The optimality of a dividend barrier strategy for Lévy insurance risk processes, with a focus on the univariate Erlang mixture

by

Javid Ali

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Actuarial Science

Waterloo, Ontario, Canada, 2011

Abstract

In insurance risk theory, the surplus of an insurance company is modelled to monitor and quantify its risks. With the outgo of claims and inflow of premiums, the insurer needs to determine what financial portfolio ensures the soundness of the company's future while satisfying the shareholders' interests. It is usually assumed that the net profit condition (i.e. the expectation of the process is positive) is satisfied, which then implies that this process would drift towards infinity. To correct this unrealistic behaviour, the surplus process was modified to include the payout of dividends until the time of ruin.

Under this more realistic surplus process, a topic of growing interest is determining which dividend strategy is optimal, where optimality is in the sense of maximizing the expected present value of dividend payments. This problem dates back to the work of Bruno De Finetti (1957) where it was shown that if the surplus process is modelled as a random walk with ± 1 step sizes, the optimal dividend payment strategy is a barrier strategy. Such a strategy pays as dividends any excess of the surplus above some threshold. Since then, other examples where a barrier strategy is optimal include the Brownian motion model (Gerber and Shiu (2004)) and the compound Poisson process model with exponential claims (Gerber and Shiu (2006)).

In this thesis, we focus on the optimality of a barrier strategy in the more general Lévy risk models. The risk process will be formulated as a spectrally negative Lévy process, a continuous-time stochastic process with stationary increments which provides an extension of the classical Cramér-Lundberg model. This includes the Brownian and the compound Poisson risk processes as special cases. In this setting, results are expressed in terms of "scale functions", a family of functions known only through their Laplace transform. In Loeffen (2008), we can find a sufficient condition on the jump distribution of the process for a barrier strategy to be optimal. This condition was then improved upon by Loeffen and Renaud (2010) while considering a more general control problem.

The first chapter provides a brief review of theory of spectrally negative Lévy processes and scale functions. In chapter 2, we define the optimal dividends problem and provide existing results in the literature. When the surplus process is given by the Cramér-Lundberg process with a Brownian motion component, we provide a sufficient condition on the parameters of this process for the optimality of a dividend barrier strategy.

Chapter 3 focuses on the case when the claims distribution is given by a univariate mixture of Erlang distributions with a common scale parameter. Analytical results for the Value-at-Risk and Tail-Value-at-Risk, and the Euler risk contribution to the Conditional Tail Expectation are provided. Additionally, we give some results for the scale function and the optimal dividends problem. In the final chapter, we propose an expectation maximization (EM) algorithm similar to that in Lee and Lin (2009) for fitting the univariate distribution to data. This algorithm is implemented and numerical results on the goodness of fit to sample data and on the optimal dividends problem are presented.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Jean-François Renaud of the Université du Québec à Montréal for his time, for introducing me to this new literature and for guiding me throughout the course of my research. Additionally, I would like to thank David Landriault and Gordon Willmot of the University of Waterloo for reviewing my thesis and for providing insightful and informative suggestions. This research has been supported by the University of Waterloo, the Natural Sciences and Engineering Research Council of Canada and the Dominion of Canada General Insurance Company.

Table of Contents

List of Tables

List of Figures

Chapter 1

Lévy Insurance Risk Processes

1.1 Introduction

For an insurance company, it is important that a "sufficient" premium is charged in order to cover any losses and achieve a profit. Due to competition in the industry, the company cannot charge too high a premium. On the other hand, if the premium is too low, the company may not be able to cover any losses and will run the risk of bankruptcy[1]. The problem of quantifying this risk, amongst other problems, has been addressed in the field of ruin theory or collective risk theory.

In order to quantify the insurer's risks, it is necessary to model the insurer's surplus over time. While this surplus may be difficult to model in reality, the simplest model should take into account the premium income, claims outgo and initial capital of the company. We will now look at a classical collective risk model that was introduced by Cramér and Lundberg in 1930, and which serves as a foundation for more complex risk models.

[1]In this thesis, we will use the terms bankruptcy and ruin interchangeably, and define them to be the first time that the insurer's surplus becomes negative.

The Cramér-Lundberg Process

In this model, the surplus process $\{X_t\}_{t\geq 0}$ is of the form

$$X_t = x + ct - \sum_{i=1}^{N_t} C_i \,. \tag{1.1}$$

The probability law and expectation of X when $X_0 = x$ will be denoted respectively by $\mathbb{P}_x$ and $\mathbb{E}_x$, where we write $\mathbb{P}_0 = \mathbb{P}$ and $\mathbb{E}_0 = \mathbb{E}$.

The model assumes that the insurance company has an initial capital of $x \geq 0$ and that premiums are collected at a continuous constant rate of c over time. The last term represents the total claims payment by time t. It is given by a compound Poisson process where the number $\{N_t\}_{t\geq 0}$ of claims by time t is a Poisson process with rate λ. Additionally, the claim sizes $\{C_i\}_{i\geq 1}$ are a sequence of independent and identically distributed (i.i.d.) positive random variables independent of $\{N_t\}_{t\geq 0}$ and with probability distribution function f_C.

It is usually assumed that the net profit condition $c - \lambda\mathbb{E}(C) > 0$ is satisfied. Under this condition, there is a positive probability that the company will not become bankrupt in the future. A consequence of this condition is that the surplus process would drift towards infinity. If we denote the time to ruin as $\tau = \inf\{t > 0 | X_t < 0\}$, then Cramér showed that the probability of ruin

$$\mathbb{P}_x(\tau < \infty) = \mathbb{P}_x(X_t < 0 \text{ for some } t < \infty)$$

decays exponentially with respect to the initial capital, x. Figure 1.1 depicts a sample path of the Cramér-Lundberg process from $t = 0$ to $t = \tau$.

Lévy Insurance Risk Processes

In modern risk theory, a class of processes that have been recently considered for modelling the insurer's surplus is the class of spectrally negative Lévy processes, which are Lévy processes[2] with no positive jumps. It provides a generalization of the Cramér-Lundberg process and allows for a more realistic modelling of the surplus process. Let $X = \{X_t, t \geq 0\}$ be a spectrally negative Lévy process with Lévy triplet (μ, σ^2, v) where

[2]For readers unfamiliar with Lévy Processes, we refer you to [18] for a comprehensive look at the theory and applications of Lévy processes.

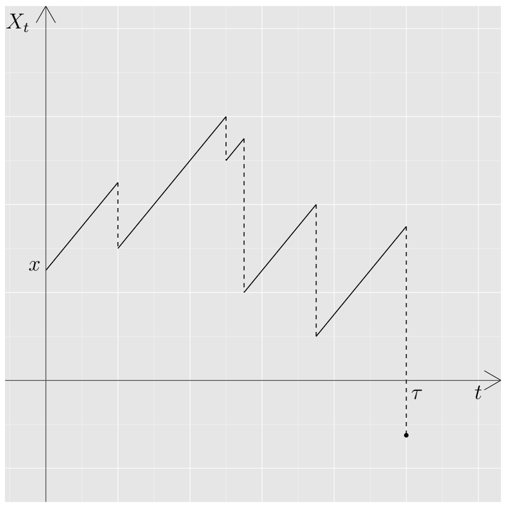


Figure 1.1: A sample path of the Cramér-Lundberg process

$\mu \in \mathbb{R}$, $\sigma \geq 0$ and v is a Lévy measure on $(0, \infty)$ satisfying $\int_{(0,\infty)}(1 \wedge x^2)v(\mathrm{d}x) < \infty$. In the literature, when we exclude from the definition the case of a negative subordinator or a deterministic drift, these processes are usually referred to as Lévy insurance risk processes.

For a spectrally negative Lévy process X, the Laplace exponent exists and is given by

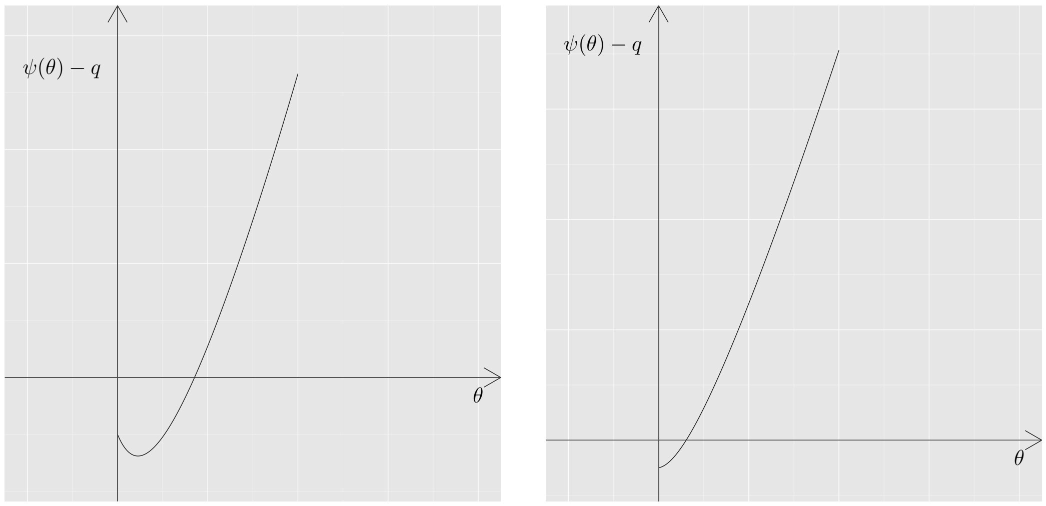
$$\psi(\theta) = \frac{1}{t}\log\left(\mathbb{E}\left(e^{\theta X_t}\right)\right) = \mu\theta + \frac{1}{2}\sigma^2\theta^2 - \int_0^\infty \left(1 - e^{-\theta x} - \theta x \mathbf{1}_{(x<1)}\right) v(\mathrm{d}x) \tag{1.2}$$

and is properly defined for $\theta \geq 0$ since the Lévy measure is concentrated on $(0, \infty)$.

We denote the right inverse of ψ as $\Phi(q) = \sup\{\theta \geq 0 : \psi(\theta) = q\}$. Some properties of the Laplace exponent are:

- $\psi(\theta)$ is strictly convex and tends to infinity as θ tends to infinity.
- On $[0, \infty)$, for $q \geq 0$, the equation $\psi(\theta) - q = 0$ has at most one real root given by $\Phi(q)$.

The shape of the Laplace exponent on $[0, \infty)$ depends on $\psi'(0) = \mathbb{E}(X)$. The two cases are shown in Figure 1.2 for the function $\psi(\theta) - q$, where $q \geq 0$.



(a) $\mathbb{E}(X_1) < 0$ (b) $\mathbb{E}(X_1) > 0$

Figure 1.2: $\psi(\theta) - q$

The Lévy-Itô Decomposition Theorem (see Theorem 3.1 of [18]) states that we can write $X = X^{(1)} + X^{(2)} + X^{(3)} + X^{(4)}$, where $X^{(1)}$ corresponds to a linear drift, $X^{(2)}$ is Brownian motion, $X^{(3)}$ is a compound Poisson process and $X^{(4)}$ is a square integrable martingale with a countably infinite number of jumps of size less than 1 over each finite time horizon. Denoting, for i = 1, 2, 3 and 4, the characteristic exponent of $X^{(i)}$ by $\psi^{(i)}(\theta)$, we have that

$$\psi^{(1)}(\theta) = \mu\theta \,, \tag{1.3}$$

$$\psi^{(2)}(\theta) = \frac{1}{2}\sigma^2\theta^2 \,, \tag{1.4}$$

$$\psi^{(3)}(\theta) = -\int_{x \geq 1} (1 - e^{-\theta x}) v(\mathrm{d}x) \,, \text{ and} \tag{1.5}$$

$$\psi^{(4)}(\theta) = \int_{x < 1} (\theta x - (1 - e^{-\theta x})) v(\mathrm{d}x). \tag{1.6}$$

For an insurance company, we can interpret X as a generalization of the classical surplus process. $X^{(1)}$ may represent the flow of premium income at a rate μ that more or less matches $X^{(3)}$, the outgo due to large, occasional claims. $X^{(4)}$ can also be seen as the outgo with respect to a countably infinite number of small claims that are offset by different drift rates. Finally, $X^{(2)}$ introduces extra variation in the system of premium inflow and claims outgo.

In this thesis, we will focus on a sub-class of these Lévy insurance risk processes. The risk process will be given by the Cramér-Lundberg process with a Brownian motion component. For convenience, we will refer to this sub-class of processes as jump-diffusion risk processes.

Jump-Diffusion Processes

For a jump-diffusion process, X is given by

$$X_t = x + \sigma B_t + ct - \sum_{i=1}^{N_t} C_i \tag{1.7}$$

where $x \geq 0$ is the initial capital, $\sigma \geq 0$, $\{B_t, t \geq 0\}$ is a standard Brownian motion process and $\{N_t\}_{t\geq 0}$ is a Poisson process with rate λ. The claim sizes $\{C_i\}_{i\geq 1}$ are a sequence of independent and identically distributed (i.i.d.) positive random variables independent of $\{N_t\}_{t\geq 0}$ and with probability distribution function f_C. Furthermore, all components in the above expression are independent of each other.

The corresponding Laplace exponent is

$$\psi(\theta) = c\theta + \frac{1}{2}\sigma^2\theta^2 - \lambda \int_0^\infty \left(1 - e^{-\theta x}\right) f_C(x)\mathrm{d}x. \tag{1.8}$$

The Laplace exponent above can be obtained from (1.2) by setting $v(\mathrm{d}x) = \lambda f_C(x)\mathrm{d}x$ and $c = \mu + \int_0^\infty x\mathbf{1}_{(x<1)}v(\mathrm{d}x)$.

Remark 1.1.1. *While we will assume $\sigma > 0$, it should be noted that we can recover the Cramér-Lundberg process by setting $\sigma = 0$ in* (1.8).

Compared to the Cramér-Lundberg process, this jump-diffusion process introduces extra volatility in the surplus process. Figure 1.3 depicts a sample path of the jump-diffusion process from $t = 0$ to $t = \tau$ (the time to ruin).

We will now discuss the Laplace transform and its properties, followed by a note on scale functions, which are both very useful tools in this literature.

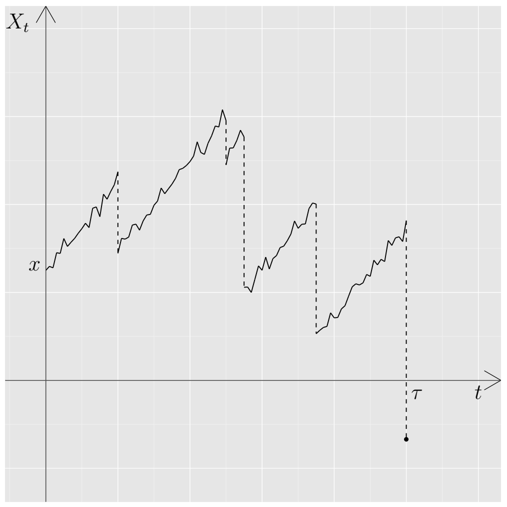


Figure 1.3: A sample path of the jump-diffusion process

1.2 Properties of the Laplace Transform

Definition 1. *The Laplace transform $L_Y(\theta)$ of a real-valued random variable Y with probability law v is defined as*

$$L_Y(\theta) = \mathbb{E}(e^{-\theta Y}) = \int_{\mathbb{R}} e^{-\theta y} v(\mathrm{d}y) \tag{1.9}$$

for some set of $\theta \in \mathbb{R}$ for which the integral converges.

Denoting the Laplace transform of the k^{th} derivative of the positive, real-valued function $f(x)$ by L_{f^k}, where we write $L_{f^0} = L_f$, we have the following useful properties from [1] and [8]:

- Laplace transform of the k^{th} derivative:

$$L_{f^k}(\theta) = \theta^k L_f(\theta) - \theta^{k-1} f(0) - \cdots - f^{(k-1)}(0). \tag{1.10}$$

- Given that $g(x)$ is another positive, real-valued function, the Laplace transform of the convolution of f and g, i.e. $(f * g)(x) = \int_0^\infty g(x-y)f(y)\,\mathrm{d}y$, is

$$L_{f*g}(\theta) = L_f(\theta)L_g(\theta). \tag{1.11}$$

- Initial Value Theorem:
$$f(0) = \lim_{\theta\to\infty} \theta L_f(\theta). \tag{1.12}$$

- Final Value Theorem:
  If the poles of $\theta L_f(\theta)$ lie either in the open left half plane or at the origin, with at most a single pole at the origin, then
$$\lim_{x\to\infty} f(x) = \lim_{\theta\to 0} \theta L_f(\theta). \tag{1.13}$$

- Cauchy Residue Theorem:
  Suppose C is a simple closed positively oriented contour. If $e^{\theta x} L_f(\theta)$ is analytic as a function of θ on and inside C except for a finite number of singular points $\theta_1, \theta_2, \ldots, \theta_n$ inside C, then the inverse Laplace transform of $L_f(\theta)$ is given by
$$L_f^{-1}(x) = \sum_{i=1}^{n} \text{Res}[e^{\theta x} L_f(\theta), \theta_i] \tag{1.14}$$
  where
$$\text{Res}[e^{\theta x} L_f(\theta), \theta_i] = \frac{1}{(k-1)!} \lim_{\theta\to\theta_i} \left[\frac{d^{k-1}}{d\theta^{k-1}} (\theta - \theta_i)^k e^{\theta x} L_f(\theta) \right]$$
  if L_f has a pole of order k at θ_i.

1.3 Scale Functions

Denote by $\{W^{(q)}; q \geq 0\}$ the q-scale functions of X; we assume *sufficient* smoothness for these functions when necessary. When $q = 0$, we write $W^{(0)} = W$. The q-scale function $W^{(q)}(x)$ is the unique, strictly increasing and continuous function on $[0, \infty)$ with Laplace transform given by

$$\int_0^\infty e^{-\theta x} W^{(q)}(x)\,\mathrm{d}x = \frac{1}{\psi(\theta) - q}\text{, for } \theta \geq \Phi(q), \tag{1.15}$$

and $W^{(q)}(x) = 0$ for all $x \leq 0$.

The scale function has been utilized widely in this literature ([7], [19], [23]) and two popular results (see [19]) from which this function derives its name are provided below.

