] = 1200.0;
   for (k = 0; k <= insPeriod*12 - 1; k++)
      j = allocJumpAndSetComponents(j, k/12.0, trans, nModel, height);

   // contract state 3 = (P,P): time 0 until insPeriod-1/12:
   // annuity benefits (paid at the beginning of each month)
   // full annuity
   height[6] = 0.0;
   height[7] = 0.0;
   height[3] = 2400.0;
   for (k = 0; k <= insPeriod*12 - 1; k++)
      j = allocJumpAndSetComponents(j, k/12.0, trans, nModel, height);

   freeArrayOfDouble(height);

   return j;
} // end jumps2Lives02a


// Function jumps2Lives02b
jump_t *jumps2Lives02b(jump_t *j, double insPeriod, long nModel, boolean_t *tr)
{
   double *height = 0;
   long k = 0;
   boolean_t trans = 0;

   height = allocateArrayOfDouble(nModel);

   // TIME-DISPLACED TRANSITION PAYMENTS:
   trans = 1;
   // paid at the end of the year the transition occurs in, jump heights == 0;
   for (k = 1; k <= insPeriod; k++)
      j = allocJumpAndSetComponents(j, k, trans, nModel, height);
```

```
    *tr = 1; // i.e. there are time-displaced transition payments

    // DISCRETE PAYMENTS FOR REMAINING IN A STATE:
    trans = 0;
    // contract states 0 = (*,*) and 4 = (*,+):
    // time 0 until insPeriod-1: contribution payments (at the beginning of each year)
    height[0] = -1.0;
    height[4] = -1.0;
    for (k = 0; k <= insPeriod-1; k++)
       j = allocJumpAndSetComponents(j, k, trans, nModel, height);

    // contract states 1 = (*,P), 6 = (P,+) and 7 = (+,P):
    // time 0 until insPeriod-1/12: annuity benefits (paid at the beginning of each month)
    // half annuity
    height[0] = 0.0;
    height[4] = 0.0;
    height[1] = 1200.0;
    height[6] = 1200.0;
    height[7] = 1200.0;
    for (k = 0; k <= insPeriod*12 - 1; k++)
       j = allocJumpAndSetComponents(j, k/12.0, trans, nModel, height);

    // contract state 3 = (P,P): time 0 until insPeriod-1/12:
    // annuity benefits (paid at the beginning of each month)
    // full annuity
    height[1] = 0.0;
    height[6] = 0.0;
    height[7] = 0.0;
    height[3] = 2400.0;
    for (k = 0; k <= insPeriod*12 - 1; k++)
       j = allocJumpAndSetComponents(j, k/12.0, trans, nModel, height);

    freeArrayOfDouble(height);

    return j;
} // end jumps2Lives02b


/* Function buildThreeJumpLists
 *
 * First, by using the function allocateThreeJumpLists, memory is allocated for the
     variable
 * jumpLists.
 *
 * Then, a function of type createJumps is called which creates the individual list of
 * discontinuities for the considered insurance contract, and the pointer to this list is
 * saved in jumpLists->j1. The value of tr1 (==1: there are time-displaced transition
     payments;
 * ==0: there are not) which has been modified in the function of type createJumps is
     copied to
 * jumpLists->transExist. It will be needed in the discounting function for the time-
     displaced
 * transition payment.
 *
```

```
 * A second list jumpLists->j2 is created which only contains those entries of j1 which
     have
 * negative jump heights representing the contribution payments.
 *
 * A third list jumpLists->j3 only consists of one single entry: the contribution payment
     at time
 * insPeriod with height = sum of all contribution payments in j2.
 *
 * return value = pointer to a variable of type threeJumpLists_t.
 */
threeJumpLists_t *buildThreeJumpLists(threeJumpLists_t *jumpLists,
                jump_t *(*createJumps)(jump_t *, double, long, boolean_t *),
                double insPeriod, long nModel)
{
   boolean_t tr1 = 0; // for j1
   long i = 0;
   jump_t *ju = 0;
   boolean_t tr2 = 0; // for j2
   double sumOfPremiums = 0.0;
   double *height = 0;
   boolean_t tr3 = 0; // for j3


   jumpLists = allocateThreeJumpLists(jumpLists);

   // j1:
   jumpLists->j1 = createJumps(jumpLists->j1, insPeriod, nModel, &tr1);
   jumpLists->transExist = tr1;

   ju = jumpLists->j1;

   // j2:
   while (ju != 0)
   {
      for (i = 0; i <= nModel -1; i++)
      {
         if (ju->height[i] < 0)
         {
            jumpLists->j2 = allocJumpAndSetComponents(jumpLists->j2,
                                 ju->time, tr2, nModel, ju->height);
            break;
         }
          }
      ju = ju->next;
   }

   // j3:
   sumOfPremiums = calculateSumOfPremiumsAllStates(jumpLists->j2, nModel);
   height = allocateArrayOfDouble(nModel);
   height[0] = (-1.0) * sumOfPremiums;
   jumpLists->j3 = allocJumpAndSetComponents(jumpLists->j3, insPeriod, tr3, nModel, height
       );
   freeArrayOfDouble(height);
   jumpLists->whichList = 1; // j1 is used for the first calculation of the reserve.
```

```
   return jumpLists;
} // end buildThreeJumpLists


/* Function insertJumpIdenticalTimesPossible
 * This function inserts the entry at the right position into the jump list j (which is
     arranged
 * in descending order with respect to t) and returns the pointer to the first element of
 * the modified list.
 */
jump_t *insertJumpIdenticalTimesPossible(jump_t *j, jump_t *entry)
{
   jump_t *jSave = j;
   jump_t *prev  = 0;

   if (j == 0) // there is no entry in the list
      return entry;
   else
   {
      // Search the next smaller entry
      while (j->time > entry->time)
      {
         if (j->next != 0)
         {
            prev = j;
            j = j->next;
         }
         else // entry is inserted after the last entry of j
         {
            j->next = entry; // entry->next = 0 automatically
            return jSave;
         }
      }