Two sided exit problem. Let $a \in \mathbb{R}^+$, $q \geq 0$, $\tau_a^+ = \inf\{t > 0\colon X_t > a\}$ and $\tau_0^- = \inf\{t > 0\colon X_t < 0\}$. Then

$$\mathbb{E}_x(e^{-q\tau_a^+}\mathbf{1}_{(\tau_0^- > \tau_a^+)}) = \begin{cases} \frac{W^{(q)}(x)}{W^{(q)}(a)} & , 0 \leq x \leq a \\ 0 & , x < 0 \end{cases}. \tag{1.16}$$

One sided exit below. Define, for $q \geq 0$,

$$Z^{(q)}(x) = \begin{cases} 1 + q\int_0^x W^{(q)}(y)\,\mathrm{d}y & , 0 \leq x \leq a \\ 1 & , x < 0 \end{cases}. \tag{1.17}$$

Then, for any $x \in \mathbb{R}$,

$$\mathbb{E}_x(e^{-q\tau_0^-}\mathbf{1}_{(\tau_0^- < \infty)}) = Z^{(q)}(x) - \frac{q}{\Phi(q)}W^{(q)}(x). \tag{1.18}$$

When $q = 0$, this result gives an expression for the probability of ruin in terms of the 0-scale function. We have

$$\mathbb{P}(\tau_0^- < \infty) = \begin{cases} 1 - \psi'(0)W(x) & , \text{if } \psi'(0) > 0 \\ 1 & , \text{if } \psi'(0) \leq 0 \end{cases}. \tag{1.19}$$

Thus, ruin is certain when $\mathbb{E}(X_1) = \psi'(0) \leq 0$, i.e. when the process drifts downwards.

We can also define the scale function under the following change of measure.

Exponential Change of Measure. If we define a change of measure

$$\left.\frac{d\mathbb{P}^{\Phi(q)}}{d\mathbb{P}}\right|_{\mathcal{F}_t} = e^{\Phi(q)X_t - qt},$$

then the process X under this new measure $\mathbb{P}^{\Phi(q)}$ is still a spectrally negative Lévy process, but with a Lévy measure $e^{-\Phi(q)x}v(\,\mathrm{d}x)$. Additionally, the Laplace exponent is given by

$$\psi_{\Phi(q)}(\theta) = \psi(\theta + \Phi(q)) - q. \tag{1.20}$$

The scale function under $\mathbb{P}^{\Phi(q)}$ is also an increasing function and is given by

$$W_{\Phi(q)}(x) = e^{-\Phi(q)x}W^{(q)}(x) \tag{1.21}$$

with a Laplace transform $L_{W_{\Phi(q)}}(\theta)$ defined for all $\theta \geq 0$. Note that $W_{\Phi(q)}(x)$ is analogous to the 0-scale function under our original measure $\mathbb{P}$.

Useful Properties

We will now discuss some useful properties of the scale function for the special case when X is a jump-diffusion process:

- The initial value theorem for Laplace transforms (see (1.12)) allows us to determine the value of the scale function and its derivative at $x = 0$. In general, for a spectrally negative Lévy process with Laplace exponent given by (1.2), we have the following results which can also be found in [23]:

$$W^{(q)}(0) = \begin{cases} 0 & , \sigma > 0 \\ 1/c & , \sigma = 0 \end{cases} \tag{1.22}$$

$$W^{(q)\prime}(0) = \begin{cases} 2/\sigma^2 & , \sigma > 0 \\ \infty & , \sigma = 0 \end{cases} \tag{1.23}$$

- The function $\frac{W^{(q)}(x)}{W^{(q)\prime}(x)}$ is increasing to $\frac{1}{\Phi(q)}$ on $(0, \infty)$ (see [4]). This property also implies that $W^{(q)}(x)$ is log-concave on $(0, \infty)$, or equivalently, that $\log\left(W^{(q)}(x)\right)$ is concave on $(0, \infty)$.

- Since the poles of the Laplace transform of $W_{\Phi(q)}$ lie in the left half plane with a single pole $\theta = \Phi(q)$ at the origin, we can use the final value theorem for Laplace transforms (see (1.13)) to deduce the following properties of $W_{\Phi(q)}$ for large values of x:
  - $\lim_{x\to\infty} W_{\Phi(q)}(x) = \frac{1}{\psi'(\Phi(q))}$.
  - $\lim_{x\to\infty} W'_{\Phi(q)}(x) = 0$.
  - $\lim_{x\to\infty} \int_0^x W_{\Phi(q)}(x-y) f_C(y)\,\mathrm{d}y = \frac{1}{\psi'(\Phi(q))}$.

- The scale function $W^{(q)}(x)$ is asymptotically equivalent to

$$\frac{e^{\Phi(q)x}}{\psi'(\Phi(q))} \text{ as } x \to \infty. \tag{1.24}$$

This follows from the property above that $W_{\Phi(q)}(x)$ is bounded above by $\frac{1}{\psi'(\Phi(q))}$. Since $W_{\Phi(q)}(x)$ is more well behaved than $W^{(q)}(x)$ for large values of x, numerical analysis is usually performed on $W_{\Phi(q)}(x)$ and the results transformed back to the original measure by (1.21). A robust algorithm which uses this property for inverting the scale function's Laplace transform is provided in [28].

Scale Functions and Integro-Differential Equations

In this section, we link the scale function methodology to the classical methodology for the jump-diffusion process when $\sigma > 0$. We will show that we can find an integro-differential equation for the scale function for this model. In classical ruin theory, an integro-differential equation is usually solved to determine the probability of ruin; this equation is very similar to the one derived below. Specifically, solving the integro-differential equation below and using (1.19), we can obtain the ruin probability in this Lévy setting.

For this model, $\{X_t, t \geq 0\}$ and its Laplace exponent are respectively given by (1.7) and (1.8). By (1.15),

$$L_{W^{(q)}}(\theta) = \frac{1}{c\theta + \frac{1}{2}\sigma^2\theta^2 + \lambda(L_C(\theta) - 1) - q}. \tag{1.25}$$

Rearranging and using the property (1.10), we have

$$\begin{aligned}\frac{1}{2}\sigma^2 L_{W^{(q)\prime\prime}}(\theta) + cL_{W^{(q)\prime}}(\theta) - (\lambda + q)L_{W^{(q)}}(\theta) + \lambda L_{W^{(q)}}(\theta)L_C(\theta)& \\ = 1 - (c + \frac{1}{2}\sigma^2\theta)W^{(q)}(0) - \frac{1}{2}\sigma^2 W^{(q)\prime}(0).&\end{aligned}$$

Since $\sigma > 0$, recall that $W^{(q)}(0) = 0$ and $W^{(q)\prime}(0) = 2/\sigma^2$. This allows us to simplify the right hand side, and we obtain the following differential equation:

$$\frac{1}{2}\sigma^2 L_{W''}(\theta) + cL_{W'}(\theta) - (\lambda + q)L_W(\theta) + \lambda L_W(\theta)L_C(\theta) = 0.$$

Finally, after inverting the Laplace transforms, we have

$$\frac{1}{2}\sigma^2 W^{(q)\prime\prime}(x) + cW^{(q)\prime}(x) - (\lambda + q)W^{(q)}(x) + \lambda\int_0^x W^{(q)}(x - y)f_C(y)\,\mathrm{d}y = 0. \tag{1.26}$$

Remark 1.3.1. *The above integro-differential equation holds for $\sigma > 0$. For $\sigma = 0$, we can show, using similar steps, that the scale function is the solution to the following non-homogeneous integro-differential equation:*

$$cW^{(q)\prime}(x) - (\lambda + q)W^{(q)}(x) + \lambda \int_0^x W^{(q)}(x-y) f_C(y)\,\mathrm{d}y = \mathbf{1}_{(x=0)}. \tag{1.27}$$

Examples

Two examples have been provided to show that the spectrally negative Lévy process can be viewed as an extension of the classical Cramér-Lundberg process.

Brownian motion process with drift

The Brownian motion process with a drift coefficient is obtained by setting $\lambda = 0$ in (1.8), and therefore, X is given by

$$X_t = x + \sigma B_t + ct - \sum_{i=1}^{N_t} C_i\,,\ t \geq 0. \tag{1.28}$$

Given that

$$\psi(\theta) = \frac{1}{2}\sigma^2\theta^2 + c\theta, \tag{1.29}$$

the Laplace transform of the scale function is

$$L_{W^{(q)}}(\theta) = \frac{1}{\frac{1}{2}\sigma^2\theta^2 + c\theta - q}\,,\ \theta > \Phi(q) \tag{1.30}$$

which is equivalent to the differential equation

$$\frac{\sigma^2}{2} W^{(q)\prime\prime}(x) + cW^{(q)\prime}(x) - qW^{(q)}(x) = 0. \tag{1.31}$$

An explicit form of the scale function can be derived for this model as follows. We have

$$L_{W^{(q)}}(\theta) = \frac{1}{\frac{1}{2}\sigma^2\theta^2 + c\theta - q}$$

$$= \frac{1}{\sqrt{c^2+2q\sigma^2}} \left(\frac{1}{\theta - r} - \frac{1}{\theta - s} \right)$$

where

$$r = \frac{-c + \sqrt{c^2+2q\sigma^2}}{\sigma^2} \qquad \text{and} \qquad s = \frac{-c - \sqrt{c^2+2q\sigma^2}}{\sigma^2}.$$

Inverting the Laplace transform, the scale function is given by

$$W^{(q)}(x) = \frac{e^{rx} - e^{sx}}{\sqrt{c^2+2q\sigma^2}}\,,\ x > 0. \tag{1.32}$$

When $q = 0$, we have

$$W^{(q)}(x) = \frac{1}{c}(1 - e^{-2cx/\sigma^2})\,,\ x > 0. \tag{1.33}$$

Since $\psi'(0) = c > 0$, by (1.19), the probability of ruin is

$$\mathbb{P}(\tau_0^- < \infty) = e^{-2cx/\sigma^2} \tag{1.34}$$

which corresponds to the expression provided in [11].

Cramér-Lundberg process with exponential claims

Another common tractable model in classical ruin theory is the Cramér-Lundberg process with exponential claims. Assuming that the rate parameter of the exponential distribution is α, this process is obtained by setting $\sigma = 0$ and $f_C(x) = \alpha e^{-\alpha x}$ in (1.8). Given that

$$\psi(\theta) = c\theta + \lambda \left(\frac{\alpha}{\alpha + \theta} - 1 \right), \tag{1.35}$$

the Laplace transform of the scale function is

$$L_{W^{(q)}}(\theta) = \frac{1}{c\theta + \lambda\left(\frac{\alpha}{\alpha+\theta} - 1\right) - q}\,,\ \theta > \Phi(q). \tag{1.36}$$

An explicit form of the scale function can also be derived for this model as follows. We have

$$\begin{aligned} L_{W^{(q)}}(\theta) &= \frac{\alpha+\theta}{c\theta^2 + (c\alpha - \lambda - q)\theta - \alpha q} \\ &= \frac{1}{c\cdot(r-s)}\left(\frac{r+\alpha}{\theta - r} - \frac{\alpha+s}{\theta - s}\right) \end{aligned}$$

where we assume that r and s ($r > s$) are the distinct roots of the equation

$$\theta^2 + \left(\alpha - \frac{\lambda+q}{c}\right)\theta - \frac{\alpha q}{c} = 0.$$

Inverting the Laplace transform, the scale function is given by

$$W^{(q)}(x) = \frac{1}{c\cdot(r-s)}\left((r+\alpha)e^{rx} - (\alpha+s)e^{sx}\right)\ ,\ x > 0. \tag{1.37}$$

When $q = 0$, we have

$$r = 0 \qquad \text{and} \qquad s = -\frac{\alpha c - \lambda}{c},$$

and

$$W^{(q)}(x) = \frac{\alpha c - \lambda e^{-(\alpha c - \lambda)x/c}}{(\alpha c - \lambda)c}\ ,\ x > 0. \tag{1.38}$$

Supposing that $\psi^{'}(0) = c - \alpha/\lambda > 0$, by (1.19), the probability of ruin is

$$\mathbb{P}(\tau_0^- < \infty) = \frac{\lambda}{\alpha c}e^{-(\alpha c - \lambda)x/c} \tag{1.39}$$

which corresponds to the expression provided in [3].

We refer the reader to [16] for more results and tractable examples on scale functions for spectrally negative Lévy processes.

Chapter 2

The Optimality of a Barrier Strategy

2.1 Introduction

As mentioned in the previous chapter, under the assumption of the net profit condition, the surplus process drifts to infinity. This is not realistic in practice. While the solvency of a company is important, the payout of dividends is also of significance especially from the point of view of shareholders. In 1957, De Finetti [10] proposed that the surplus process be modified to include the payout of dividends to shareholders. Specifically, he aimed to maximize the expected present value of dividend payments until the time of ruin, and showed that a barrier strategy was the optimal strategy when the surplus process was modelled by a discrete random walk. Since then, considerable research has been done on this topic using the more general Lévy insurance risk models and under more realistic assumptions. While the optimal dividend strategy can be complex, a simple barrier strategy has been found to be the optimal strategy for a large sub-class of Lévy insurance risk surplus processes.

In this chapter, our objective is to provide sufficient conditions for the optimality of a dividend barrier strategy. We will first define the problem and then present and explain some existing results in this literature. Finally, when the surplus is modelled by a jump-diffusion process, we conclude with a sufficient condition on the parameters of the surplus process for the optimality of a barrier strategy.

2.2 Definition of the Problem

The notation in this chapter has been adapted from [23]. Let $X = \{X_t, t \geq 0\}$ be the Lévy insurance risk process introduced in Chapter 1. Denoting π as the dividend strategy, and D_t^π as the total amount of dividends paid up to time t, the net surplus process $\{U_t^\pi, t \geq 0\}$ is defined by $U_t^\pi = X_t - D_t^\pi$. Given the ruin time $\tau^\pi = \inf\{t > 0 : U_t^\pi < 0\}$ and appreciation rate $q \geq 0$, the main function of interest, called the value function, is

$$v_\pi(x) = \mathbb{E}_x \left[\int_0^{\tau^\pi} e^{-qt} \, \mathrm{d}D_t^\pi \right]. \tag{2.1}$$

We aim to find an optimal strategy π_* where

$$v_{\pi^*}(x) = \sup_{\pi \in \mathcal{C}} v_\pi(x). \tag{2.2}$$

and $\mathcal{C}$ is the set of admissible dividend strategies. Roughly speaking, a dividend strategy is admissible if at time t, the payment of dividends does not result in ruin under the modified/net surplus process.

We denote the barrier strategy at level a as π_a, and define the cumulative dividends paid until time t to be

$$D_t^a = (\sup_{0 \leq s \leq t} X_s - a)^+. \tag{2.3}$$

Also, if the barrier strategy at level a is applied, then we denote the value function by $v_a(x)$. If at any time the surplus is above a, such a strategy pays out immediately the excess of the surplus over a as dividends to shareholders. Figure 2.1 depicts the sample path of the original ($\{X_t\}_{t \geq 0}$) and modified ($\{U_t^\pi\}_{t \geq 0}$) surplus processes when a horizontal barrier strategy is applied at a.

Our objective of maximizing the present value of dividend payments is mainly from the view of the shareholders. Additionally, if too low a barrier level is applied, then ruin can occur early. On the other hand, if the barrier level is too high, then there will be too few payments of dividends to shareholders as the surplus process would take too long to hit the barrier.

It is also important to note that various dividend strategies[1] can be employed by an insurance company. However, we will only focus on the conditions for the optimality of

[1]For the interested reader, we refer you to [2] for a summary of some well known strategies in this literature.

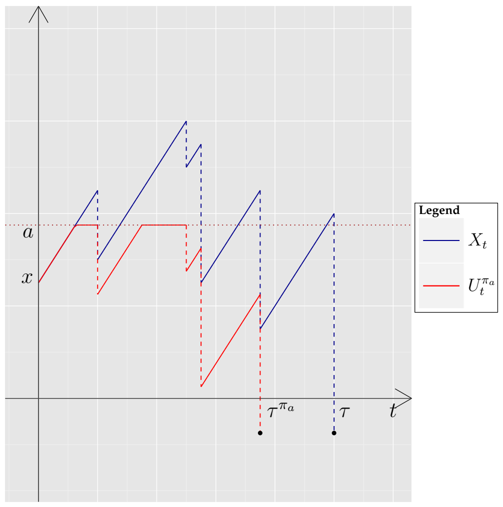


Figure 2.1: A sample path of the surplus process under a barrier strategy at a

a dividend barrier strategy. In [11] and [9], it was proven that a dividend barrier strategy is optimal for the Brownian motion process with drift and the Cramér-Lundberg process with exponential claims (the two examples provided at the end of the previous chapter).

2.3 Useful Results

In this section, we present a few interesting results from [5], [27], [17] and [23], some of which will be used in later sections.

Theorem 2.3.1 (Avram, Palmowski and Pistorius (2007), and Renaud and Zhou (2007))**.** *Assume $W^{(q)}$ is continuously differentiable on $(0,\infty)$. The value function of the barrier strategy at level $a \geq 0$ is given by*

$$v_a(x) = \begin{cases} \frac{W^{(q)}(x)}{W^{(q)\prime}(a)} & \textit{if } x \leq a \\ x - a + \frac{W^{(q)}(a)}{W^{(q)\prime}(a)} & \textit{if } x > a \end{cases}. \tag{2.4}$$

From the above theorem, if $x \leq a$, we see that an optimal barrier level is given by

$$a^* = \sup\{a \geq 0 : W^{(q)\prime}(a) \leq W^{(q)\prime}(x)\, \forall x \geq 0\}. \tag{2.5}$$

Additionally, assuming a barrier strategy is applied at level a, we can interpret the expression for the value function as follows: if the initial capital is not more than the barrier level (that is, $x \leq a$), then the present value of dividends is given by $\frac{W^{(q)}(x)}{W^{(q)\prime}(a)}$. Alternatively, if the initial capital is greater than the barrier level (that is, $x > a$), then the amount $x - a$ is paid out as dividends immediately at time 0 and the present value of the remaining dividends is given by $\frac{W^{(q)}(a)}{W^{(q)\prime}(a)}$ (which is consistent with the expression for the present value of dividends for a surplus process starting at $x = a$).

We will now proceed with the results for the optimality of a barrier strategy. The first result can be found in Loeffen (2008) and involves the scale function which was introduced earlier.