      // Inserting entry before j:
      entry->next = j;
      if (prev != 0) // entry is not inserted before the first existing element of the
          list
      {
         prev->next = entry;
         return jSave;
      }
      else
         return entry;
   }

} // end insertJumpIdenticalTimesPossible


/* Function calculateSumOfPremiumsAllStates
 * This function calculates the sum of all discrete payments of negative height no matter
     for
 * which contract state they are defined.
```

```
 */
double calculateSumOfPremiumsAllStates(jump_t *j, long nModel)
{
   double sum = 0.0;
   long i = 0;

   while (j != 0)
   {
      for (i = 0; i <= nModel-1; i++)
      {
         if (j->height[i] < 0)
            sum += j->height[0];
      }
      j = j->next;
   }
   sum *= (-1.0); // the function shall return the positive value of the sum.

   return sum;
} // end calculateSumOfPremiumsAllStates


/* Function singlePremiumTimeZero
 *
 * By means of this function it can be determined whether there is a single premium
     payment
 * at time 0 or not. At time 0, the insurance contract is in initial state 0, so a single
 * premium payment at time 0 - if there is one - has to be defined for state 0!
 * Element after element of the jump, it is determined whether there is a negative jump
     height
 * in state 0 or not (the elements are arranged in descending order with respect to t).
 * If there is a negative jump height, the corresponding point of time t is compared to 0.
 *   t==0: single premium payment at time 0.
 *         the value of the single premium payment is returned.
 *   t!=0: there is at least one payment after time 0
 *         => there is no single premium payment at time 0
 *         => the value 1.0 is (non-negative value) is returned
 */
double singlePremiumTimeZero(jump_t *j)
{
   double eps = 1e-13;

   while (j != 0)
   {
      if (j->height[0] < 0)
      {
         if ((j->time < eps) && (j->time > (-1)*eps))
            return j->height[0]; // single premium payment at t=0; return value negative
         else
            return 1.0; // return value positive <=> no single premium payment at time 0
          }
      else
         j = j->next;
   }
   return 1.0; // There is no negative discrete payment.
```

```
} // end singlePremiumTimeZero


// Function freeJumpList
void freeJumpList(jump_t *j)
{
   jump_t *next;

   while (j != 0)
   {
      next = j->next;
      if (j->height)
      {
         free(j->height);
         j->height = 0;
      }
      free(j);

      j = next;
   }
} // end freeJumpList


// Function freeThreeJumpLists
void freeThreeJumpLists(threeJumpLists_t *jumpLists)
{
   if (jumpLists != 0)
   {
      freeJumpList(jumpLists->j1);
      freeJumpList(jumpLists->j2);
      freeJumpList(jumpLists->j3);
      free(jumpLists);
      jumpLists = 0;
   }
} // end freeThreeJumpLists
```

```
/* File jump.h
 * contains the definition of the data structure jump_s and its definition as jump_t as
     well as
 * the declaration of all functions concerning the discontinuities ("jumps") and the
 * discrete payments
 */

#ifndef JUMP_H
#define JUMP_H

#include "boolean.h"

typedef struct jump_s
```

```
{
   double time;
   double *height;
   boolean_t trans;      // trans == 1: time-displaced transition payment at this point
                         // of time => jump height == 0
   struct jump_s *next; // pointer to the list's next element
} jump_t;

typedef struct threeJumpLists_s
{
   jump_t *j1; // list of discontinuities (discrete payments and
   // other discontinuities) of the considered insurance contract
   jump_t *j2; // list of discontinuities which only contains those entries of j1 which
       have
   // negative height (=contributions). This list is required to calculate the expected
   // discounted contributions at time 0.
   jump_t *j3; // list of discontinuities which only contains one single entry: a
       contribution
   // payment at time insPeriod with height = sum of the contribution heights of j2.
   // Tis list is used to calculate premFactor if useChar==1.
   boolean_t transExist;
   // transExist == 1: there are entries with height==0 which correspond to
       discontinuities
   // caused by time-displaced transition payments.
   // transExist == 0: there are no time-displaced transition payments.
   long whichList; // indicates which of the jump lists j1, j2 and j3 has to be used.
} threeJumpLists_t;

jump_t *allocateJumpEntry(long nModel);

threeJumpLists_t *allocateTwoJumpLists(threeJumpLists_t *jumpLists);

jump_t *allocJumpAndSetComponents(jump_t *j, double t, boolean_t trans, long nModel,
    double *height);

jump_t *jumpsConstTermPremAnnually(jump_t *j, double insPeriod,long nModel, boolean_t *tr)
    ;
jump_t *jumpsConstTermAndEndowPremAnnually(jump_t *j, double insPeriod, long nModel,
    boolean_t *tr);
jump_t *jumpsAnnuityAnnuallyPremAnnually(jump_t *j, double insPeriod, long nModel,
    boolean_t *tr);
jump_t *jumps2Lives02a(jump_t *j, double insPeriod, long nModel, boolean_t *tr);
jump_t *jumps2Lives02b(jump_t *j, double insPeriod, long nModel, boolean_t *tr);

threeJumpLists_t *buildThreeJumpLists(threeJumpLists_t *jumpLists,
                  jump_t *(*createJumps)(jump_t *, double, long, boolean_t *),
                  double insPeriod, long nModel);

jump_t *insertJumpIdenticalTimesPossible(jump_t *j, jump_t *entry);

double calculateSumOfPremiumsAllStates(jump_t *j, long nModel);

double singlePremiumTimeZero(jump_t *j);
```

```
void freeJumpList(jump_t *j);

void freeThreeJumpLists(threeJumpLists_t *jumpLists);

#endif // JUMP_H
```

```
/* File more.c
 * contains the definition of further functions
 */

#include <stdlib.h>
#include <stdio.h>
#include "more.h"

// function freeThreePointers
void freeThreePointers(double *y1, double *y2, double *y3)
{
   if (y1)
   {
      free(y1);
      y1 = 0;
   }
   if (y2)
   {
      free(y2);
      y2 = 0;
   }
   if (y3)
   {
      free(y3);
      y3 = 0;
   }
} // end freeThreePointers


/* function allocateArrayOfLong:
 * allocates memory for an array of length n and data type long
 */
long *allocateArrayOfLong(long n)
{
   long *array = (long *)calloc(n, sizeof(long));
   if (array == 0)
   {
      fprintf(stderr, "Error (allocating memory for array of long)\n");
      exit(-1);
   }

   return array;
} // end allocateArrayOfLong
```

```
// function freeArrayOfLong:
void freeArrayOfLong(long *array)
{
   if (array)
   {
      free(array);
      array = 0;
   }
} // end freeArrayOfLong


/* function allocateArrayOfDouble:
 * allocates memory for an array of length n and data type double
 */
double *allocateArrayOfDouble(long n)
{
   double *array = (double *)calloc(n, sizeof(double));
   if (array == 0)
   {
      fprintf(stderr, "Error (allocating memory for array of double)\n");
      exit(-1);
   }

   return array;
} // end allocateArrayOfDouble


// function freeArrayOfDouble:
void freeArrayOfDouble(double *array)
{
   if (array)
   {
      free(array);
      array = 0;
   }
} // end freeArrayOfDouble
```

```
/* File more.h
 * contains the declaration of further functions
 */

void freeThreePointers(double *y1, double *y2, double *y3);

long *allocateArrayOfLong(long n);
void freeArrayOfLong(long *array);

double *allocateArrayOfDouble(long n);
```

```
void freeArrayOfDouble(double *array);
```

```
/* File parameters.c
 * contains the definition of all functions concerning the parameters
 */

#include <stdio.h>
#include <stdlib.h>
#include "ajk.h"
#include "more.h"
#include "parameters.h"
#include "vDelta.h"

// function allocateAllParameters
allParameters_t *allocateAllParameters()
{
   allParameters_t *allP = calloc(1, sizeof(allParameters_t));
   if (allP == 0)
   {
      fprintf(stderr, "Error (allocating memory for allP)!\n");
      exit(-1);
   }

   return allP;
}


/* Function setAllParameters
 * each component of a variable of data structure allParameters_t
 * is given a concrete value or pointer to a function
 */
boolean_t setAllParameters(
   allParameters_t *allP,
   long pers,
   long nPers,
   long nModel,
   long *x,
   double insPeriod,
   parametersInt_t **parInt,
   double (*calcInt)     (double, long, long, parametersInt_t **, long *,
                          boolean_t, const double *, double),
   double (*calcValA) (double, long, long),
   double (*calcValA1) (double, long, long),
   double (*calcValA2) (double, long, long),
   double upper,
   double premFactor,
   jump_t *(*createJumps) (jump_t *, double, long, boolean_t *),
   threeJumpLists_t *jumpLists,
   double delta,
```

```
    double (*calcDiscFunc)     (double, double, double),
    boolean_t useChar)
{
    allP->pers       = pers;
    allP->nPers      = nPers;
    allP->nModel     = nModel;
    allP->x          = x;
    allP->insPeriod = insPeriod;

    allP->parInt     = parInt;
    allP->calcInt    = calcInt;

    allP->calcValA = calcValA;
    allP->calcValA1 = calcValA1;
    allP->calcValA2 = calcValA2;
    allP->upper    = upper;

    allP->premFactor  = premFactor;
    allP->createJumps = createJumps;
    allP->jumpLists   = jumpLists;

    allP->delta           = delta;
    allP->calcDiscFunc    = calcDiscFunc;

    allP->useChar   = useChar;

    return true;
} // end setAllParameters


// Definition of the function setInsuranceModel01M:
boolean_t setInsuranceModel01M(allParameters_t *allP)
{
    /* TERM INSURANCE:
         * male insured person, x = 50
     * insurance period = 20 years
     * force of interest (delta, constant) = 0.03
     * death benefit = 100000 (constant), paid at the end of the year the insured person
         dies in
     * contributions: constant height, paid at the beginning of the year
     */

    // Defining and initialising the different variables:
    long pers        = 0;
    long nPers       = 0;
    long nModel      = 0;
    long *x          = 0;
    double insPeriod = 0.0;

    parametersInt_t **parInt = 0;
    double (*calcInt)(double, long, long, parametersInt_t **,
                      long *, boolean_t, const double *, double) = 0;

    double (*calcValA) (double, long, long) = 0;
```

```
double (*calcValA1)(double, long, long) = 0;
double (*calcValA2)(double, long, long) = 0;
double upper = 0.0;

double premFactor = 0.0;
jump_t *(*createJumps)(jump_t *, double, long, boolean_t *) = 0;
threeJumpLists_t *jumpLists = 0;

double delta = 0.0;
double (*calcDiscFunc)(double, double, double) = 0;

boolean_t useChar = 0;


double    *p = 0;
boolean_t ok = 0;


// Assignment:
pers      = 1;
nPers     = 2;
nModel    = 2;
insPeriod = 20.0;

x = allocateArrayOfLong(pers);
x[0] = 50;

p = allocateArrayOfDouble(nPers);
p[0] = 0.0;
p[1] = 1.0;

parInt = allocateParametersInt(pers, nPers);
ok = setParametersInt(parInt, 0, 0, nPers, 0.000134, 0.0000353, 1.1020, p);
if (ok != 1) // != 1: an error has occurred in setParametersInt
{
   fprintf(stderr, "Error(setParametersInt)! \n");
   return(-1);
}
calcInt = &calculateIntensityModel01M;

calcValA1 = &calculateConstValueA;
calcValA2 = &calculateConstValueAZero;
calcValA  = calcValA1;
// upper is not modified (upper == 0.0)