Theorem 2.3.2 (Loeffen (2008)). *Suppose that $W^{(q)}$ is sufficiently smooth and $W^{(q)}(a) \leq W^{(q)}(b)$ for all $a^* \leq a \leq b$. Then the barrier strategy at a^* is an optimal strategy.*

In words, it states that if the global minimum of the first derivative of the scale function is the last local minimum, then an optimal strategy is a barrier strategy. This forms the basis of all results in this chapter. In general, it is difficult to determine if this theorem is satisfied as scale functions for which explicit expressions exist are few. This leads to theorem 3 of Loeffen (2008) which provides us with a condition on the Lévy measure for the optimality of a barrier strategy.

Theorem 2.3.3 (Loeffen (2008)). *Suppose that the Lévy measure v of X has a completely monotone density, that is, $v(\mathrm{d}x) = \mu(x)\,\mathrm{d}x$, where $\mu : (0, \infty) \to (0, \infty)$ has derivatives $\mu^{(n)}$ of all orders which satisfy*

$$(-1)^n \mu^{(n)}(x) \geq 0 \text{ for } n = 0, 1, 2, \ldots. \tag{2.6}$$

Then $W^{(q)\prime}$ is strictly convex on $(0, \infty)$ for all $q > 0$. Consequently, Theorem 2.3.2 holds and the barrier strategy at a^* is an optimal strategy for the control problem.

In the case of the Cramér-Lundberg model with a Brownian component, there exists an optimal barrier strategy if the claims distribution is completely monotone. Examples of such distributions include the exponential, Weibull, Pareto and hyper-exponential distribution, or any distribution that is a mixture of these distributions. Throughout the past two years, researchers have succeeded in providing weaker conditions on the Lévy measure for the optimality of a barrier strategy.

Before we proceed to the next two theorems, note that a function $f : \mathbb{R} \to \mathbb{R}^+$ is log-convex if $\log f(x)$ is a convex function. Additionally, the tail of a Lévy measure is defined as a function $x \mapsto v(x, \infty)$ where $x \in (0, \infty)$.

Theorem 2.3.4 (Kyprianou, Rivero, Song (2008)). *Suppose that X has a Lévy density that is log-convex. Then the barrier strategy at a^* is optimal.*

This theorem is an improvement of the previous theorem as it only places a restriction on the second derivative. Therefore, we only need to calculate the second derivative (instead of all derivatives) of the log of the Lévy density to determine if the theorem is satisfied. Note that the examples of completely monotone distributions are also log-convex. The above result was pushed further by Loeffen and Renaud in 2010 in a more general setting by providing an even weaker condition on the Lévy measure.

Theorem 2.3.5 (Loeffen, Renaud (2010)). *Suppose the tail of the Lévy measure is log-convex. Then, for all $q \geq 0$, $W^{(q)}$ has a log-convex first derivative.*

Since the log-convexity of $W^{(q)'}$ implies the convexity of $W^{(q)'}$, theorem 2.3.2 is satisfied and the optimal strategy is a barrier strategy. It should also be noted that log-convexity of the Lévy measure implies the log-convexity of the tail of a Lévy measure. As such, the class of densities that satisfy this condition contains the class of log-convex densities. Note that for jump-diffusion processes, the requirement of log-convexity of the tail of the Lévy measure is equivalent to having log-convexity of the tail/survival probability of the claims distribution.

2.4 Conditions for the Optimality of a Barrier Strategy

Objectives

Theorems 2.3.3, 2.3.4 and 2.3.5 provide conditions on the Lévy measure for the optimality of a barrier strategy. While these conditions may apply to a wide class of distributions, there are still some important distributions in insurance theory such as the log-normal and Erlang distributions that do not satisfy such conditions.

For example, if we consider an Erlang(2, α) claims distribution, then the tail probability is

$$S_C(x) = 1 - F_C(x) = e^{-\alpha x} + \alpha x e^{-\alpha x} \tag{2.7}$$

and

$$\frac{\partial^2}{\partial x^2} S_C(x) = \alpha^2 e^{-\alpha x}(-1 + \alpha x). \tag{2.8}$$

Note that $\frac{\partial^2}{\partial x^2} S_C(x) < 0$ for $x < \frac{1}{\alpha}$ and therefore, the tail probability is not convex on $x \in (0, \frac{1}{\alpha})$. Since log-convexity implies convexity, this implies that the tail probability is not log-convex. Hence, we cannot conclude that a barrier strategy is optimal in the case of an Erlang(2, α) claims distribution.

Remark 2.4.1. *In [6], Azcue and Muller (2005) considered a jump-diffusion process with an Erlang(2, α) claims distribution. When $\sigma = 1.4$, $c = 21.4$, $\lambda = 10$, $\alpha = 1$ and $q = 0.1$, they found that an optimal strategy was given by a band strategy with the following value function:*

$$v_{\pi^*}(x) = \begin{cases} x + 2.119 & , x \in [0, 1.803) \\ 0.0944e^{-1.4882x} - 9.431e^{-0.0793x} + 11.257e^{0.03957x} & , x \in [1.803, 10.22) \\ x + 2.456 & , x \in [10.22, \infty) \end{cases} . \qquad (2.9)$$

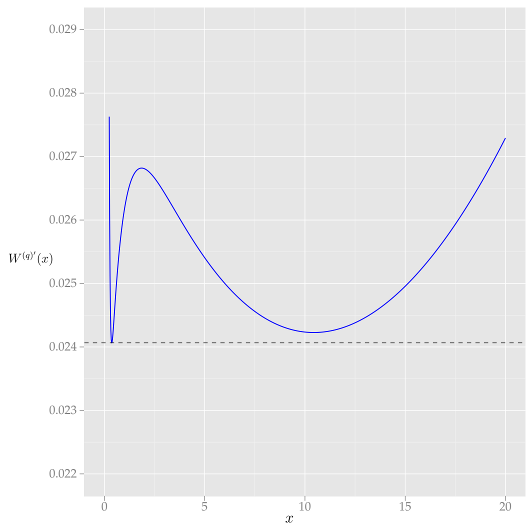
For the rest of this chapter, we provide conditions on the parameters of the surplus process for an optimal barrier strategy, and also look at the asymptotic behaviour of these parameters on the existence of a barrier strategy.

Motivation

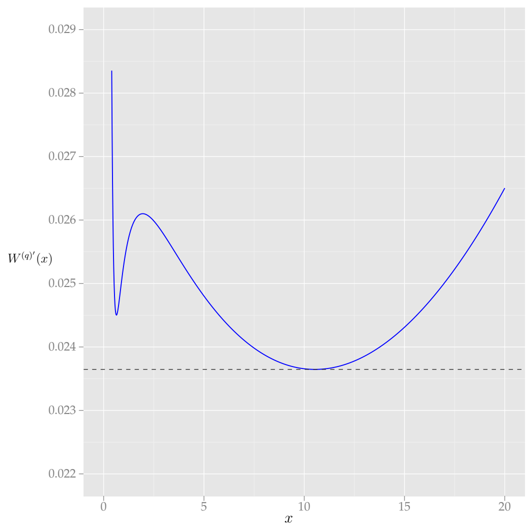
Our motivation for finding sufficient conditions on the parameters of the surplus process that result in an optimal barrier strategy stems from a numerical example provided in [23]. In this example, the surplus process is a jump-diffusion process with an Erlang(2, α) claims distribution. Fixing the parameters to $c = 21.4$, $\lambda = 10$, $\alpha = 1$ and $q = 0.1$, the derivative of the scale function was considered when $\sigma = 1.4$ and $\sigma = 2$. We reproduce the graphs for these two cases in Figure 2.2.

Note that for the first case, the band strategy given in remark 2.4.1 is an optimal strategy. However, in the latter case, by theorem 2.3.2, we see that a barrier strategy is optimal. The graphs also give us the impression that a barrier strategy at level a^* becomes optimal when σ increases. Alternatively, if we consider the case when $\sigma = 1.4$ and increase the value of α from 1 to 1.01, a barrier strategy is once again optimal as shown in Figure 2.3. Thus, there seems to be some relationship between the model parameters and the existence of an optimal barrier strategy. We now look at the Laplace exponent to obtain an explanation of this relationship.

For each of the scenarios described earlier, note that the process is drifting upwards (i.e. $\psi'(0^+) = \mathbb{E}(X_1) > 0$) so the Laplace exponent has a shape similar to that of Figure 2.2b. However, what is not apparent is that the value of $\Phi(q)$ had decreased in the

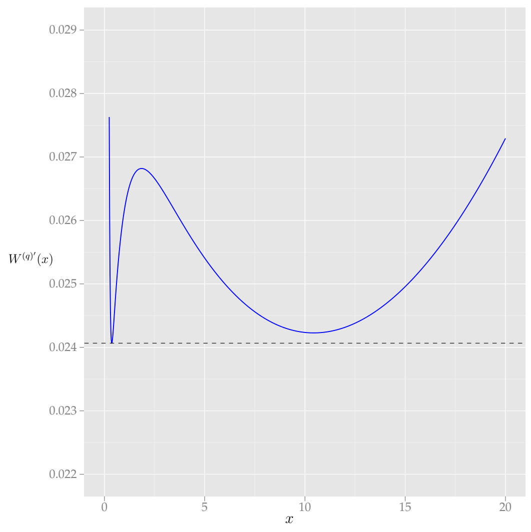


(a) $\sigma = 1.4$

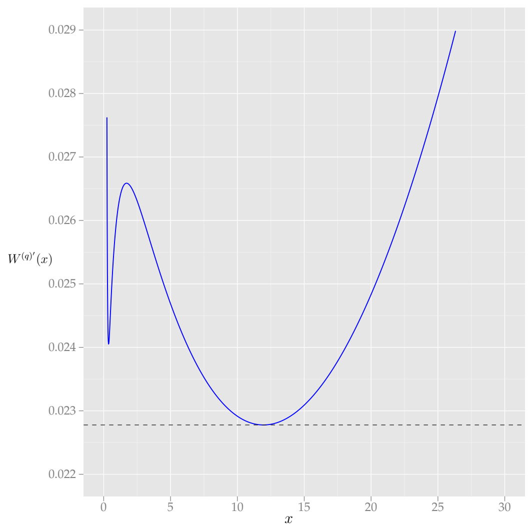


(b) $\sigma = 2.0$

Figure 2.2: The behaviour of $W^{(q)'}(x)$ as σ increases



(a) $\alpha = 1.00$



(b) $\alpha = 1.01$

Figure 2.3: The behaviour of $W^{(q)'}(x)$ as α increases

latter case of each scenario when a barrier strategy became optimal. Our explanation is as follows: since a barrier strategy is optimal for the Brownian motion process, a sufficiently large σ will introduce enough volatility in the jump-diffusion process such that a barrier strategy is eventually optimal.

Mathematically speaking, for this jump-diffusion process, we have

$$\psi(\theta) = c\theta + \frac{1}{2}\sigma^2\theta^2 + \lambda\left(\frac{\alpha^2}{(\alpha+\theta)^2} - 1\right). \tag{2.10}$$

and

$$\psi^{'}(\theta) = c + \sigma^2\theta - \frac{2\lambda\alpha^2}{(\alpha+\theta)^3}. \tag{2.11}$$

Since the Laplace exponent is a convex function on $(0, \infty)$, its derivative is an increasing function. Assuming $\psi^{'}(0^+) = (c - \frac{2\lambda}{\alpha}) > 0$, then (2.11) implies that the overall derivative increases if either σ or α increases. We can now see from Figure 1.2b that an overall increasing derivative results in a smaller value of $\Phi(q)$.

In addition, we have that

$$W^{(q)}(x) = \sum_{i=1}^{4} D_i e^{\theta_i x} \tag{2.12}$$

where $(\theta_i)_{i=1}^4$ are the roots of

$$(\psi(\theta) - q)(\alpha+\theta)^2 = 0 \tag{2.13}$$

and for $i = 1, \ldots, 4$,

$$D_i = \frac{1}{\psi^{'}(\theta_i)}. \tag{2.14}$$

Therefore,

$$W^{(q)'}(x) = \sum_{i=1}^{4} D_i \theta_i e^{\theta_i x}. \tag{2.15}$$

Remark 2.4.2. *Since one of the roots of* (2.13) *is* $\Phi(q)$, *the expressions* (2.12) *and* (2.15) *always contain the terms* $\frac{e^{\Phi(q)x}}{\psi^{'}(\Phi(q))}$ *and* $\frac{\Phi(q)e^{\Phi(q)x}}{\psi^{'}(\Phi(q))}$ *respectively.*

Hence, we can conclude that the largest real component of the roots $(\theta_i)_{i=1}^4$ is $\Phi(q)$ and as a result, the set $(Re(\theta_i))_{i=1}^4$ is bounded above by $\Phi(q)$. It will be shown in the next section (in a more general setting) that the real components of all other roots are negative. As such, we believe that a second consequence of a small $\Phi(q)$ is that, for small x, the term in $\Phi(q)$ would not have much effect on the derivative of the scale function and thus, the other terms (which are exponentially decreasing) would determine its behaviour. This implies that the scale function's derivative would be decreasing for earlier values of x and then be eventually increasing due to the term in $\Phi(q)$ having a stronger effect for large x. As such, $W^{(q)'}(x)$ would be convex and by Theorem 2.3.2, a barrier strategy would be an optimal dividend strategy.

2.5 A Sufficient Condition on the Parameters of the Surplus Process

Corollary 2.5.1. *When the surplus is modelled by a jump-diffusion process, a barrier strategy is an optimal strategy if* $\Phi(q) < \frac{q}{c}$.

The above corollary is a result of theorem 2.3.2. While the bound on $\Phi(q)$ can be seen numerically as too tight a bound and is more or less an asymptotic result, it provides useful insights on the existence of an optimal dividend barrier strategy as we will see in the next section. We will now look at the proof of this corollary.

Proof. For a jump-diffusion process, we obtained an alternative representation of the scale function in the previous chapter in the form of an integro-differential equation. By (1.26), we have

$$\frac{1}{2}\sigma^2 W^{(q)''}(x) + cW^{(q)'}(x) - (\lambda + q)W^{(q)}(x) + \lambda \int_0^x W^{(q)}(x-y) f_C(y)\,\mathrm{d}y = 0$$

where $f_C(y)$ is the p.d.f. of the claims distribution.

If a^* denotes the optimal barrier level, then theorem 2.3.2 states that a barrier strategy is optimal if $W^{(q)'}(b) \geq W^{(q)'}(a)$ for $b \geq a \geq a^*$. Therefore, if $W^{(q)''}(x) \geq 0$ on (a^*, ∞), then a barrier strategy is optimal. Using this condition and the above equation, if

$$cW^{(q)'}(x) - (\lambda + q)W^{(q)}(x) + \lambda \int_0^x W^{(q)}(x-y) f_C(y)\,\mathrm{d}y \leq 0$$

on (a^*, ∞), then a barrier strategy is optimal.

Since $W^{(q)'}(x) > 0$ (the scale function is a strictly increasing function), we can rewrite the above inequality as

$$W^{(q)'}(x)\left[c - (\lambda+q)\frac{W^{(q)}(x)}{W^{(q)'}(x)} + \lambda\int_0^x \frac{W^{(q)}(x-y)}{W^{(q)'}(x)} f_C(y)\,\mathrm{d}y\right] \leq 0$$

which implies that

$$c - (\lambda+q)\frac{W^{(q)}(x)}{W^{(q)'}(x)} + \lambda\int_0^x \frac{W^{(q)}(x-y)}{W^{(q)'}(x)} f_C(y)\,\mathrm{d}y \leq 0. \tag{2.16}$$

We look at the third term in the inequality (2.16). Again, since $W^{(q)}(x)$ is an increasing function,

$$\begin{aligned}
\lambda\int_0^x \frac{W^{(q)}(x-y)}{W^{(q)'}(x)} f_C(y)\,\mathrm{d}y &\leq \lambda\frac{W^{(q)}(x)}{W^{(q)'}(x)}\int_0^x f_C(y)\,\mathrm{d}y \\
&\leq \lambda\frac{W^{(q)}(x)}{W^{(q)'}(x)}\int_0^\infty f_C(y)\,\mathrm{d}y \\
&= \lambda\frac{W^{(q)}(x)}{W^{(q)'}(x)} \\
&\leq \frac{\lambda}{\Phi(q)}
\end{aligned}$$

where in the last line, we use the fact that $\frac{W^{(q)}(x)}{W^{(q)'}(x)}$ is bounded by $\frac{1}{\Phi(q)}$. Thus, this third term is an increasing function with a maximum value of $\frac{\lambda}{\Phi(q)}$.

Consider now the first two terms in the inequality (2.16). As mentioned in the last chapter (alternatively, see [4]), the function $\frac{W^{(q)}(x)}{W^{(q)'}(x)}$ is an increasing function on $(0, \infty)$. Therefore, for $x \in (a^*, \infty)$,

$$c - (\lambda+q)\frac{W^{(q)}(x)}{W^{(q)'}(x)}$$

is a decreasing function with a maximum of

$$c - (\lambda+q)\frac{W^{(q)}(a^*)}{W^{(q)'}(a^*)}$$

attained at a^*.

To ensure that (2.16) holds, we set the sum of the maximum of the first two terms and the maximum of the third term to be less than or equal to zero. Thus,

$$c - (\lambda + q)\frac{W^{(q)}(a^*)}{W^{(q)\prime}(a^*)} + \frac{\lambda}{\Phi(q)} \leq 0$$

or

$$\frac{W^{(q)}(a^*)}{W^{(q)\prime}(a^*)} \geq \frac{c + \frac{\lambda}{\Phi(q)}}{\lambda + q}.$$

Again, since $\frac{W^{(q)}(a^*)}{W^{(q)\prime}(a^*)} \leq \frac{1}{\Phi(q)}$, we have

$$\frac{1}{\Phi(q)} \geq \frac{c + \frac{\lambda}{\Phi(q)}}{\lambda + q}$$

which implies that

$$\frac{1}{\Phi(q)} > \frac{c}{q} \quad \text{or} \quad \Phi(q) < \frac{q}{c}.$$

□

2.6 Insights into the Sufficient Condition

In words, we can interpret the bound provided in the previous section as requiring $\Phi(q)$ to be sufficiently small for the existence of an optimal barrier strategy. In this section, we will look at the implications of this bound on the Brownian motion component and the claims distribution.