// premFactor is not modified (premFactor == 0.0)
createJumps = &jumpsConstTermPremAnnually;
jumpLists = buildThreeJumpLists(jumpLists, createJumps, insPeriod, nModel);

delta = 0.03;
calcDiscFunc = &calculateConstantDiscountingFunction;

useChar = 0;
```

```
    ok = setAllParameters(allP, pers, nPers, nModel, x, insPeriod, parInt, calcInt,
        calcValA,
                          calcValA1, calcValA2, upper, premFactor, createJumps, jumpLists,
                              delta,
                          calcDiscFunc, useChar);
    if (ok != 1) // != 1: an error has occurred in setAllParameters
    {
       fprintf(stderr, "Error(setAllParameters)! \n");
       return(-1);
    }

    freeArrayOfDouble(p);
    return true;
} // end setInsuranceModel01M


// Definition of the function setInsuranceModel02M:
boolean_t setInsuranceModel02M(allParameters_t *allP)
{
    /* TERM AND ENDOWMENT INSURANCE:
     * male insured person, x = 50
     * insurance period = 20 years
     * force of interest (delta, constant) = 0.03
     * death benefit = 100000 (constant),
     * paid at the end of the year the insured person dies in
     * endowment benefit: 300000 (paid at t=insPeriod)
     * contributions: constant height, paid at the beginning of the year
     */

    // Defining and initialising the different variables:
    long pers        = 0;
    long nPers       = 0;
    long nModel      = 0;
    long *x          = 0;
    double insPeriod = 0.0;

    parametersInt_t **parInt = 0;
    double (*calcInt)(double, long, long, parametersInt_t **,
                          long *, boolean_t, const double *, double) = 0;

    double (*calcValA) (double, long, long) = 0;
    double (*calcValA1) (double, long, long) = 0;
    double (*calcValA2) (double, long, long) = 0;
    double upper = 0.0;

    double premFactor = 0.0;
    jump_t *(*createJumps) (jump_t *, double, long, boolean_t *) = 0;
    threeJumpLists_t *jumpLists = 0;

    double delta = 0.0;
    double (*calcDiscFunc)(double, double, double) = 0;

    boolean_t useChar = 0;
```

```
    double *p    = 0;
    boolean_t ok = 0;

    // Assignment:
    pers      = 1;
    nPers     = 2;
    nModel    = 2;
    insPeriod = 20.0;

    x = allocateArrayOfLong(pers);
    x[0] = 50;

    p = allocateArrayOfDouble(nPers);
    p[0] = 0.0;
    p[1] = 1.0;

    parInt = allocateParametersInt(pers, nPers);
    ok = setParametersInt(parInt, 0, 0, nPers, 0.000134, 0.0000353, 1.1020, p);
    if (ok != 1) // != 1: an error has occurred in setParametersInt
    {
       fprintf(stderr, "Error(setParametersInt)! \n");
       return(-1);
    }
    calcInt = &calculateIntensityModel01M;

    calcValA1 = &calculateConstValueA;
    calcValA2 = &calculateConstValueAZero;
    calcValA  = calcValA1;
    // upper is not modified (upper == 0.0)

    // premFactor is not modified (premFactor == 0.0)
    createJumps = &jumpsConstTermAndEndowPremAnnually;
    jumpLists = buildThreeJumpLists(jumpLists, createJumps, insPeriod, nModel);

    delta = 0.03;
    calcDiscFunc = &calculateConstantDiscountingFunction;

    useChar = 0;

    ok = setAllParameters(allP, pers, nPers, nModel, x, insPeriod, parInt, calcInt,
        calcValA,
                          calcValA1, calcValA2, upper, premFactor, createJumps, jumpLists,
                              delta,
                          calcDiscFunc, useChar);
    if (ok != 1) // != 1: an error has occurred in setAllParameters
    {
       fprintf(stderr, "Error(setAllParameters)! \n");
       return(-1);
    }

    freeArrayOfDouble(p);
    return true;
} // end setInsuranceModel02M
```

```
// Definition of the function setInsuranceModel04M:
boolean_t setInsuranceModel04M(allParameters_t *allP)
{
   /* ANNUITY:
    * male insured person, x = 30
    * insurance period = 70 years
    * force of interest (delta, constant) = 0.03
    * contributions: constant height, paid at the beginning of the year k = 1, ..., 35
       * annuity benefits = 1000 (constant), paid at the beginning of the year k = 36,
          ..., 70
    */

   // Defining and initialising the different variables:
   long pers        = 0;
   long nPers       = 0;
   long nModel      = 0;
   long *x          = 0;
   double insPeriod = 0.0;

   parametersInt_t **parInt = 0;
   double (*calcInt)(double, long, long, parametersInt_t **,
                     long *, boolean_t, const double *, double) = 0;

   double (*calcValA)(double, long, long) = 0;
   double (*calcValA1) (double, long, long) = 0;
   double (*calcValA2) (double, long, long) = 0;
   double upper = 0.0;

   double premFactor = 0.0;
   jump_t *(*createJumps) (jump_t *, double, long, boolean_t *) = 0;
   threeJumpLists_t *jumpLists = 0;

   double delta = 0.0;
   double (*calcDiscFunc)    (double, double, double) = 0;

   boolean_t useChar = 0;


   double *p    = 0;
   boolean_t ok = 0;

   // Assignment:
   pers      = 1;
   nPers     = 2;
   nModel    = 2;
   insPeriod = 70.0;

   x = allocateArrayOfLong(pers);
   x[0] = 30;

   p = allocateArrayOfDouble(nPers);
   p[0] = 0.0;
```

```
    p[1] = 1.0;

    parInt = allocateParametersInt(pers, nPers);
    ok = setParametersInt(parInt, 0, 0, nPers, 0.000134, 0.0000353, 1.1020, p);
    if (ok != 1) // != 1: an error has occurred in setParametersInt
    {
       fprintf(stderr, "Error(setParametersInt)! \n");
       return(-1);
    }
    calcInt = &calculateIntensityModel01M;

    calcValA1 = &calculateConstValueAZero;
    calcValA2 = &calculateConstValueAZero;
    calcValA  = calcValA1;
    // upper is not modified (upper == 0.0)

    // premFactor is not modified (premFactor == 0.0)
    createJumps = &jumpsAnnuityAnnuallyPremAnnually;
    jumpLists = buildThreeJumpLists(jumpLists, createJumps, insPeriod, nModel);

    delta = 0.03;
    calcDiscFunc = &calculateConstantDiscountingFunction;

    useChar = 0;

    ok = setAllParameters(allP, pers, nPers, nModel, x, insPeriod, parInt, calcInt,
        calcValA,
                          calcValA1, calcValA2, upper, premFactor, createJumps, jumpLists,
                              delta,
                          calcDiscFunc, useChar);

    if (ok != 1) // != 1: an error has occurred in setAllParameters
    {
       fprintf(stderr, "Error(setAllParameters)! \n");
       return(-1);
    }

    freeArrayOfDouble(p);
    return true;
} // end setInsuranceModel04M


// Definition of the function setInsuranceModel2Lives02a:
boolean_t setInsuranceModel2Lives02a(allParameters_t *allP)
{
    /* LONG TERM CARE INSURANCE (SYMMETRIC):
     * 1st insured person (person no.0): male,   x[0] = 50
     * 2nd insured person (person no.1): female, x[1] = 50
     * 3 single states: active (*), in need of care (C), dead (+)
     * insurance period = 40 years
     * force of interest (delta, constant) = 0.03
     * 9 contract states: contract state 0 = (*,*),
        *                      contract state 1 = (*,P),
     *                      contract state 2 = (P,*),
```

```
     *                    contract state 3 = (P,P),
 *                     contract state 4 = (*,+),
 *                     contract state 5 = (+,*),
 *                     contract state 6 = (P,+),
 *                     contract state 7 = (+,P),
 *                     contract state 8 = (+,+)
 *
 * Payments for remaining in a contract state:
 * contract state 0:       contributions (constant, paid at the beginning of each year)
 * contract state 1, 2:    no contributions, no benefits
 * contract state 6, 7:    annuity (half)
 * contract state 3:       annuity (full)
 * contract state 4, 5, 8: no contributions, no benefits
 *
 * Payments for transitions:
 * contract state 0 -> 4,
 * contract state 1 -> 4,
 * contract state 0 -> 5,
 * contract state 2 -> 5:
 *                payment of 100000 (benefit for the first death)
 *                paid at the end of the year the transition occurs in
 * For all other transitions there are no benefits.
 * Assumption: independent mortality probabilities and intensities
 */

// Defining and initialising the different variables:
long pers        = 0;
long nPers       = 0;
long nModel      = 0;
long *x          = 0;
double insPeriod = 0.0;

parametersInt_t **parInt = 0;
double (*calcInt)(double, long, long, parametersInt_t **,
                  long *, boolean_t, const double *, double) = 0;

double (*calcValA) (double, long, long) = 0;
double (*calcValA1) (double, long, long) = 0;
double (*calcValA2) (double, long, long) = 0;
double upper = 0.0;

double premFactor = 0.0;
jump_t *(*createJumps) (jump_t *, double, long, boolean_t *) = 0;
threeJumpLists_t *jumpLists = 0;

double delta = 0.0;
double (*calcDiscFunc)(double, double, double) = 0;

boolean_t useChar = 0;


double *p1   = 0;
double *p2   = 0;
boolean_t ok = 0;
```

```
// Assignment:
pers      = 2;
nPers     = 3;
nModel    = 9;
insPeriod = 40.0;

x = allocateArrayOfLong(pers);
x[0] = 50;
x[1] = 50;

// FOR EACH OF THE TWO INSURED PERSONS
// p1 = weight[transition to a certain single state | given that
//             the single state (*) (= 0 in parInt) is left]
p1 = allocateArrayOfDouble(nPers); // can be used for both persons
p1[0] = 0.0; // == 0, because the single state (*) is left!
p1[1] = 1.0; // the same probability of dying and of becoming in need of care
p1[2] = 1.0;

// p2 = weight[transition to a certain single state | given that
//             the single state (P) (= 1 in parInt) is left]
p2 = allocateArrayOfDouble(nPers); // can be used for both persons
p2[0] = 0.0; // == 0, because (P) -> (*) is not possible!
p2[1] = 0.0; // == 0, because the single state (P) is left!
p2[2] = 2.0; // single state (P): probability of dying is twice as high as in single
    state (*)

parInt = allocateParametersInt(pers, nPers);
// person no.0, leaving the single state (*) (= 0 in parInt):
ok = setParametersInt(parInt, 0, 0, nPers, 0.000134, 0.0000353, 1.1020, p1);
if (ok != 1) // != 1: an error has occurred in setParametersInt
{
   fprintf(stderr, "Error(setParametersInt)! \n");
   return(-1);
}
// person no.0, leaving the single state (P) (= 1 in parInt):
ok = setParametersInt(parInt, 0, 1, nPers, 0.000134, 0.0000353, 1.1020, p2);
if (ok != 1)
{
   fprintf(stderr, "Error(setParametersInt)! \n");
   return(-1);
}
// person no.1, leaving the single state (*) (= 0 in parInt):
ok = setParametersInt(parInt, 1, 0, nPers, 0.000080, 0.0000163, 1.1074, p1);
if (ok != 1)
{
   fprintf(stderr, "Error(setParametersInt)! \n");
   return(-1);
}
// person no.1, leaving the single state (P) (= 1 in parInt):
ok = setParametersInt(parInt, 1, 1, nPers, 0.000080, 0.0000163, 1.1074, p2);
if (ok != 1)
{
   fprintf(stderr, "Error(setParametersInt)! \n");
```