Recall that the function

$$\psi(\theta) - q = \frac{\sigma^2}{2}\theta^2 + c\theta + \lambda(L_C(\theta) - 1) - q$$

is increasing and convex on $(0, \infty)$.

Assuming $\mathbb{E}(X_1) = c - \lambda\mathbb{E}(C_1) > 0$, $\psi(\theta) - q$ exhibits the shape depicted in 1.2b, with $\Phi(q)$ given by the x-intercept. We now look at the first derivative of the function $\psi(\theta) - q$. We have

$$\begin{aligned}\frac{\partial}{\partial\theta}(\psi(\theta) - q) &= \sigma^2\theta + c + \lambda L_C^{'}(\theta) \\ &= (c - \lambda\mathbb{E}(C)) + \sigma^2\theta + \lambda(\mathbb{E}(C) + L_C^{'}(\theta)) \\ &> (c - \lambda\mathbb{E}(C)) + \sigma^2\theta \end{aligned} \tag{2.17}$$

since $\mathbb{E}(C) + L_C^{'}(\theta) > 0$.

If the overall derivative of the scale function is increased, then the positive root $\Phi(q)$ is smaller. Since the Laplace exponent is convex on $(0, \infty)$, this implies that the derivative is an increasing function. By (2.17), we see that we can decrease $\Phi(q)$ by either decreasing the expected value of the claims distribution, or more importantly, by increasing σ. Therefore, as σ is increased, eventually $\Phi(q) < \frac{q}{c}$ and a barrier strategy becomes optimal. A numerical example is provided in the last section of our final chapter.

Chapter 3

The Univariate Erlang Mixture

3.1 Introduction

In this chapter, we focus on the univariate Erlang mixture. This mixture of Erlang distributions possesses several appealing properties that makes it useful in modelling insurance losses. The univariate and multivariate mixtures can converge pointwise to any positive univariate and multivariate continuous distribution respectively. After providing definitions of these Erlang mixtures, we discuss and provide closed form expressions for some risk measures such as Value-at-Risk (VaR) and Conditional Tail Expectation (CTE), and for the Euler risk contribution to CTE.

Additionally, as we noted in the first chapter, tractable examples of scale functions are few and numerical algorithms are usually required to invert the Laplace transform. In the latter part of this chapter, we will give a closed-form expression for the scale function when the process is a jump-diffusion process with a univariate Erlang Mixture claims distribution, and also present analytical results with respect to the optimal dividends problem.

3.2 Definitions

In [30], it was proven that for any positive continuous distribution with density $f(x)$ and distribution function $F(x)$,

$$\hat{f}(x|\theta) = \sum_{i=1}^{\infty} \left[F(i\theta) - F((i-1)\theta)\right] \frac{x^{i-1}e^{-x/\theta}}{\theta^i (i-1)!}\,,\ x > 0 \tag{3.1}$$

the distribution function of (3.1) converges to $F(x)$ pointwise, as $\theta \to 0$. An alternative proof is given in [21] and is reproduced below.

Proof (from [21]). Let $\Phi(z) = \displaystyle\int_0^\infty e^{izx} f(x)dx$, where $i = \sqrt{-1}$, represent the characteristic function of $f(x)$, and $\phi_\theta(z)$ represent the characteristic function of (3.1). Then

$$\begin{aligned}
\phi_\theta(z) &= \sum_{i=1}^{\infty}[F(i\theta) - F((i-1)\theta)](1 - i\theta z)^{-1} \\
&= \sum_{i=1}^{\infty}\int_{(i-1)\theta}^{i\theta} (1 - i\theta z)^{-i} f(x)dx \\
&= \sum_{i=1}^{\infty}\int_{(i-1)\theta}^{i\theta} (1 - i\theta z)^{-\lceil x/\theta \rceil} f(x)dx \\
&= \int_0^{\infty} (1 - i\theta z)^{-\lceil x/\theta \rceil} f(x)dx.
\end{aligned}$$

Note that $(1 - i\theta z)^{-\lceil x/\theta \rceil}$ is bounded from above when $\theta|z| < 1$ and additionally, that $\lim_{\theta\to 0}(1 - i\theta z)^{-\lceil x/\theta \rceil} = e^{izx}$. Therefore, by the Dominated Convergence Theorem, $\lim_{\theta\to 0} \phi_\theta(z) = \Phi(z)$ for all z. Furthermore, by the Lévy Continuity Theorem, the distribution function of (3.1) converges to that of $f(x)$ pointwise. □

3.2.1 Univariate Erlang Mixture

The univariate Erlang mixture is defined as a mixture of Erlang distributions with a common scale parameter. The density function is given by

$$f(x|\theta,\vec{\alpha}) = \sum_{i=1}^{M} \alpha_i \frac{x^{r_i-1}e^{-x/\theta}}{\theta^{r_i}(r_i-1)!}\,,\; x>0 \tag{3.2}$$

where $M \in \mathbb{Z}^+$ and $r_1 < r_2 < \cdots < r_M$ are positive integers. Additionally, $\vec{\alpha} = \{\alpha_i\}_{i=0}^{M}$ and $\alpha_i \geq 0$ with $\sum_{i=1}^{\infty} \alpha_i = 1$. The parameter α_i can be interpreted as the weight corresponding to an Erlang distribution with shape parameter r_i and scale parameter θ.

We can see that (3.2) bears a similar form to (3.1). Therefore, this finite mixture allows us to approximate any positive continuous distribution to some specified accuracy.

3.2.2 Multivariate Erlang Mixture

The multivariate counterpart to (3.2) is

$$f(\vec{x}|\theta,\vec{\alpha}) = \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}} \prod_{i=1}^{k} \frac{x_i^{m_i-1}e^{-x_i/\theta}}{\theta^{m_i}(m_i-1)!}\,,\; \vec{x}>0 \tag{3.3}$$

where $\vec{x} = (x_1, \ldots, x_k)$, and $\vec{m} = (m_1, \ldots, m_k) \in \mathbb{Z}^{+k}$. Moreover, the weights are given by $\vec{\alpha} = (\alpha_{\vec{m}} | \alpha_{\vec{m}} \geq 0; m_i = 1, 2, \ldots; i = 1, 2, \ldots, k)$ and satisfy $\sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}} = 1$.

This distribution can also approximate any positive continuous multivariate distribution to some specified accuracy. In the next section, we will present some analytical results based on both the multivariate and univariate Erlang mixtures.

3.3 Risk Measures

Let X be a random variable with a univariate Erlang mixture distribution. For simplicity, we assume that the probability density function of X is given by

$$f_X(x) = \sum_{i=1}^{M^*} \alpha_i \frac{x^{i-1} e^{-x/\theta}}{\theta^i (i-1)!}\,,\ x > 0. \tag{3.4}$$

Note that (3.4) differs from (3.2) in the shape parameters. In (3.4), the shape parameters are $(1, \ldots, M^*)$, whereas in (3.2), they are given by $(r_1, \ldots, r_M)$. If we assume that $r_M \leq M^*$, to retrieve (3.2) from (3.4), we simply set $\alpha_i = 0$ for $i \notin (r_1, \ldots, r_M)$. Some of the results in this section can be found in [21].

3.3.1 Value-at-Risk (VaR)

The Value-at-Risk (VaR) of a univariate Erlang mixture at some confidence level p, V_p is the solution to the equation

$$e^{-V_p/\theta} \sum_{i=0}^{M^*-1} \sum_{j=i+1}^{M^*} \alpha_j \frac{V_p^i}{\theta^i i!} = 1 - p. \tag{3.5}$$

Proof. To obtain the expression for VaR, we use the result that

$$\int_x^\infty \frac{y^{k-1} e^{-x/\theta}}{\theta^k (k-1)!}\,\mathrm{d}y = \sum_{j=0}^{k-1} \frac{e^{-x/\theta} x^j}{\theta^j j!}.$$

Note that the survival distribution of X can now be written as

$$\begin{aligned} S_X(x) = \int_x^\infty f_X(x)\,\mathrm{d}x &= \int_x^\infty \sum_{i=1}^{M^*} \alpha_i \frac{x^{i-1} e^{-x/\theta}}{\theta^i (i-1)!}\,\mathrm{d}x \\ &= \sum_{i=1}^{M^*} \alpha_i \int_x^\infty \frac{x^{i-1} e^{-x/\theta}}{\theta^i (i-1)!}\,\mathrm{d}x \end{aligned}$$

$$= \sum_{i=1}^{M^*} \alpha_i \sum_{j=0}^{i-1} \frac{e^{-x/\theta} x^j}{\theta^j j!} \tag{3.6}$$

Therefore, V_p is such that

$$\begin{aligned}
1 - p = S_X(V_p) &= \sum_{i=1}^{M^*} \alpha_i \sum_{j=0}^{i-1} \frac{e^{-x/\theta} x^j}{\theta^j j!} \\
&= \sum_{j=0}^{M^*-1} \sum_{i=j+1}^{M^*} \alpha_i \frac{e^{-x/\theta} x^j}{\theta^j j!} \\
&= \sum_{i=0}^{M^*-1} \sum_{j=i+1}^{M^*} \alpha_j \frac{e^{-x/\theta} x^i}{\theta^i i!}
\end{aligned}$$

where we rename the indices of summation in the last line. □

Remark 3.3.1. *The expression in [21] should be corrected for the extra term in the most outer sum since the index in their first summation ends at $i = M^*$ instead of $i = M^* - 1$.*

3.3.2 Conditional Tail Expectation (CTE)

The conditional tail expectation (CTE) at a confidence level p is given by

$$CTE(100p) = \frac{\theta e^{-V_p/\theta}}{1-p} \sum_{i=0}^{M^*-1} \sum_{j=i}^{M^*-1} \sum_{k=j+1}^{M^*} \alpha_k \frac{V_p^i}{\theta^i i!} + V_p. \tag{3.7}$$

Proof. Using the expression (3.6) for $S_X(x)$, we have

$$\begin{aligned}
CTE(100p) &= \mathbb{E}(X|X > V_p) \\
&= \mathbb{E}(X - V_p|X > V_p) + V_p \\
&= \frac{1}{1-p} \int_{V_p}^{\infty} S_X(x)\, \mathrm{d}x + V_p
\end{aligned}$$

$$= \frac{1}{1-p}\sum_{i=1}^{M^*}\alpha_i\sum_{j=0}^{i-1}\int_{V_p}^{\infty}\frac{e^{-x/\theta}x^j}{\theta^j j!}\,\mathrm{d}x + V_p$$

$$= \frac{\theta}{1-p}\sum_{i=1}^{M^*}\alpha_i\sum_{j=0}^{i-1}\int_{V_p}^{\infty}\frac{e^{-x/\theta}x^j}{\theta^{j+1} j!}\,\mathrm{d}x + V_p$$

$$= \frac{\theta}{1-p}\sum_{i=1}^{M^*}\alpha_i\sum_{j=0}^{i-1}\int_{V_p}^{\infty}\frac{e^{-x/\theta}x^j}{\theta^{j+1} j!}\,\mathrm{d}x + V_p$$

$$= \frac{\theta}{1-p}\sum_{i=1}^{M^*}\alpha_i\sum_{j=0}^{i-1}\sum_{k=0}^{j}\frac{e^{-V_p/\theta}{V_p}^k}{\theta^k k!} + V_p$$

$$= \frac{\theta e^{-V_p/\theta}}{1-p}\sum_{i=1}^{M^*}\sum_{j=0}^{i-1}\sum_{k=0}^{j}\alpha_i\frac{{V_p}^k}{\theta^k k!} + V_p$$

$$= \frac{\theta e^{-V_p/\theta}}{1-p}\sum_{j=0}^{M^*-1}\sum_{k=0}^{j}\sum_{i=j+1}^{M^*}\alpha_i\frac{{V_p}^k}{\theta^k k!} + V_p$$

$$= \frac{\theta e^{-V_p/\theta}}{1-p}\sum_{k=0}^{M^*-1}\sum_{j=k}^{M^*-1}\sum_{i=j+1}^{M^*}\alpha_i\frac{{V_p}^k}{\theta^k k!} + V_p$$

where we once again rename the indices of summation in the last line. □

Remark 3.3.2. *The formula in [21] should be corrected for the extra term in the most outer sum since the index in their first summation ends at $i = M^*$ instead of $i = M^* - 1$.*

Remark 3.3.3. *Note that we can derive an alternative form for the CTE of X at a confidence level p by using $\mathbb{E}(X|X > V_p) = \frac{1}{1-p}\left\{\mathbb{E}(X) - \mathbb{E}\left(X \cdot I_{\{X<V_p\}}\right)\right\}$. It is given by*

$$CTE(100p) = \frac{\theta^2}{1-p}\sum_{n=0}^{\infty}\sum_{i=n}^{\infty}\alpha_i \cdot i \cdot p(v, n+1, \theta). \tag{3.8}$$

3.4 Euler Risk Contributions

3.4.1 The Aggregate Loss

Given a multivariate Erlang mixture (3.3), the aggregate loss $S = X_1 + X_2 + \cdots + X_k$ has a univariate Erlang mixture with weights

$$\alpha_i^S = \sum_{m_1+\cdots+m_k=i} \alpha_{\vec{m}}.$$

Proof. The moment generating function (m.g.f.) of an Erlang random variable with shape parameter m_j and scale parameter θ is given by

$$(1-\theta t_j)^{-m_j}.$$

Therefore, the m.g.f. of $\vec{X} = (X_1, \ldots X_k)$ is

$$M_{\vec{X}}(t_1, \ldots, t_k) = \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}} \prod_{i=1}^{k} (1-\theta t_j)^{-m_j}$$

and the m.g.f. of S is

$$\begin{aligned} M_S(t) = M_{\vec{X}}(t, \ldots, t) &= \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}} \prod_{i=1}^{k} (1-\theta t)^{-m_j} \\ &= \sum_{i=1}^{\infty} \sum_{m_1+\cdots+m_k=i} \alpha_{\vec{m}} (1-\theta t)^{-i} \\ &= \sum_{i=1}^{\infty} \alpha_i^S (1-\theta t)^{-i}. \end{aligned}$$

Inverting the m.g.f, we have

$$f_S(s) = \sum_{i=1}^{\infty} \alpha_i^S p(s, i, \theta)$$

where

$$p(s, i, \theta) = \frac{s^{i-1} e^{-s/\theta}}{\theta^i (i-1)!} \, , \; s > 0. \tag{3.9}$$

$\square$

3.4.2 The Euler Risk Contribution to CTE

Given that $VaR_p(S) = v$, the Euler risk contribution of the j-th loss $(1 \leq j \leq k)$ to CTE at some specified confidence level p is given by

$$\mathbb{E}(X_j | S > V) = \frac{\theta^2}{1-p} \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}} m_j \sum_{n=0}^{m_1 + \ldots m_k} \frac{e^{-v/\theta} v^n}{\theta^{n+1} n!}.$$

Proof. We calculate the risk contribution of X_j to the CTE. The proof of the risk contribution is similar to that in [13]. Since $S = X_1 + X_2 + \cdots + X_k$, the m.g.f. of the bivariate distribution of $(S - X_j, X_j)$ is given by

$$\begin{aligned} &\sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}} (1 - \theta t_j)^{-m_j} (1 - \theta t)^{-(m_1 + \cdots + m_k - m_j)} \\ &= \sum_{i=1}^{\infty} \sum_{m_j=1}^{\infty} \sum_{m_1 + \cdots + m_k - m_j = i} \alpha_{\vec{m}} (1 - \theta t_j)^{-m_j} (1 - \theta t)^{-i}. \end{aligned}$$

Using expression (3.9), we have

$$f_{(S - X_j, X_j)}(\tau, x) = \sum_{i=1}^{\infty} \sum_{m_j=1}^{\infty} \sum_{m_1 + \cdots + m_k - m_j = i} \alpha_{\vec{m}} \, p(x, m_j, \theta) \, p(\tau, i, \theta).$$

We want to find $\mathbb{E}(X_j | S > v) = \frac{1}{1-p} \{E(X_j) - \mathbb{E}(X_j I_{\{S < v\}})\}$.

$$\begin{aligned}
\mathbb{E}(X_j I_{\{S<v\}}) &= \int_0^\infty \int_0^v x\, f_{(S-X_j,X_j)}(s-x,x)\,\mathrm{d}s\,\mathrm{d}x \\
&= \sum_{i=1}^\infty \sum_{m_k=1}^\infty \sum_{m_1+\cdots+m_k-m_j=i} \alpha_{\vec{m}} \int_0^\infty \int_0^v x\, p(s-x,i,\theta)\, p(x,m_j,\theta)\,\mathrm{d}s\,\mathrm{d}x.
\end{aligned}$$

Consider the multiple integral inside the sum. We have

$$\begin{aligned}
&\int_0^\infty \int_0^v x\, p(s-x,i,\theta)\, p(x,m_j,\theta)\,\mathrm{d}s\,\mathrm{d}x \\
&\quad= \int_0^v \int_0^v x\, p(s-x,i,\theta)\, p(x,m_j,\theta)\,\mathrm{d}s\,\mathrm{d}x
\end{aligned}$$

and with a change of variables $x = u$ and $s = u + w$, it can be written as

$$\begin{aligned}
&\int_0^v \int_0^{v-u} u\, p(w,i,\theta)\, p(u,m_j,\theta)\,\mathrm{d}w\,\mathrm{d}u \\
&= m_j\,\theta \int_0^v \int_0^{v-u} p(w,i,\theta)\, p(u,m_j+1,\theta)\,\mathrm{d}w\,\mathrm{d}u \\
&= m_j\,\theta\, F_{Y^*}(v)
\end{aligned}$$

where $F_{Y^*}(v)$ is the c.d.f. of an Erlang distribution with shape parameter $i + m_j + 1 = m_1 + \cdots + m_k + 1$ and scale parameter θ. The last line follows from the fact that the multiple integral in the previous line is the convolution of two Erlang random variables. Furthermore, the sum of Erlang random variables with a common scale parameter is an Erlang random variable with the same scale parameter, but with a shape parameter equal to the sum of the shape parameters of the two Erlang random variables. Thus,

$$F_{Y^*}(v) = 1 - \sum_{n=0}^{m_1+\cdots+m_k} \frac{e^{-v/\theta} v^n}{\theta^n n!}$$

and

$$\mathbb{E}(X_j I_{\{S<v\}}) = \sum_{m_1=1}^\infty \cdots \sum_{m_k=1}^\infty \alpha_{\vec{m}}\, m_j\, \theta \left(1 - \sum_{n=0}^{m_1+\cdots+m_k} \frac{e^{-v/\theta} v^n}{\theta^n n!}\right)$$