```
      return(-1);
   }

   calcInt = &calculateIntensityModel2Lives02;

   calcValA1 = &calculateConstValueA2Lives02;
   calcValA2 = &calculateConstValueAZero;
   calcValA  = calcValA1;
   // upper is not modified (upper == 0.0)

   // premFactor is not modified (premFactor == 0.0)
   createJumps = &jumps2Lives02a;
   jumpLists = buildThreeJumpLists(jumpLists, createJumps, insPeriod, nModel);

   delta = 0.03;
   calcDiscFunc = &calculateConstantDiscountingFunction;

   useChar = 0;

   ok = setAllParameters(allP, pers, nPers, nModel, x, insPeriod, parInt, calcInt,
       calcValA,
                         calcValA1, calcValA2, upper, premFactor, createJumps, jumpLists,
                             delta,
                         calcDiscFunc, useChar);
   if (ok != 1) // != 1: an error has occurred in setAllParameters
   {
      fprintf(stderr, "Error(setAllParameters)! \n");
      return(-1);
   }

   freeArrayOfDouble(p1);
   freeArrayOfDouble(p2);
   return true;
} // end setInsuranceModel2Lives02a


// Definition of the function setInsuranceModel2Lives02b:
boolean_t setInsuranceModel2Lives02b(allParameters_t *allP)
{
   /* LONG TERM CARE INSURANCE (NON-SYMMETRIC):
    * 1st insured person (person no.0, maintaining position):
    *                     male,   x[0] = 50
    * 2nd insured person (person no.1): female, x[1] = 50
    * 3 single states: active (*), in need of care (C), dead (+)
    * insurance period = 40 years
    * force of interest (delta, constant) = 0.03
    * 9 contract states: contract state 0 = (*,*),
       *                     contract state 1 = (*,P),
    *                     contract state 2 = (P,*),
       *                     contract state 3 = (P,P),
    *                     contract state 4 = (*,+),
    *                     contract state 5 = (+,*),
    *                     contract state 6 = (P,+),
    *                     contract state 7 = (+,P),
```

```
 *                    contract state 8 = (+,+)
 *
 * Payments for remaining in a contract state:
 * contract state 0, 4:    contributions (constant, paid at the beginning of each year)
 * contract state 1, 6, 7: annuity (half)
 * contract state 3:       annuity (full)
 * contract state 2, 5, 8: no contributions, no benefits
 *
 * Payments for transitions:
 * contract state 0 -> 4,
 * contract state 1 -> 4,
 * contract state 0 -> 5,
 * contract state 2 -> 5:
 *                payment of 100000 (benefit for the first death)
 *                paid at the end of the year the transition occurs in
 * For all other transitions there are no benefits.
 * Assumption: independent mortality probabilities and intensities
 */

// Defining and initialising the different variables:
long pers        = 0;
long nPers       = 0;
long nModel      = 0;
long *x          = 0;
double insPeriod = 0.0;

parametersInt_t **parInt = 0;
double (*calcInt) (double, long, long, parametersInt_t **,
                   long *, boolean_t, const double *, double) = 0;

double (*calcValA) (double, long, long) = 0;
double (*calcValA1) (double, long, long) = 0;
double (*calcValA2) (double, long, long) = 0;
double upper = 0.0;

double premFactor = 0.0;
jump_t *(*createJumps) (jump_t *, double, long, boolean_t *) = 0;
threeJumpLists_t *jumpLists = 0;

double delta = 0.0;
double (*calcDiscFunc)(double, double, double) = 0;

boolean_t useChar = 0;


double *p1   = 0;
double *p2   = 0;
boolean_t ok = 0;

// Assignment:
pers      = 2;
nPers     = 3;
nModel    = 9;
insPeriod = 40.0;
```

```
x = allocateArrayOfLong(pers);
x[0] = 50;
x[1] = 50;

// FOR EACH OF THE TWO INSURED PERSONS
// p1 = weight[transition to a certain single state | given that
//             the single state (*) (= 0 in parInt) is left]
p1 = allocateArrayOfDouble(nPers); // can be used for both persons
p1[0] = 0.0; // == 0, because the single state (*) is left!
p1[1] = 1.0; // the same probability of dying
p1[2] = 1.0; //                        and of becoming in need of care

// p2 = weight[transition to a certain single state | given that
//             the single state (P) (= 1 in parInt) is left]
p2 = allocateArrayOfDouble(nPers); // can be used for both persons
p2[0] = 0.0; // == 0, because (P) -> (*) is not possible!
p2[1] = 0.0; // == 0, because the single state (P) is left!
p2[2] = 2.0; // single state (P): probability of dying is twice as high
             // as in single state (*)

parInt = allocateParametersInt(pers, nPers);
// person no.0, leaving the single state (*) (= 0 in parInt):
ok = setParametersInt(parInt, 0, 0, nPers, 0.000134, 0.0000353, 1.1020, p1);
if (ok != 1) // != 1: an error has occurred in setParametersInt
{
   fprintf(stderr, "Error(setParametersInt)! \n");
   return(-1);
}
// person no.0, leaving the single state (P) (= 1 in parInt):
ok = setParametersInt(parInt, 0, 1, nPers, 0.000134, 0.0000353, 1.1020, p2);
if (ok != 1)
{
   fprintf(stderr, "Error(setParametersInt)! \n");
   return(-1);
}
// person no.1, leaving the single state (*) (= 0 in parInt):
ok = setParametersInt(parInt, 1, 0, nPers, 0.000080, 0.0000163, 1.1074, p1);
if (ok != 1)
{
   fprintf(stderr, "Error(setParametersInt)! \n");
   return(-1);
}
// person no.1, leaving the single state (P) (= 1 in parInt):
ok = setParametersInt(parInt, 1, 1, nPers,
                      0.000080, 0.0000163, 1.1074, p2);
if (ok != 1)
{
   fprintf(stderr, "Error(setParametersInt)! \n");
   return(-1);
}