Since

$$\mathbb{E}(X_j) = \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}}\, m_j\, \theta,$$

we have

$$\begin{aligned}
\mathbb{E}(X_j) - \mathbb{E}(X_j I_{\{S<v\}}) &= \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}}\, m_j\, \theta \left(\sum_{n=0}^{m_1+\cdots+m_k} \frac{e^{-v/\theta} v^n}{\theta^n n!} \right) \\
&= \theta^2 \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}}\, m_j \left(\sum_{n=0}^{m_1+\cdots+m_k} p(v, n+1, \theta) \right)
\end{aligned}$$

and so,

$$\begin{aligned}
\mathbb{E}(X_j | S > v) &= \frac{1}{1-p} \left\{ \mathbb{E}(X_j) - \mathbb{E}(X_j I_{\{S<v\}}) \right\} \\
&= \frac{\theta^2}{1-p} \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}}\, m_j \left(\sum_{n=0}^{m_1+\cdots+m_k} p(v, n+1, \theta) \right).
\end{aligned}$$

□

Additionally, we show below that the sum of the individual risk contributions to CTE gives the CTE of the aggregate loss.

$$\begin{aligned}
\mathbb{E}\left(\sum_{j=1}^{k} X_j | S > v \right) &= \frac{\theta^2}{1-p} \sum_{j=1}^{k} \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}}\, m_j \left(\sum_{n=0}^{m_1+\cdots+m_k} p(v, n+1, \theta) \right) \\
&= \frac{\theta^2}{1-p} \sum_{m_1=1}^{\infty} \cdots \sum_{m_k=1}^{\infty} \alpha_{\vec{m}} (m_1 + \cdots + m_k) \left(\sum_{n=0}^{m_1+\cdots+m_k} p(v, n+1, \theta) \right) \\
&= \frac{\theta^2}{1-p} \sum_{i=1}^{\infty} \sum_{m_1+\cdots+m_k=i}^{\infty} \alpha_{\vec{m}}\, i \left(\sum_{n=0}^{i} p(v, n+1, \theta) \right) \\
&= \frac{\theta^2}{1-p} \sum_{i=1}^{\infty} \alpha_i^S\, i \left(\sum_{n=0}^{i} p(v, n+1, \theta) \right)
\end{aligned}$$

$$
\begin{aligned}
&= \frac{\theta^2}{1-p}\sum_{i=1}^{\infty}\sum_{n=0}^{i}\alpha_i^S\, i\, p(v, n+1, \theta) \\
&= \frac{\theta^2}{1-p}\sum_{n=0}^{\infty}\sum_{i=n}^{\infty}\alpha_i^S\, i\, p(v, n+1, \theta) \\
&= CTE_p(S)
\end{aligned}
$$

where the second to last line follows from a change in the order of summation. Note that the formula for CTE is the second to last line follows from (3.8).

3.5 The Optimal Dividends Problem

3.5.1 Jump-Diffusion Processes Revisited

Henceforth, we change the argument of the Laplace transform from θ to t to prevent any conflicts with the scale parameter of the univariate Erlang distribution, and we also assume that $q > 0$. For a jump-diffusion process where C_i has a univariate Erlang distribution given by

$$f_C(x) = \sum_{i=1}^{M}\alpha_i \frac{x^{i-1}e^{-x/\theta}}{\theta^i(i-1)!}\ ,\ x > 0, \tag{3.10}$$

we can obtain an explicit expression for the scale function. By (1.8), the Laplace exponent becomes

$$\psi(t) = \frac{1}{2}\sigma^2 t^2 + ct + \lambda\left(\sum_{k=1}^{M}\alpha_k(1+\theta t)^{-k} - 1\right) - q \tag{3.11}$$

and we can write the Laplace transform of the scale function as

$$L_{W^{(q)}}(t) = \frac{1}{\psi(t) - q}. \tag{3.12}$$

Multiplying both the numerator and denominator by $(1+\theta t)^M$, we have

$$L_{W^{(q)}}(t) = \frac{(1+\theta t)^M}{(\psi(t) - q)(1+\theta t)^M} \tag{3.13}$$

and so the Laplace transform is a rational function with a polynomial of order M in the numerator, and one of order $M+2$ in the denominator. Assuming that the (possibly complex) roots $(t_i)_{i=1}^{M+2}$ of

$$(\psi(t)-q)(1+\theta t)^k = 0 \tag{3.14}$$

are distinct, (3.12) can be rewritten using a partial fraction decomposition as

$$\sum_{i=1}^{M+2} \frac{D_i}{t-t_i} \tag{3.15}$$

where $\{D_i\}_{i=1}^{M+2}$ are given by

$$D_i = \frac{1}{\psi'(\theta_i)}. \tag{3.16}$$

Inverting the Laplace transform, we have

$$W^{(q)}(x) = \sum_{i=1}^{M+2} D_i e^{\theta_i x}. \tag{3.17}$$

Remark 3.5.1. *The expression for the D_i's are obtained by the Cauchy residue theorem (see 1.14) and a simple application of L'Hôpital's rule. An alternate representation for the D_i's is given by*

$$D_i = \frac{(1+\theta t_i)^{(M+2)}}{\sigma^2/2 \prod_{j=1, j\neq i}^{M+2} (t_i - t_j)}. \tag{3.18}$$

Remark 3.5.2. *If the roots are not distinct, we can still obtain an expression for the scale function by means of Cauchy residue theorem or the method of partial fraction decomposition.*

Remark 3.5.3. *We can see that* (3.17) *is infinitely differentiable and so, our function of interest $W^{(q)'}$ is given by*

$$W^{(q)'}(x) = \sum_{i=1}^{M+2} D_i \theta_i e^{\theta_i x}. \tag{3.19}$$

Since most quantities of interest in this literature are provided in terms of scale functions, given this expression for $W^{(q)}(x)$, it is easy to find closed form expressions for such quantities. For example, the expression (1.16) for the two sided exit problem under this process is given by

$$\mathbb{E}_x(e^{-q\tau_a^+}\mathbf{1}_{(\tau_0^- > \tau_a^+)}) = \begin{cases} \frac{\sum_{i=1}^{M+2} D_i e^{\theta_i x}}{\sum_{i=1}^{M+2} D_i e^{\theta_i a}} & , 0 \le x \le a \\ 0 & , x < 0 \end{cases} \tag{3.20}$$

and setting $q = 0$, the expression (1.19) for the probability of ruin can be written as

$$\mathbb{P}(\tau_0^- < \infty) = \begin{cases} 1 - (c - \lambda\theta \sum_{i=1}^{M} \alpha_i r_i) \sum_{i=1}^{M+2} D_i e^{\theta_i x} & , \text{ if } c - \lambda\theta\sum_{i=1}^{M}\alpha_i r_i > 0 \\ 1 & , \text{ otherwise} \end{cases}. \tag{3.21}$$

3.5.2 Analysis of the Roots

For notation purposes, for any $z \in \mathbb{C}$ (where $\mathbb{C}$ is the set of all complex numbers), $\Re(z)$ denotes the real part of z, whereas $\Im(z)$ denotes the imaginary part of z.

There are at most $M + 1$ complex roots of (3.14), which occur in conjugate pairs. The largest and only positive real root is $\Phi(q)$ and for convenience, we let $t_1 = \Phi(q)$. We also have that the real part of all other roots is negative; that is, $\Re(t_i) < 0$ for $i = 2, \ldots M + 2$. As such, $\Phi(q) = t_1 = \sup\{t_i,\ i = 1, \ldots, M + 2\}$.

To prove that $Re(t_i) < 0$ for $i = 2, \ldots, M + 2$, we use Rouché's theorem which is given below (alternatively, see [1]).

Theorem 3.5.1 (Rouché's theorem). *Let γ be a simple closed curve, and let the functions $f(z)$ and $g(z)$ be analytic both in the region enclosed by γ and on the curve γ. Assume that f and g satisfy $|g(z)| < |f(z)|$ on the curve γ. Then $f(z)$ and $f(z) + g(z)$ have the same number of zeros in the region enclosed by γ.*

The proof of our claim is provided below.

Proof. Suppose that γ is a semi-circular contour of radius R in the complex plane centred at the origin. The closed curve is given by the arc Re^{it} that extends from $t = -\pi/2$ to $t = \pi/2$, and also by the line segment on the imaginary axis from $-iR$ to iR. Mathematically,

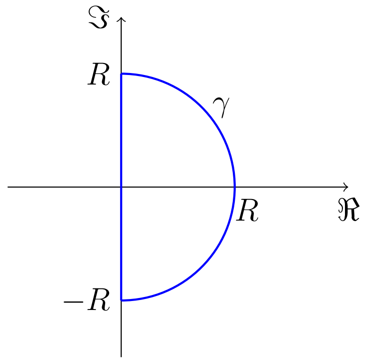


Figure 3.1: The closed curve γ on the complex plane

$\gamma = \{Re^{it} \,|\, -\pi/2 \le t \le \pi/2\} \cup \{t \,|\, -iR \le t \le iR\}$. The contour is shown graphically in Figure 3.1. We denote

$$f(z) = \frac{1}{2}\sigma^2 z^2 + cz - (\lambda + q)\text{ , and } g(z) = \lambda \sum_{k=1}^{M} \alpha_k (1+\theta z)^{-k}$$

so that $f(z) + g(z) = \psi(z) - q$.

When $\Re(z) \ge 0$, $|1+\theta z| \ge 1$ since $\theta > 0$. Therefore, for $\Re(z) \ge 0$,

$$\begin{aligned} |g(z)| &\le \lambda \sum_{k=1}^{M} \alpha_k \left|(1+\theta z)^{-k}\right| && \text{(by the triangle inequality)} \\ &= \lambda \sum_{k=1}^{M} \alpha_k (|1+\theta z|)^{-k} \\ &\le \lambda \sum_{k=1}^{M} \alpha_k (1)^{-k} \\ &= \lambda \end{aligned}$$

since $\sum_{k=1}^{M} \alpha_k = 1$. This implies that $|g(z)| \le \lambda$ on γ and inside the region enclosed by γ when $\Re(z) \ge 0$.

To analyse $f(z)$ on the line segment, let $z = it$ for $t \in [-R, R]$. We have

$$|f(z)| = \left|\frac{1}{2}\sigma^2 z^2 + cz - (\lambda + q)\right|$$

$$\begin{aligned}
&= \left| -\frac{1}{2}\sigma^2 t^2 + cit - (\lambda + q) \right| && \text{(by substituting } z = it\text{)} \\
&= \left| \left(-\frac{1}{2}\sigma^2 t^2 - (\lambda + q) \right) + cit \right| \\
&\geq \left| \left(-\frac{1}{2}\sigma^2 t^2 - (\lambda + q) \right) \right| && \text{(since } |z| \geq |\Re(z)|\text{)} \\
&= \frac{1}{2}\sigma^2 t^2 + (\lambda + q) \\
&> \lambda && \text{(since } q > 0\text{).}
\end{aligned}$$

Finally, we analyse $f(z)$ on the semi-circular contour $\{Re^{it} \mid -\pi/2 \leq t \leq \pi/2\}$. On this semi-circle,

$$\begin{aligned}
|f(z)| &\geq \left| \frac{1}{2}\sigma^2 z^2 + cz \right| - (\lambda + q) && \text{(by the triangle inequality)} \\
&= \left| z \right| \cdot \left| \frac{1}{2}\sigma^2 z + c \right| - (\lambda + q) && \text{(since } |z_1 z_2| = |z_1||z_2| \text{ for } z_1, z_2 \in \mathbb{C}\text{)} \\
&= R \cdot \left| \frac{1}{2}\sigma^2 z + c \right| - (\lambda + q) && \text{(since } |z| = |Re^{it}| = R\text{)} \\
&\geq R \cdot \left| \Re\left(\frac{1}{2}\sigma^2 z + c \right) \right| - (\lambda + q) && \text{(since } |z| \geq |\Re(z)|\text{)} \\
&= R \cdot \left| \frac{1}{2}\sigma^2 R\cos(t) + c \right| - (\lambda + q) && \text{(since } z = R\cos(t) + iR\sin(t)\text{)} \\
&\geq R \cdot c - (\lambda + q) && \text{(since } \cos(t) \geq 0 \text{ for } -\pi/2 \leq t \leq \pi/2\text{).}
\end{aligned}$$

Therefore, $|f(z)| > \lambda$ if $R > \frac{2\lambda+q}{c}$.

Overall, we have shown that $|f(z)| > \lambda \geq |g(z)|$ on the curve γ for some $R > \frac{2\lambda+q}{c}$. Suppose that we set R to be large enough so that the positive root of the quadratic equation $f(z)$ lies in the region enclosed by γ. Therefore, by Rouché's theorem, $f(z)$ and $f(z) + g(z) = \psi(z) - q$ have the same number of roots in the region enclosed by γ. Since $f(z)$ has one root in this semi-circle, then so does $\psi(z) - q$. If we let $R \to \infty$, the proof still holds and hence, we can conclude that $\psi(z) - q$ has exactly one root in the right-half plane which is given by $z = \Phi(q)$. As such, all other roots lie in the left-half plane. □

From the above analysis, only one term (the term in $\Phi(q)$) in (3.17) is exponentially increasing while the other terms are exponentially decreasing.

Remark 3.5.4. *Similar proofs of this result under more general settings can be found in [16] and [22].*

3.6 On the optimality of a barrier strategy

A sufficient condition

Suppose that $M \to \infty$ in (3.2). Then, by (3.6), the tail probability for the univariate Erlang mixture is given by

$$\begin{aligned}
&\sum_{i=1}^{\infty} \alpha_i \sum_{j=0}^{i-1} \frac{e^{-x/\theta} x^j}{\theta^j j!} \\
&= \sum_{j=0}^{\infty} \left(\sum_{i=j+1}^{\infty} \alpha_i \right) \frac{e^{-x/\theta} x^j}{\theta^j j!} \\
&= \sum_{j=0}^{\infty} \bar{P}_j \frac{e^{-x/\theta} x^j}{\theta^j j!} \qquad (3.22)
\end{aligned}$$

where

$$\bar{P}_j = \sum_{i=j+1}^{\infty} \alpha_i. \qquad (3.23)$$

Note that $1 = \bar{P}_1 \geq \bar{P}_2 \geq \bar{P}_3 \geq \dots$, which implies that $\{\bar{P}_j\}_{j \geq 1}$ is a non-decreasing sequence. By theorem 3.2 of [12], (3.22) has a density which is logarithmically convex on $(0, \infty)$ if α_{k+1}/α_k is increasing in $k = 1, 2, \dots$. This condition on the weights of the mixture is equivalent to having $\alpha_k \alpha_{k+2} - \alpha_{k+1}^2 \geq 0$, $k = 1, 2, \dots$, i.e. $\{\alpha_k\}_{k \geq 1}$ is a log-convex sequence.

Therefore, if $\{\alpha_k\}_{k \geq 1}$ is a log-convex sequence, then the density of the univariate Erlang mixture given by (3.2) is log-convex. By theorem 2.3.4, for a jump-diffusion process with a claims distribution given by this mixture, a dividend barrier strategy is an optimal strategy.

An asymptotic condition

From the previous chapter, recall that we provided a sufficient condition on the parameters of the surplus process for the optimality of a barrier strategy. Specifically, if $\Phi(q) < \frac{q}{c}$, then a barrier strategy is an optimal strategy. With respect to the surplus being modelled by a jump-diffusion process with a univariate Erlang mixture claims distribution, we expect that a barrier strategy becomes optimal when either the volatility σ or scale parameter θ is sufficiently large. A numerical example is provided in the last section of the next chapter.

Chapter 4

An Algorithm for Fitting Univariate Erlang Mixtures to Data

4.1 Introduction

In the last chapter, we have dealt with the univariate Erlang mixture and showed that many quantities of interest in the insurance industry have closed form expressions. Moreover, a tractable form for the scale function translates into closed form expressions for many quantities in the literature on Lévy insurance risk processes. We will now see how this mixture performs from a numerical point of view.

In this chapter, we propose an alternative algorithm to that provided in [20] for fitting the univariate Erlang distribution to data. This algorithm uses the same EM algorithm as in [20] for determining the weights (α_i, $i = 1, \ldots, M$) and scale parameter (θ), but we employ a different method for selecting the shape parameters (r_i, $i = 1, \ldots, M$). For completeness and convenience, the entire algorithm is provided. Using this algorithm, we will then show the goodness of fit of this distribution to data from some common distributions. Finally, to summarize our work in the past chapters, we will provide a numerical example for the optimal dividends problem where the optimal strategy is a barrier strategy.

4.2 The Expectation Maximization (EM) Algorithm

The EM algorithm is a popular and useful tool in statistical estimation problems. It is used to compute maximum likelihood estimates for incomplete data problems, where with some additional data, the maximum likelihood estimation procedure would be simpler. This general iterative algorithm, which was proposed by Dempster, Laird, and Rubin in 1977, allows us to reformulate an incomplete data problem to a complete data problem where the maximum likelihood estimation is more tractable. Additionally, compared to other iterative algorithms such as the Newton-Raphson and Fisher's scoring methods, it is numerically stable and has reliable convergence.

We will now look at the EM algorithm presented in [20] for a univariate Erlang mixture.

The univariate distribution (3.2) can be written as

$$f(x|\vec{\Phi}) = \sum_{i=1}^{M} \alpha_i p(x|r_i, \theta)$$

where

$$p(x|r_i, \theta) = \frac{x^{r_i-1} e^{-x/\theta}}{\theta^{r_i}(r_i-1)!}\, ,\ x > 0. \tag{4.1}$$

Assume that the dataset is given by $\vec{x} = (x_1, \ldots, x_n)$. The log-likelihood is

$$l(\vec{\Phi}|\vec{x}) = \sum_{j=1}^{n} \log\left(\sum_{i=1}^{M} \alpha_i p(x|r_i, \theta)\right).$$

However, this is difficult to maximize as it involves the logarithm of a sum. We assume that there exists some unobservable data $Y_j \in (r_1, \ldots, r_M)$, $j = 1, \ldots, n$. The complete data set is now $(\vec{x}, \vec{Y}) = \{(x_1, Y_1), \ldots, (x_n, Y_n)\}$. When $Y_j = r_i$, this indicates that the j-th data point x_j was generated from an Erlang distribution with density function $p(x|r_i, \theta)$. Incorporating the unobservable data into our model, the complete-data log-likelihood function is now given by

$$l(\vec{\Phi}|\vec{x}, \vec{Y}) = \sum_{j=1}^{n} \log p(x_j, Y_j|\vec{\Phi}).$$

Given current estimates of the parameters $\vec{\Phi}^{(k-1)} = \{\vec{\alpha}^{(k-1)}, \theta^{(k-1)}\}$, the density function of Y_j is

$$q(y|x_j, \vec{\Phi}^{(k-1)}) = \frac{p(x_j, y|\vec{\Phi}^{(k-1)})}{p(x_j|\vec{\Phi}^{(k-1)})}$$

where $p(x|\vec{\Phi}^{(k-1)})$ is the marginal density function of X.

We can now determine the expectation of the complete-data log-likelihood function (this step is called the E-step). We have

$$\begin{aligned} Q(\vec{\Phi}|\vec{\Phi}^{(k-1)}) &= \sum_{j=1}^{n} \int \log p(x_j, y|\vec{\Phi}) \cdot q(y|x_j, \vec{\Phi}^{(k-1)}) dy \\ &= \sum_{j=1}^{n} \sum_{i=1}^{M} \log p(x_j, r_i|\vec{\Phi}) \cdot q(r_i|x_j, \vec{\Phi}^{(k-1)}). \end{aligned}$$

The M-step involves maximizing the expectation calculated in the E-step. That is,

$$\vec{\Phi}^{(k)} = \max_{\vec{\Phi}} Q(\vec{\Phi}|\vec{\Phi}^{(k-1)}).$$

Given that we can write

$$\begin{aligned} q(r_i|x_j, \vec{\Phi}^{(k-1)}) &= \frac{p(x_j, r_i|\vec{\Phi}^{(k-1)})}{p(x_j|\vec{\Phi})} \\ &= \frac{\alpha_i x_j^{r_i - 1} \cdot e^{-x_j/\theta} / [\theta^{r_i}(r_i - 1)!]}{\sum_{m=1}^{M} \alpha_m x_j^{r_m - 1} \cdot e^{-x_j/\theta} / [\theta^{r_m}(r_m - 1)!]}, \end{aligned} \tag{4.2}$$

we can find (by setting the score function to zero) that the parameters that maximize this likelihood are given by

$$\alpha_i^{(k)} = \frac{1}{n} \sum_{j=1}^{n} q(r_i|x_j, \vec{\Phi}^{(k-1)}), \ i = 1, \ldots, M, \tag{4.3}$$

$$\theta^{(k)} = \frac{\sum_{j=1}^{n} x_j/n}{\sum_{i=1}^{M} r_i \alpha_i^{(k)}}. \tag{4.4}$$

Therefore, since we have expressions for the parameters at each iteration, this algorithm is purely iterative. As proposed in [29], at the end of each iteration, we can check for convergence by either

- computing the maximum distance of the estimated parameters of successive iterations, or
- computing the relative change in the log-likelihood $Q(\vec{\Phi}, \vec{\Phi}^{(k-1)})$ at successive iterations

and stopping when some predefined tolerance level is reached.

As also proposed in [29], since large factorials may be required, to prevent numerical underflow or overflow, we compute the logarithm of the factorials using

$$\log(n!) = \sum_{i=1}^{n} \log(i)$$

and rewrite $p(x_j, r_i|\vec{\Phi}^{(k-1)})$ in the form

$$p(x_j, r_i|\vec{\Phi}^{(k-1)}) = \alpha_i \exp\{(r_i - 1)\log(x_j) - x_j/\theta - r_i \log(\theta) - \log((r_i - 1)!)\}.$$

We can also prevent numerical underflow or overflow in the algorithm by scaling the values appropriately in the dataset. For example, suppose we scale the values by a factor of β, where $\beta \in \mathbb{R}$ and obtain a new dataset $(\beta x_1, \ldots, \beta x_n)$. Furthermore, suppose that, for this scaled dataset, the maximum likelihood estimate of the scale parameter is $\hat{\theta}$. Then, for our regular dataset, the maximum likelihood estimate of the scale parameter will be $\hat{\theta}/\beta$.

4.3 Parameter Initialization

We set M equal to the number of data points n and $r_i = i$, $i = 1, 2, \ldots, M$. Based on the Tijms' approximation (3.1), given θ, α_i is estimated as the frequency of data points on the interval $((i-1)\theta, i\theta]$. We keep only non-zero α_i's since any zero α_i's will stay at zero throughout the iterations. Initializing the scale parameter θ is most challenging since the other parameters are dependent on its value. To select an initial value for θ, we consider n possible values for θ, which are given by

$$\hat{\theta}_j = \frac{\sum_{i=1}^{n} x_i}{n \cdot j}, \; j = 1, \ldots, n$$

For $j = 1, \ldots, n$, we do the following:

- Given $\hat{\theta}_j$, we initialize r_i and α_i, $i = 1, \dots, M$ as above. We "refine" $\hat{\theta}_j$ once by computing (4.4), and then re-estimating α_i, $i = 1, \dots, M$ as above.
- After removing the Erlang branches with zero weights (i.e. with $\alpha_i = 0$), to prevent overfitting, if the total number of parameters is less than n, run the EM algorithm on these initial parameter estimates with a weak convergence tolerance criterion.
- Compute the Consistent Akaike Information Criterion (CAIC) as

  $$CAIC = -2l + k(\ln(n) + 1),$$

  where k is the number of parameters, n is the number of data points and l is the log likelihood. Denote the value as $CAIC_j$.
- Compute the difference between the fifth moments of the sample data and model. Denote the value as d_j.

We now have two sets of model selection criteria, namely $\mathcal{C} = \{CAIC_j\}_{j=1}^{n}$ and $\mathcal{D} = \{d_j\}_{j=1}^{n}$. If any element in $\mathcal{D}$ is negative, discard it along with the corresponding element in $\mathcal{C}$. Find the statistical ranks of elements of $\mathcal{C}$ and $\mathcal{D}$, and choose the initial value of theta that has the minimum average rank. This is best explained by example. For simplicity, suppose that $n = 4$, and we have $\mathcal{C} = \{760, 925, 800, 810\}$ and $\mathcal{D} = \{20, 16, 10, 12\}$. Therefore, $rank(\mathcal{C}) = \{1, 4, 2, 3\}$ and $rank(\mathcal{D}) = \{4, 3, 1, 2\}$, which implies that the average ranked set is $\{2.5, 3.5, 1.5, 2.5\}$. Since the third element is the minimum of the set, we will choose θ_3 as the initial value of the scale parameter.

Remark 4.3.1. *The CAIC considers the balance between likelihood and model complexity and penalizes more for extra parameters than the Akaike Information Criterion (AIC). Since we have started with a large number of Erlangs, we may have insignificant Erlang branches in our mixture. As such, this criteria alone is most likely to choose a value of θ that corresponds to a small number of Erlang branches. This is the reason for considering the second criteria of comparing the fifth moments of the sample data and the model. This criteria focuses more on the tail of the distribution, and there is a trade-off between fitting the tail well and the number of initial Erlangs. We discarded any negative d_j's above because we prefer to initially overestimate the probability in the tail rather than underestimate it. Underestimating the tail is most likely to occur if the initial number of Erlangs is large. The value of θ which performs the "best" with respect to these two criteria is chosen as the initial value.*

4.4 Final Model Selection

Currently, we may be using too many branches in the mixture and need to remove some branches to compensate for overfitting. We address this overfitting issue by once again using the CIAC. We aim to minimize this criterion. After finding our initial parameters, a "greedy" backward elimination procedure is used where we remove the Erlang branch with the least weight (the smallest α_i), re-run the above EM algorithm and recalculate the CAIC. We continue removing Erlang branches with the least weight until the CAIC cannot be improved (i.e. until the CAIC starts to increase).

At this point, we have a mixture of Erlangs where the removal of the least weight does not decrease the CAIC. We will sort the remaining Erlang branches in order of their corresponding weight, from smallest to largest. Denote this set of sorted branches by $\{e_{(1)}, \ldots e_{(M^*)}\}$, where M^* is the number of remaining branches. The remainder of the algorithm is outlined below:

1. Set $k = 1$.
2. Remove the branch $e_{(k)}$, re-run the EM algorithm and recalculate the CAIC.
3. If the CAIC decreases, go to step 1. Otherwise, re-insert the branch $e_{(k)}$ and continue to step 4.
4. Set $k = k+1$. If k is less than or equal to the number of remaining Erlang branches, go to step 2. Otherwise, stop and select the current set of parameters as the optimal parameters.

4.5 A note on the model selection algorithm

An implementation of the algorithm in C++ has been provided in Appendix B for the interested reader. It was written using a free and open source, cross-platform C++ library named Eigen [15]. Our convergence criterion for the EM algorithm was stopping when the relative change in the log-likelihood was less than $tol = 10^{-4}$. To find the initial value of θ, we parallelize the algorithm provided in section 4.3. While this algorithm produces fairly good initial estimates of θ, this initial procedure is very subjective. Additionally, the user may desire control over the initial number of Erlangs or the initial fit to the data. Therefore, for each of the values of θ considered in section 4.3, the corresponding number of Erlangs and other goodness-of-fit information is output to a file. Using this information, the user can have better control over the initial parameters.

In the implementation of EM algorithm, we remove any Erlang branches that were less than $\frac{tol}{M}$, which is a very conservative tolerance. Apart from this, we did not use any other convergence acceleration methods. The final part of the algorithm (which was outlined at the end of the previous section) also uses multi-threading to take advantage of the acceleration that is provided on computers with multiple and quad-core processors. Any parallel programming is done using the OpenMP API specification, which is already implemented by many popular C++ compilers.

4.6 Numerical Results

For each of the following distributions, a sample of size 1000 was generated and the model selection algorithm was used to find the optimal set of parameters:

- The uniform distribution over the range $(0, 1)$,
- The generalized Pareto distribution with a location parameter of 2 and scale parameter of 2,
- A mixture of two gamma distributions with shape parameters 2.6 and 6.3, scale parameters 0.3125 and 0.8333, and corresponding weights 0.2 and 0.8,
- The inverse gamma distribution with a rate parameter of 30 and shape parameter of 0.2, and
- The log-normal distribution whose logarithm has mean equal to 0.03 and standard deviation 0.2.

To show visually the goodness of fit, we present the histogram of the data with an overlay of the fitted Erlang mixture, and more importantly, the percentile-percentile (PP) and quantile-quantile (QQ) plots. Since the histogram may be subjective with respective to the number/size of bins used, the reader should focus more on the PP and QQ plots when assessing the quality of fit to the data. Additionally, to quantify this goodness of fit, we perform the Anderson-Darling and Kolmogorow-Smirnov tests. The descriptions of these plots and tests are provided in [20]. The optimal set of parameters are provided in Appendix A.

Uniform distribution

A mixture of 6 Erlangs has been fitted to the 1000 sample points that were generated from the uniform distribution. The results are presented below.

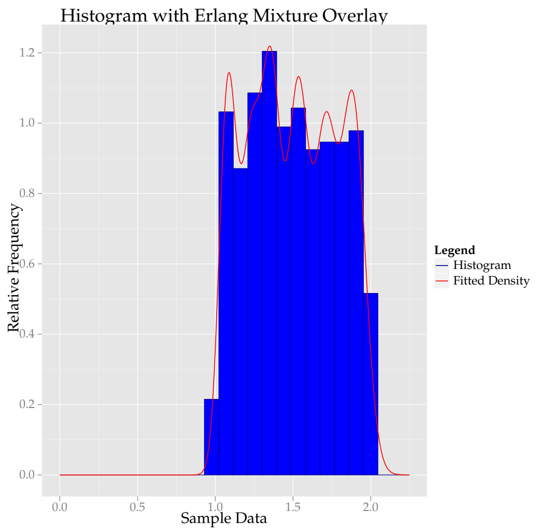
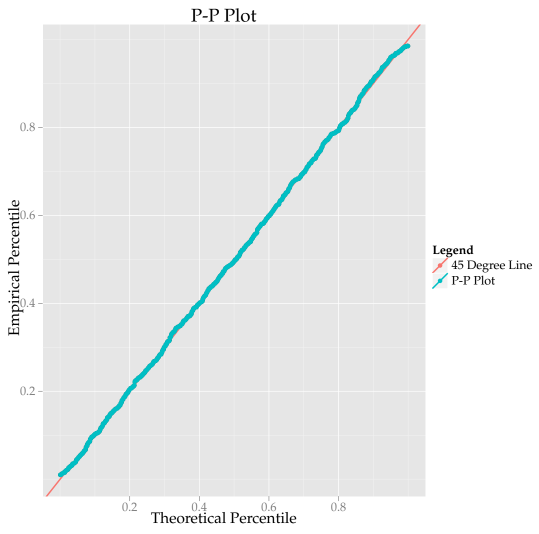
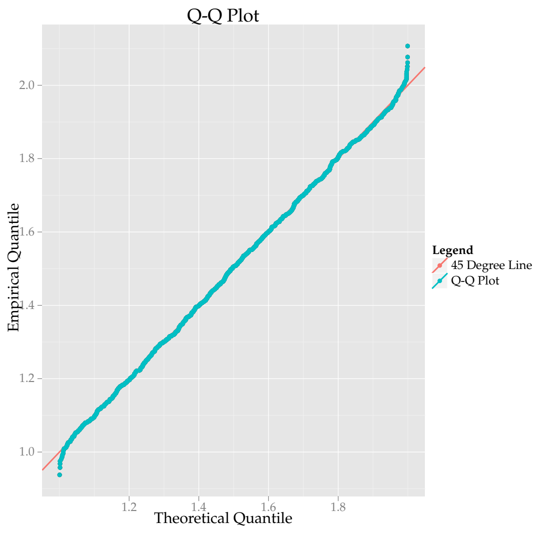


Figure 4.1: Histogram for the uniform distribution with an overlay of a mixture of 6 Erlangs



(a) PP Plot



(b) QQ Plot

Figure 4.2: PP and QQ plots for the uniform distribution

Test	Statistic	P-Value	Pass/Fail
Two-sample Kolmogorov-Smirnov test	0.015	0.9999	Pass
Anderson-Darling Test	-1.00397	0.62867	Pass

Table 4.1: Statistical tests for the uniform distribution

Generalized Pareto distribution

A mixture of 6 Erlangs has been fitted to the 1000 sample points that were generated from the generalized Pareto distribution. The results are presented below.

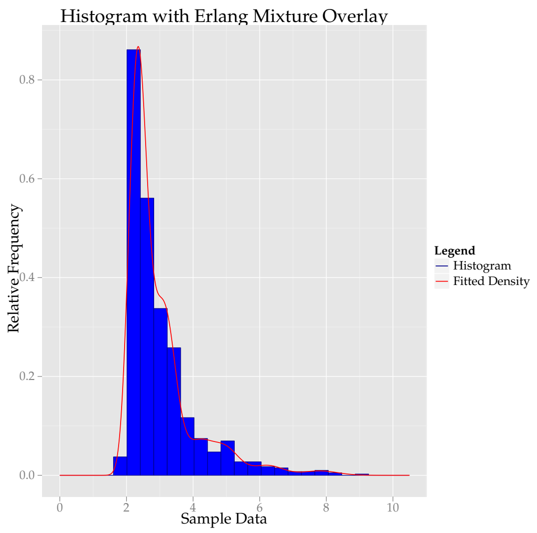
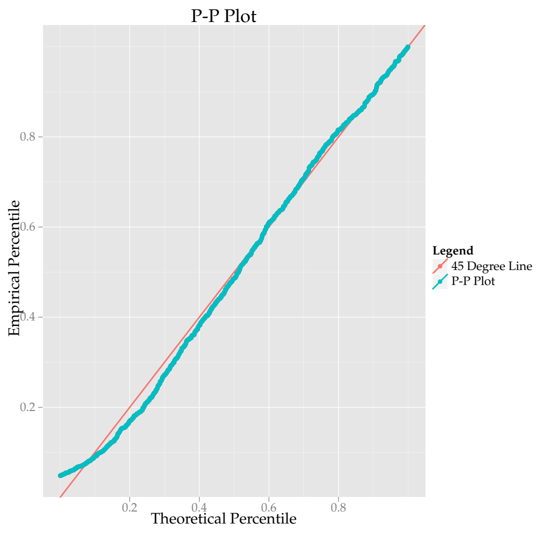
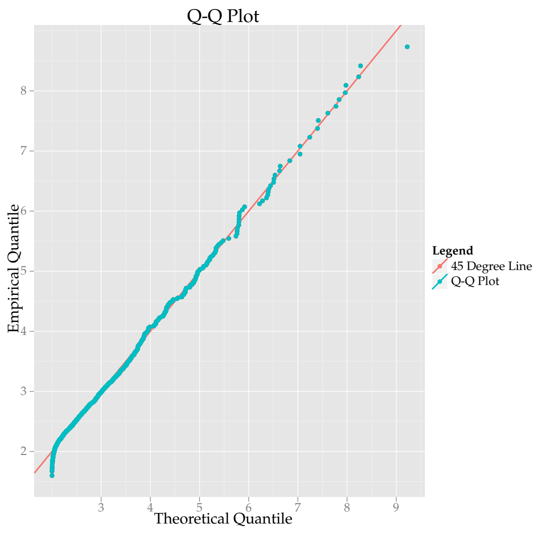


Figure 4.3: Histogram for the generalized Pareto distribution with an overlay of a mixture of 6 Erlangs



(a) PP Plot



(b) QQ Plot

Figure 4.4: PP and QQ plots for the generalized Pareto distribution

Test	Statistic	P-Value	Pass/Fail
Two-sample Kolmogorov-Smirnov test	0.048	0.1995	Pass
Anderson-Darling Test	1.40240	0.08638	Pass

Table 4.2: Statistical tests for the Generalized Pareto distribution

Mixture of two gamma distributions

A mixture of 3 Erlangs has been fitted to the 1000 sample points that were generated from the mixture of two gamma distributions. The results are presented below.