calcInt = &calculateIntensityModel2Lives02;
```

```
    calcValA1 = &calculateConstValueA2Lives02;
    calcValA2 = &calculateConstValueAZero;
    calcValA  = calcValA1;
    // upper is not modified (upper == 0.0)

    // premFactor is not modified (premFactor == 0.0)
    createJumps = &jumps2Lives02b;
    jumpLists = buildThreeJumpLists(jumpLists, createJumps, insPeriod, nModel);

    delta = 0.03;
    calcDiscFunc = &calculateConstantDiscountingFunction;

    useChar = 0;

    ok = setAllParameters(allP, pers, nPers, nModel, x, insPeriod, parInt, calcInt,
        calcValA,
                          calcValA1, calcValA2, upper, premFactor, createJumps, jumpLists,
                              delta,
                          calcDiscFunc, useChar);
    if (ok != 1) // != 1: an error has occurred in setAllParameters
    {
       fprintf(stderr, "Error(setAllParameters)! \n");
       return(-1);
    }

    freeArrayOfDouble(p1);
    freeArrayOfDouble(p2);
    return true;
} // end setInsuranceModel2Lives02b


// Function freeAllParameters
void freeAllParameters(allParameters_t *allP)
{
    if (allP != 0)
    {
       freeParametersInt(allP->pers, allP->nPers, allP->parInt);
       freeArrayOfLong(allP->x);
       freeThreeJumpLists(allP->jumpLists);
       free(allP);
       allP = 0;
    }
} // end freeAllParameters
```

```
/* File parameters.h
 * contains the definition of the data structure allParameters_s
 * and its definition as allParameters_t as well as
 * the declaration of all functions concerning the parameters.
 */
```

```
#ifndef PARAMETERS_H
#define PARAMETERS_H

#include "intensities.h"
#include "jump.h"
#include "boolean.h"

typedef struct allParameters_s
{
   // General parameters:
   long pers;         // number of insured persons in the contract
   long nPers;        // number of states per person (single states)
   long nModel;       // number of contract states
   long *x;           // initial ages of the insured persons
   double insPeriod; // insurance period

   /**********/

   // Intensities:
   parametersInt_t **parInt;
   double (*calcInt)    (double, long, long, parametersInt_t **, long *,
                         boolean_t, const double *, double);

   /**********/

   // Payments:
   double (*calcValA) (double, long, long);
   double (*calcValA1) (double, long, long);
   double (*calcValA2) (double, long, long);
   double upper; // actual upper bound for discounting the time-displaced transition
       payments

   // Jumps (Discontinuities):
   double premFactor;
   jump_t *(*createJumps) (jump_t *, double, long, boolean_t *);
   threeJumpLists_t *jumpLists;

   /**********/

   // Interest and discounting:
   double delta; // force of interest (constant)
   double (*calcDiscFunc)    (double, double, double);

   // Taking into account the character of the insurance contract:
   boolean_t useChar; // the character is taken into account (1)
                      // the character is not taken into account (0)
} allParameters_t;


// Declaration of the function allocateAllParameters:
allParameters_t *allocateAllParameters();
```

```
// Declaration of the function setAllParameters:
boolean_t setAllParameters(
   allParameters_t *allP,
   long pers,
   long nPers,
   long nModel,
   long *x,
   double insPeriod,
   parametersInt_t **parInt,
   double (*calcInt)    (double, long, long, parametersInt_t **, long *,
                         boolean_t, const double *, double),
   double (*calcValA) (double, long, long),
   double (*calcValA1) (double, long, long),
   double (*calcValA2) (double, long, long),
   double upper,
   double premFactor,
   jump_t *(*createJumps) (jump_t *, double, long, boolean_t *),
   threeJumpLists_t *jumpLists,
   double delta,
   double (*calcDiscFunc)    (double, double, double),
   boolean_t useChar);


// Declaration of the functions setInsuranceModel...:
boolean_t setInsuranceModel01M(allParameters_t *allP);
boolean_t setInsuranceModel02M(allParameters_t *allP);
boolean_t setInsuranceModel04M(allParameters_t *allP);

boolean_t setInsuranceModel2Lives02a(allParameters_t *allP);
boolean_t setInsuranceModel2Lives02b(allParameters_t *allP);

// Declaration of the functions freeAllParameters:
void freeAllParameters(allParameters_t *allP);

#endif // PARAMETERS_H
```

```
/* File payments.c
 * contains the definition of the function calculatePaymentValue
 */

#include <stdio.h>
#include "payments.h"

// function calculatePaymentValue
double calculatePaymentValue(double t, long j, long k, allParameters_t *allP)
{
   double disc = 0.0;
   double payment = 0.0;
```

```
    if (j == k) // disc=v(tUpper,tLower)=v(t,t)=1.0; no discounting!
        disc = allP->calcDiscFunc(t, t, allP->delta);
    else
        if (allP->jumpLists->transExist == 0)
            // there are no jumps of height 0 (= the time-displaced transition payment times)
            // <=> transition payments are effected immediately <=> no discounting!
            disc = allP->calcDiscFunc(t, t, allP->delta);
        else
            disc = allP->calcDiscFunc(t, allP->upper, allP->delta);

    payment = allP->calcValA(t, j, k);

    if (payment < 0) // contribution payment
        payment *= allP->premFactor;
    return disc * payment;
} // end calculatePaymentValue
```