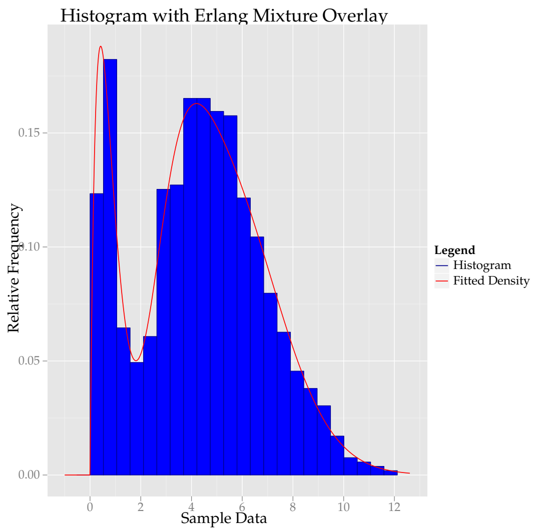
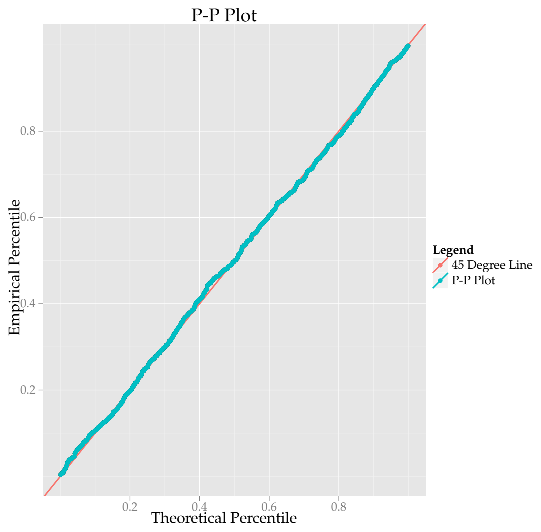
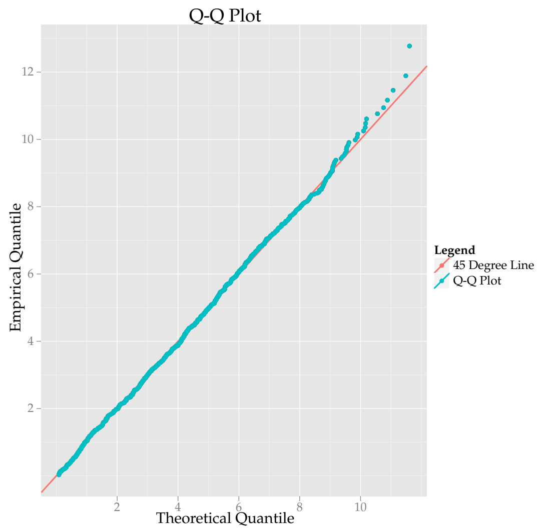


Figure 4.5: Histogram for the mixture of two gamma distributions with an overlay of a mixture of 3 Erlangs



(a) PP Plot



(b) QQ Plot

Figure 4.6: PP and QQ plots for the mixture of two gamma distributions

Test	Statistic	P-Value	Pass/Fail
Two-sample Kolmogorov-Smirnov test	0.021	0.9802	Pass
Anderson-Darling Test	-1.02723	0.63529	Pass

Table 4.3: Statistical tests for the mixture of two gamma distributions

Inverse gamma distribution

A mixture of 8 Erlangs has been fitted to the 1000 sample points that were generated from the inverse gamma distribution. The results are presented below.

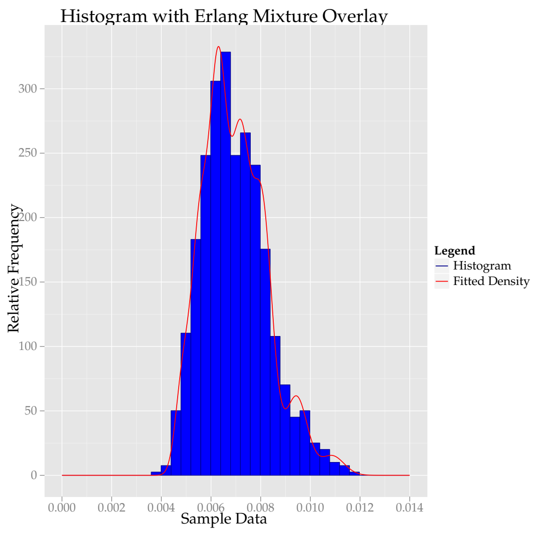
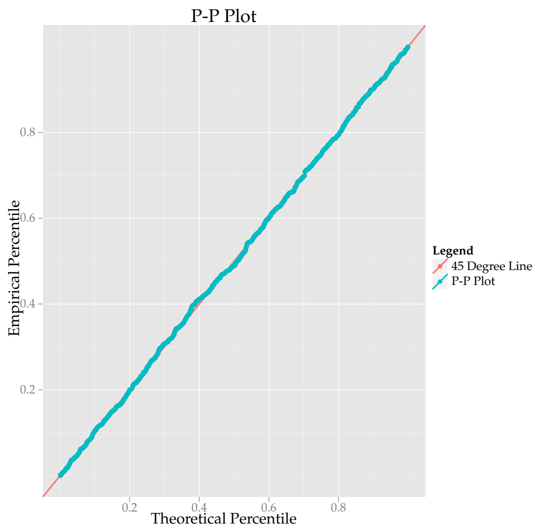
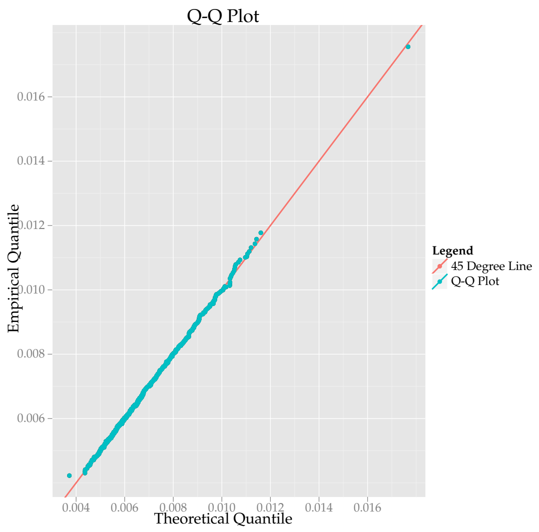


Figure 4.7: Histogram for the inverse gamma distribution with an overlay of a mixture of 8 Erlangs



(a) PP Plot



(b) QQ Plot

Figure 4.8: PP and QQ plots for the inverse gamma distribution

Test	Statistic	P-Value	Pass/Fail
Two-sample Kolmogorov-Smirnov test	0.015	0.9999	Pass
Anderson-Darling Test	-1.21674	0.68711	Pass

Table 4.4: Statistical tests for the inverse gamma distribution

Log-normal distribution

A mixture of 9 Erlangs has been fitted to the 1000 sample points that were generated from the log-normal distribution. The results are presented below.

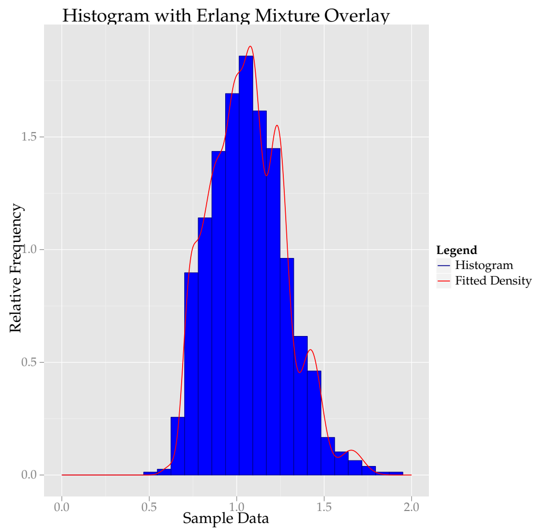
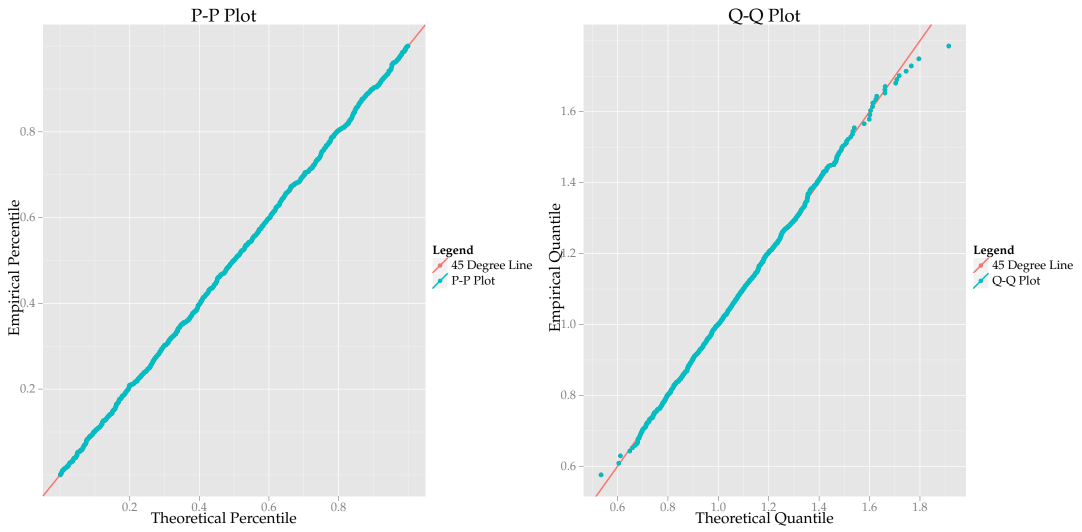


Figure 4.9: Histogram for the log-normal distribution with an overlay of a mixture of 9 Erlangs



(a) PP Plot (b) QQ Plot

Figure 4.10: PP and QQ plots for the log-normal distribution

Test	Statistic	P-Value	Pass/Fail
Two-sample Kolmogorov-Smirnov test	0.01	1	Pass
Anderson-Darling Test	-1.24060	0.69334	Pass

Table 4.5: Statistical tests for the log-normal distribution

4.7 Conclusions on the Goodness of Fit

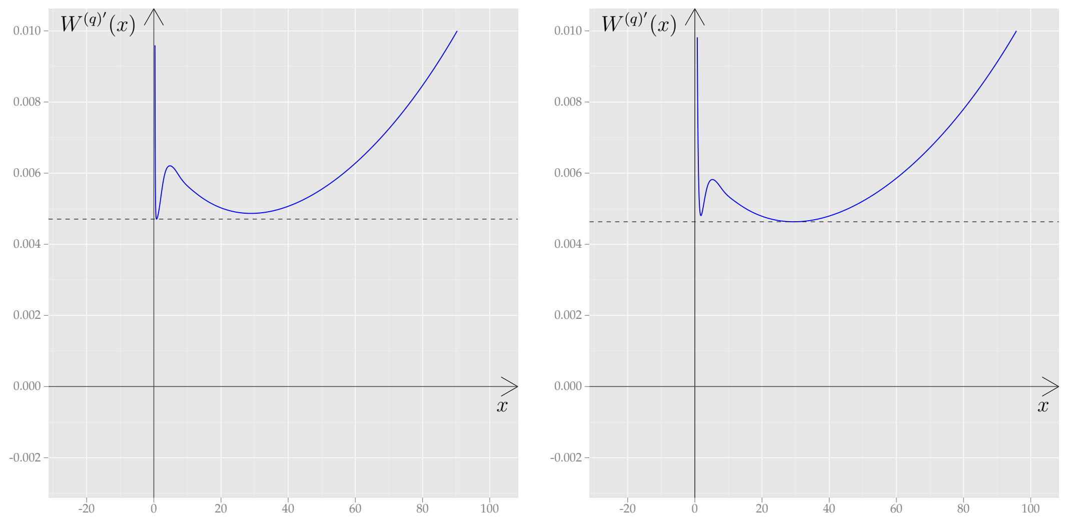
We see in the last section that the univariate Erlang mixture provides a good fit to the sample data from the various distributions. In the histograms, the overlay is seen to cover the tails of the distributions. With respect to the PP and QQ plots, the bulk of the points lie on the fitted line, and these results agree with the Kolmogorov-Smirnov and Anderson-Darling tests. We must remember that we can improve the goodness of fit by starting with a larger number of Erlangs (which is more or less synonymous with using a smaller initial value for the scale parameter). Alternatively, to prevent overfitting, we can start with a smaller number of Erlangs (which is more or less synonymous with using a larger initial value for the scale parameter).

4.8 A Jump-Diffusion Process with an Erlang Mixture Claims Distribution

In this section, we will provide a numerical example of the behaviour of $W^{(q)\prime}(x)$ with respect to σ when the surplus process is a jump-diffusion process. The claims distribution is the univariate Erlang mixture produced from running the model selection algorithm on the sample data from the mixture of two gamma distributions. Specifically, we have

$$f_C(x) = \sum_{i=1}^{3} \alpha_i \frac{x^{r_i-1} e^{-x/\theta}}{\theta^{r_i}(r_i-1)!} \tag{4.5}$$

where for $i = 1, 2$ and 3, α_i, r_i and θ are given in Table A.3. Additionally, we set $c = 46.9$, $\lambda = 10$ and $q = 0.1$. Note that the $c - \lambda\mathbb{E}(C_1) > 0$ and so the net profit condition holds. The derivative of the scale function is a mixture of 18 exponentials with possibly complex exponents and is given by (3.19) with $M = 16$. The graphs of $W^{(q)\prime}(x)$ for $\sigma = 2.8$ and $\sigma = 5.4$ are given in figure 4.11.



(a) $\sigma = 2.8$ (b) $\sigma = 5.4$

Figure 4.11: The behaviour of $W^{(q)\prime}(x)$ with respect to σ for a jump-diffusion process

In figure 4.11a, the optimal barrier level is approximately $a = 0.00471$ when $x \approx 0.82$, but we cannot conclude that the barrier strategy at a is the optimal strategy. When

σ is increased to 5.4, we see in figure 4.11a that the last local minimum is the global minimum and by theorem 2.3.2, a barrier strategy is an optimal strategy. Moreover, the optimal barrier level is approximately $a^* = 0.00463$ when $x \approx 29.48$.

If we let σ be sufficiently large so that $\Phi(q) < q/c$, then figure 4.12 depicts the shape of $\log W^{(q)'}(x)$. Additionally, since the $\log$ function is monotone increasing, we can see that the last local minimum is the global minimum and by theorem 2.3.2, a barrier strategy is once again an optimal strategy. Moreover, the optimal barrier level is approximately $a^* = 3.17 \times 10^{-5}$ when $x \approx 37.92$.

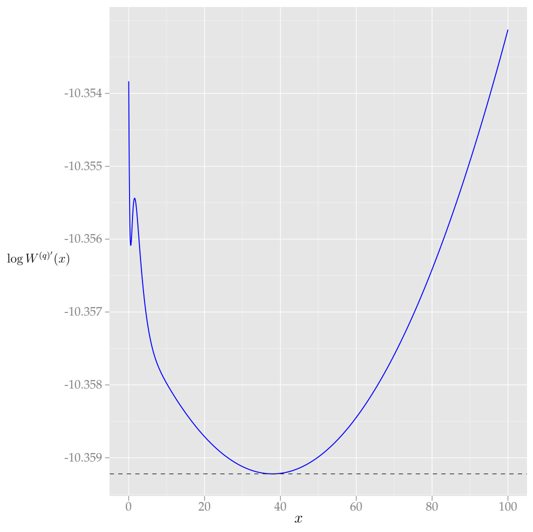


Figure 4.12: $W^{(q)'}(x)$ when $\Phi(q) < q/c$

Chapter 5

Concluding Remarks

In the past two years, there have been considerable advancements in the challenging field of Lévy insurance risk processes. It turns out that the barrier strategy is optimal for a wide class of Lévy measures. In this thesis, we provided a bound on the parameters of the surplus process for the optimality of a dividend barrier strategy when the surplus was modelled by a jump diffusion process. This bound was mainly asymptotic and can be improved upon in the future to achieve a better understanding of the relationship between the Brownian motion component and the jump distribution, and its effect on the existence of an optimal barrier strategy. Additionally, it was shown numerically at the end of the previous chapter for a specific jump-diffusion process that a barrier strategy is optimal when the Brownian motion component is increased, and this agrees with our earlier analysis.

Many results in this field are provided in terms of scale functions and there are a few cases where it's Laplace transform is analytically invertible. As such, we turned our focus to the case when the claims distribution of the jump diffusion process was given by a mixture of Erlangs. This claims distribution allows us to approximate any positive continuous distribution to any desired accuracy. Under this specific surplus process, the scale function admits a closed form representation and is given by a sum of exponential terms. The exponents are the roots of a polynomial, and thus, the scale function can be easily derived in practice. Moreover, we saw that closed form expressions can be derived for risk measures, and for the Euler risk contribution to Conditional Tail Expectation. Overall, for these features, this univariate Erlang mixture seems like a viable and tractable model for the claims process.

In the last chapter, we proposed an algorithm for fitting the univariate Erlang mixture to data. Furthermore, an implementation of this algorithm in C++ was provided in Appendix A. The algorithm's structure allows us to use parallel programming techniques to achieve better performance on multi-core processors. It is important to note that the degree of overfitting can be controlled by the initial value of the scale parameter; a smaller initial value usually results in a larger number of Erlangs being fitted to the data, and vice-versa. Numerical results have also been provided to show that this distribution fits well to sample data from various distributions.

In the future, the current surplus process could be modified to include solvency constraints. The original problem may produce barrier levels that are too low and in reality, there exist solvency measures and requirements, internal and external, which may prevent such a strategy from being implemented. In [26], we can find results for diffusion processes when the probability of a negative surplus is restricted to be below some desired level. In addition to this restriction, another possible constraint that can be imposed is a VaR constraint. As many institutions have used VaR as a risk management tool to determine, for example, their regulatory capital and the adequacy of surplus, such a constraint will result in more acceptable and more realistic dividend barrier levels.