```
/* File payments.h
 * contains the declaration of the function calculatePaymentValue
 */

#include "parameters.h"

double calculatePaymentValue(double t, long j, long k, allParameters_t *allP);
```

```
/* File regulaFalsi.c
 * contains the definition of the functions regulaFalsi and calcSecantAndCutWithXAxis
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_odeiv.h>
#include "regulaFalsi.h"
#include "calcReserve.h"
#include "more.h"

// function regulaFalsi
double *regulaFalsi(double premFactor1, double *y1, double premFactor2, double *y2, double
    *y3,
                    allParameters_t *allP, double tEnd, double *time, double *h,
                        gsl_odeiv_step *s,
                    gsl_odeiv_control *c, gsl_odeiv_evolve *e, gsl_odeiv_system *sys,
                        double eps_prem)
```

```
{
   while (1)
   {
      double premFactor3 = 0.0;
      long j = 0;

      premFactor3 = calcSecantAndCutWithXAxis(premFactor1, y1[0], premFactor2, y2[0]);
      printf("premFactor3 = %.10e \n", premFactor3);

      allP->premFactor = premFactor3; // update premFactor in allP

      for (j = 0; j <= allP->nModel - 1; j++)
         y3[j] = 0.0; // reset y3

      y3 = calculateReserve(y3, allP, tEnd, time, h, s, c, e, sys);
      for (j = 0; j <= allP->nModel - 1; j++)
         printf("y3[%ld] = %.3e  ", j, y3[j]);
      printf("\n");

      if (fabs(y3[0]) <= eps_prem)
      {
         free(y1);
         free(y2);
         return y3;
      }
      else
      {
         if (y3[0] > 0)
         {
            for (j = 0; j <= allP->nModel - 1; j++)
               y1[j] = y3[j];
            premFactor1 = premFactor3;
         }
         else
         {
            for (j = 0; j <= allP->nModel - 1; j++)
               y2[j] = y3[j];
            premFactor2 = premFactor3;
         }
      }
   }
} // end regulaFalsi


// function calcSecantAndCutWithXAxis
double calcSecantAndCutWithXAxis(double p1_x, double p1_y, double p2_x, double p2_y)
{
   double t = (-1) * p1_y / (p2_y - p1_y);
   return p1_x + t * (p2_x - p1_x);
} // end calcSecantAndCutWithXAxis
```

```c
/* File regulaFalsi.h
 * contains the definition of the functions regulaFalsi and calcSecantAndCutWithXAxis
 */

#include "parameters.h"

double *regulaFalsi(double premFactor1, double *y1, double premFactor2, double *y2, double
    *y3,
                    allParameters_t *allP, double tEnd, double *time, double *h,
                        gsl_odeiv_step *s,
                    gsl_odeiv_control *c, gsl_odeiv_evolve *e, gsl_odeiv_system *sys,
                        double eps_prem);

double calcSecantAndCutWithXAxis(double p1_x, double p1_y, double p2_x, double p2_y);
```

```c
/* File vDelta.c
 * contains the definition of the the discounting function (v(tLower, tUpper, delta))
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "vDelta.h"

// function calculateConstantDiscountingFunction
double calculateConstantDiscountingFunction(double tLower, double tUpper, double delta)
{
      return exp((-1)*delta*(tUpper - tLower));
} // end calculateConstantDiscountingFunction
```

```c
/* File vDelta.h
 * contains the declaration of the discounting function (v(tLower, tUpper, delta))
 */

double calculateConstantDiscountingFunction(double tLower, double tUpper, double delta);
```

List of Figures

Bibliography

[Auzinger, 2007] Auzinger, W. (2007). Einführung in die Numerik der Differentialgleichungen. Teil I: Gewöhnliche Differentialgleichungen (in English). Vorlesungsskriptum.

[Hartman, 1964] Hartman, P. (1964). *Ordinary differential equations.* New York-London-Sydney: John Wiley and Sons, Inc. XIV, 612 p. .

[Kloeden and Platen, 1992] Kloeden, P. E. and Platen, E. (1992). *Numerical solution of stochastic differential equations.*, Book chapter 8. Applications of Mathematics. 23. Berlin: Springer-Verlag. xxxv, 632 p. .

[Koller, 2000] Koller, M. (2000). *Stochastic models in life insurance. (Stochastische Modelle in der Lebensversicherung.).* Berlin: Springer. xii, 186 S.

[Milbrodt and Helbig, 1999] Milbrodt, H. and Helbig, M. (1999). *Mathematical methods for personal insurance. (Mathematische Methoden der Personenversicherung.).* Berlin: Walter de Gruyter. xi, 654 S.

[Møller and Steffensen, 2007] Møller, T. and Steffensen, M. (2007). *Market-Valuation Methods in Life and Pension Insurance.* Cambridge Univ Pr.

[Norberg, 2001] Norberg, R. (2001). Financial Mathematics in Life and Pension Insurance (Summer School in Mathematical Finance, Dubrovnik, 2001).

[Press et al., 2007] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical recipes. The art of scientific computing. 3rd ed.* Cambridge: Cambridge University Press. xxi, 1235 p.

[Stetter, 1985] Stetter, H. (1985). Vorlesungsskriptum aus "Numerische Mathematik" - Kapitel 7, Numerische Behandlung gewöhnlicher Differentialgleichungen, auf Basis des Buches "Numerische Mathematik 2" von Stoer, J. und Bulirsch, R.

[Wolthuis, 1994] Wolthuis, H. (1994). *Life insurance mathematics (The Markovian Model).* CAIRE Education Series 2. CAIRE: Brussels. x, 255 p. .