APPENDICES

Appendix A

Parameters of the Fitted Distributions

Table A.1: Fitted Parameters for the uniform distribution

i	Shape Parameter, r_i	Weight, α_i
1	369	0.15647697930800109
2	419	0.13162608369306494
3	464	0.17278119487835644
4	523	0.17943433053112884
5	583	0.16679480481929373
6	643	0.19288660677015473
Scale Parameter, θ		0.0029420252125011391

Table A.2: Fitted Parameters for the generalized Pareto distribution

i	Shape Parameter, r_i	Weight, α_i
1	73	0.59754633925241762
2	98	0.258276980750832
3	130	0.058377087038487631
4	154	0.052565776308206101
5	193	0.022051441166845528
6	241	0.011182375483211757
Scale Parameter, θ		0.032591372951062007

Table A.3: Fitted Parameters for the mixture of two gamma distributions

i	Shape Parameter, r_i	Weight, α_i
1	2	0.21406401214945295
2	10	0.45172930549903734
3	16	0.33420668235150935
Scale Parameter, θ		0.41877847415502589

Table A.4: Fitted Parameters for the inverse gamma distribution

i	Shape Parameter, r_i	Weight, α_i
1	247	0.063117836046342496
2	281	0.15404344468519354
3	318	0.26707468196859258
4	362	0.23050369897960649
5	406	0.20087520249933072
6	475	0.065868334874041909
7	547	0.017517799947890836
8	881	0.00099900099900099878
Scale Parameter, θ		1.9935560855250973e-005

Table A.5: Fitted Parameters for the log-normal distribution

i	Shape Parameter, r_i	Weight, α_i
1	243	0.0034072257115805868
2	291	0.091941680764468625
3	325	0.088443301325459564
4	354	0.11641001074839706
5	391	0.17525584216878654
6	432	0.21577396755272912
7	488	0.20855269871596979
8	562	0.082431908375046367
9	653	0.017783364637562603
Scale Parameter, θ		0.0025405392162721214

Appendix B

A C++ Implementation of the Model Selection Algorithm

The implementation consists of three files which have been commented for ease of readability. The files include:

1. The header file “EMData.h” which contains the declaration of the “EMData” class. This class consists of an object which will represent a mixture of Erlangs, and also methods which are useful for executing the EM algorithm.

2. The file “Emdat.cc” in which the objects and methods declared in the “EMData” class are implemented.

3. The main file “emalg.cpp” which contains the main method, the stepwise algorithm and other methods which are not specific to the “EMData” class.

The header file - EMData.h

```
// This code uses the free Eigen library available at http://eigen.tuxfamily.org.
#include <iostream>
#include <fstream>
#include <limits>
#include <iomanip>
#include <vector>
#include <algorithm>
```

```
#include <time.h>
#include <Eigen/Dense>
#include <Eigen/StdVector>
#include <omp.h>

#ifndef EMData_h__
#define EMData_h__

/**
* The Emdat Class. The constructor returns an object with the parameters of the univariate
* Erlang mixture, and other information required for the EM algorithm. The methods in this
* class allow us to execute the EM algorithm on an Emdat object. To understand the methods,
* it is best to first understand the private members. As such, please scroll to the end of
* the file to learn about these private members.
*/
class EMData{
public:
    /**
     * Default constructor. Creates an empty Emdat object. It is included for completeness.
     */
    EMData(void);

    /**
     * Main Constructor. Constructs an Emdat object that represents a mixture of Erlangs.
     * @param numErlangs The number of Erlangs in the mixture.
     */
    EMData(int numErlangs);

    /**
     * Default Destructor. Deletes an Emdat object.
     */
    virtual ~EMData(void);

    /**
     * Sets the number of Erlangs.
     * @param newM The new number of Erlangs.
     */
    void setM(int newM);

    /**
     * Gets M, the number of Erlangs.
     * @return M.
     */
    int getM();

    /**
     * Sets the value of theta from either the previous or current iterations.
     * @param fromPrevIteration If true, the value of theta from the previous iteration will
     *                          be set, otherwise the value of theta from the current
     *                          iteration will be set.
     * @param theta             The new value for theta.
     */
    void setTheta(bool fromPrevIteration, double theta);

    /**
     * Gets the value of theta from either the previous or current iterations.
     * @param fromPrevIteration If true, the value of theta from the previous iteration will
```

```
 *                          be returned, otherwise the value of theta from the current
 *                          iteration will be returned.
 * @return prevTheta if fromPrevIteration == true, currTheta otherwise.
 */
double getTheta(bool fromPrevIteration);

/**
 * Sets the values of the weights (alphas) from either the previous or current iterations.
 * @param fromPrevIteration If true, the weights from the previous iteration (prevAlphas)
 *                          will be set, otherwise the weights from the current iteration
 *                          (currAlphas) will be set.
 * @param [in] values   The new weights.
 */
void setAlphas(bool fromPrevIteration, Eigen::ArrayXd& values);

/**
 * Gets the values of the weights (alphas) from either the previous or current iterations.
 * @param fromPrevIteration If true, a reference to the weights from the previous ↩
    iteration
 *                          (prevAlphas) will be returned, otherwise a reference to the
 *                          weights from the current iteration (currAlphas) will be
 *                          returned.
 * @return A non-constant reference to prevAlphas if fromPrevIteration == true,
 *         currAlphas otherwise.
 */
Eigen::ArrayXd& getAlphas(bool fromPrevIteration);

/**
 * Rebalance the weights (alphas). Performed if a branch is to be removed. The weight
 * corresponding to the branch that is to be removed will be reallocated to either the
 * next branch or the previous branch in the mixture.
 * @param pos               The position of the branch to be removed.
 * @param fromPrevIteration If true, the weights from the previous iteration (prevAlphas)
 *                          will be rebalanced, otherwise the weights from the current
 *                          iteration (currAlphas) will be rebalanced.
 * @return True if there are at least two branches in the mixture, false otherwise.
 */
bool rebalanceAlphas(int pos, bool fromPrevIteration);

/**
 * Gets the values of the shape parameters.
 * @return A non-constant reference to sample.
 */
Eigen::ArrayXi& getShapeParams(void);

/**
 * Sets the value of the log likelihood.
 * @param newLogLik The new value of the log likelihood.
 */
void setLogLikelihood(double newLogLik);

/**
 * Gets the value of the log likelihood.
 * @return logLikelihood.
 */
double getLogLikelihood(void);
```

```
/**
 * Computes the log factorials.
 */
void computeLogFactorials(void);

/**
 * Gets the values of the log factorials.
 * @return A non-constant reference to logFactorials.
 */
Eigen::ArrayXd& getLogFactorials(void);

/**
 * Removes a branch from the Erlang mixture.
 * @param pos The index of the branch to be removed.
 * @return True if 'pos' is a valid index, false otherwise.
 */
bool removeBranch(int pos);

/**
 * Computes the CAIC.
 * @param sampleSize The size of the sample data, i.e. the number of data points.
 * @return The CAIC.
 */
double computeCaic(int sampleSize);

/**
 * Computes the probabilities for either conditProbs or jointProbs, given a sample data
 * point.
 * @param j                  The j-th sample data point, 0 <= j < sampleData.size().
 * @param normalize          If true, the probabilities for conditProbs will be computed,
 *                           otherwise the probabilities for jointProbs will be computed.
 * @param [in] sampleData    The vector of the sample data.
 * @param [in] logSampleData The vector of the logged sample data.
 */
void computeProbs(int j, bool normalize, Eigen::ArrayXd& sampleData,
    Eigen::ArrayXd& logSampleData);

/**
 * Execute the EM Algorithm. It is the iterative algorithm to find the parameters that
 * maximize the expectation of the complete-data likelihood function.
 * Prerequisite: The method setParams should be executed first to set the initial
 *               parameters for the algorithm.
 * @param tolerance           The tolerance of convergence of the log-likelihood.
 * @param [in] sampleData     The vector of the sample data.
 * @param [in] logSampleData  The vector of the logged sample data.
 * @param meanSampleData      The mean of the sample data.
 * @return True if the algorithm completes successfully, false otherwise. The algorithm
 *         will complete successfully if
 *         1) the log-likelihood converges,
 *         2) the scale parameter is non-zero,
 *         3) the removal of any insignificant branches and rebalancing of the remaining
 *            weights are completed successfully.
 */
bool emAlg(double tolerance, Eigen::ArrayXd& sampleData, Eigen::ArrayXd& logSampleData,
    double meanSampleData);

/**
```

```
     * Sets the parameters of this EMData object to those parameters of another EMData object.
     * Usually done immediately before the EM algorithm is executed. Specifically, the
     * parameters for the previous iteration of this EMData object to those parameters for the
     * current iteration of the other EMData object.
     * @param [in] dat Another EMData object.
     */
    void setParams(EMData* dat);

    /**
     * Copies the parameters of another EMData object to this EMData object. Usually done
     * immediately after the EM algorithm is executed, but only if the parameters of the other
     * EMData object are more optimal than those in this EMData object. Specifically, the
     * parameters for the current iteration of this EMData object to those parameters for the
     * current iteration of the other EMData object.
     * @param [out] dat Another EMData object.
     */
    void copyParams(EMData* dat);

    /**
     * Removes any branches with "insignificant" weights. A branch is removed if the
     * corresponding weight is less than "tolerance divided by M", but only when
     * tolerance is less than 0.01.
     * @param tolerance The tolerance used in determining if a branch is insignificant.
     * @return true if the removal of the branch and rebalancing of the remaining weights are
     *         completed successfully, false otherwise.
     */
    bool filterWeights(double tolerance);

private:
    // The number of Erlang branches.
    int M;
    // The scale parameter values from the previous and current EM iterations respectively.
    double prevTheta, currTheta;
    // The vectors of M weight parameters from the previous and current EM iterations
    // respectively. For this storage container and any others that are used, we do not resize
    // the container to size M; we simply use the first M positions to store the values.
    Eigen::ArrayXd prevAlphas, currAlphas;
    // The vector of M shape parameters.
    Eigen::ArrayXi shapeParams;
    // The value of the log likelihood.
    double logLikelihood;
    // A vector of M elements where the i-th element is log((r[i] - 1)!), and r[i] is the
    // shape parameter of the i-th Erlang branch.
    Eigen::ArrayXd logFactorials;
    // A vector of M elements where the i-th element is the conditional probability
    // q(r[i] | x[j]), r[i] is the shape parameter of the i-th Erlang branch, and x[j] is the
    // j-th sample data point, 0 <= i < M, 0 <= j < sampleData.size(). It is calculated using
    // the parameters of the previous EM iteration.
    Eigen::ArrayXd conditProbs;
    // A vector of M elements where the i-th element is joint probability p(x[j], r[i]), r[i]
    // is the shape parameter of the i-th Erlang branch, and x[j] is the j-th sample data
    // point, 0 <= i < M, 0 <= j < sampleData.size(). It is calculated using the parameters of
    // the current EM iteration.
    Eigen::ArrayXd jointProbs;
};

#endif // EMData_h__
```

The implementation of the EMData class - Emdat.cc

```
// This code uses the free Eigen library available at http://eigen.tuxfamily.org.
#include <EMData.h>

/**
 * "Erase" an element in a storage container. The element is erased in the sense that all
 * elements to the right of this element are shifted one position to the left.
 * @param arr   A reference to a contiguous storage container,
 *              e.g. std::vector, Eigen::ArrayXd, etc.
 * @param pos   the position of the element to be erased
 * @return True if 'pos' is a valid position, false otherwise.
 */
template <typename T> bool erase(T& arr, int pos){
    if (pos < 0 || pos > arr.size() - 1){
        std::cout << "Error, unable to erase position. Out of bounds." << std::endl;
        return false;
    }
    else if (pos != arr.size() - 1){
        std::copy(arr.data() + pos + 1, arr.data() + arr.size(), arr.data() + pos);
    }
    return true;
}

EMData::EMData(){}

EMData::EMData(int numErlangs){
    M = numErlangs;
    currAlphas.setZero(numErlangs);
    prevAlphas.setZero(numErlangs);
    conditProbs.setZero(numErlangs);
    jointProbs.setZero(numErlangs);
    logFactorials.setZero(numErlangs);
    shapeParams.setZero(numErlangs);
    logLikelihood = 0.0;
    prevTheta = 0.0;
    currTheta = 0.0;
}

EMData::~EMData(){}

void EMData::setM(int newM){
    M = newM;
}

int EMData::getM(){
    return M;
}

void EMData::setTheta(bool fromPrevIteration, double theta){
    if (fromPrevIteration){
        prevTheta = theta;
    }
    else currTheta = theta;
```

```
}

double EMData::getTheta(bool fromPrevIteration){
    if (fromPrevIteration){
        return prevTheta;
    }
    else return currTheta;
}

void EMData::setAlphas(bool fromPrevIteration, Eigen::ArrayXd& values){
    if (fromPrevIteration){
        std::copy(values.data(), values.data() + M, prevAlphas.data());
    }
    else{
        std::copy(values.data(), values.data() + M, currAlphas.data());
    }
}

Eigen::ArrayXd& EMData::getAlphas(bool fromPrevIteration){
    if (fromPrevIteration){
        return prevAlphas;
    }

    return currAlphas;
}

bool EMData::rebalanceAlphas(int pos, bool fromPrevIteration){
    Eigen::ArrayXd& alphas = getAlphas(fromPrevIteration);

    if (pos != M - 1){
        alphas(pos + 1) += alphas(pos);
        alphas(pos) = 0.0;
    }
    else if (pos != 0){
        alphas(pos - 1) += alphas(pos);
        alphas(pos) = 0.0;
    }
    else{
        std::cout << "Error, unable to rebalance alphas. M = " << M << std::endl;
        return false;
    }
    return true;
}

Eigen::ArrayXi& EMData::getShapeParams(){
    return shapeParams;
}

void EMData::setLogLikelihood(double newLogLik){
    logLikelihood = newLogLik;
}

double EMData::getLogLikelihood(){
    return logLikelihood;
}

void EMData::computeLogFactorials(){
    double cumulativeLogFactorial = 0.0;
    logFactorials.resize(M);
    int m = 0;
    if (shapeParams(0) - 1 == 0){
        logFactorials(0) = 0.0;
        m++;
    }
    for (int i = 1; i < shapeParams(M-1); i++){
        cumulativeLogFactorial += log((double)i);
        if (i == shapeParams(m) - 1){
            logFactorials(m) = cumulativeLogFactorial;
            m++;
        }
    }
}

Eigen::ArrayXd& EMData::getLogFactorials(){
    return logFactorials;
}

bool EMData::removeBranch(int pos){
    // Erase the element at index 'pos' from each of the storage containers.
    if (!erase(currAlphas, pos)){
        return false;
    }
    if (!erase(prevAlphas, pos)){
        return false;
    }
    if (!erase(shapeParams, pos)){
        return false;
    }
    if (!erase(logFactorials, pos)){
        return false;
    }

    // Find the revised value of M.
    M = M - 1;

    return true;
}

double EMData::computeCaic(int sampleSize){
    double caic = -2*logLikelihood + (2*M + 1)*(log((double) sampleSize) + 1);
    return caic;
}

void EMData::computeProbs(int j, bool normalize, Eigen::ArrayXd& sampleData,
    Eigen::ArrayXd& logSampleData){
        double theta = getTheta(normalize);

        if (normalize){
            // Compute the conditional probabilities.
            conditProbs.head(M) = prevAlphas.head(M)*
                exp((shapeParams.head(M).cast<double>() - 1.0)*(logSampleData(j) - log(theta))
                - logFactorials.head(M) - sampleData(j)/theta - log(theta));
            double normalizingFactor = conditProbs.head(M).sum();
            // Normalize only if the sum of the probabilities are not close to zero.
            if (normalizingFactor > std::numeric_limits<double>::min()){
                conditProbs.head(M) /= normalizingFactor;
            }
        }
        else{
            // Compute the joint probabilities.
            jointProbs.head(M) = currAlphas.head(M)*
                exp((shapeParams.head(M).cast<double>() - 1.0)*(logSampleData(j) - log(theta))
                - logFactorials.head(M) - sampleData(j)/theta - log(theta));
        }
}

bool EMData::emAlg(double tolerance, Eigen::ArrayXd& sampleData,
    Eigen::ArrayXd& logSampleData, double meanSampleData){
        // This vector of 2 elements will represent the values of the expected log-likelihood
        // at the current and previous iterations respectively.
        std::vector<double> expLogLik(2, -std::numeric_limits<double>::infinity());
        bool isOptimal = false;

        // Loop until the log-likelihood cannot be improved.
        while(!isOptimal){
            setLogLikelihood(0.0);
            expLogLik.at(1) = expLogLik.at(0);
            expLogLik.at(0) = 0.0;

            // Find the new optimal set of weights.
            for (int i = 0; i < (int)sampleData.size(); i++){
                computeProbs(i, true, sampleData, logSampleData);
                currAlphas.head(M) += conditProbs.head(M);
            }

            currAlphas /= (double)sampleData.size();

            // Find the new optimal value of theta.
            setTheta(false, getTheta(false) + currAlphas.head(M).matrix().
                dot(shapeParams.head(M).matrix().cast<double>()));

            // Stop and return false if the value of theta is zero or very close to zero.
            if (getTheta(false) < std::numeric_limits<double>::min()){
                std::cout << "Theta is equal to zero! " << std::endl;
                return false;
            }

            setTheta(false, meanSampleData/getTheta(false));

            // Compute the new value of the log-likelihood.
            for (int i = 0; i < (int)sampleData.size(); i++){
                computeProbs(i, false, sampleData, logSampleData);
                computeProbs(i, true, sampleData, logSampleData);
                expLogLik.at(0) += (log(jointProbs.head(M)).matrix()).
                    dot(conditProbs.head(M).matrix());
                setLogLikelihood(getLogLikelihood() + log(jointProbs.head(M).sum()));

                // Stop and return false if the log-likelihood is not finite.
                if (abs(expLogLik.at(0)) > std::numeric_limits<double>::max() ||
                    abs(getLogLikelihood()) > std::numeric_limits<double>::max()){
                    std::cout << "The log-likelihood is not finite for theta = "
                    << getTheta(false) << ". " << std::endl;
                return false;
            }
        }

        // Remove any insignificant weights.
        int prevM = M;

        if (!filterWeights(tolerance)){
            return false;
        }

        // If no weights were removed, check for convergence of the log-likelihood. If
        // convergence has not occurred, continue with the algorithm, otherwise exit the
        // loop.
        if (prevM == M){
            double rel_diff = (expLogLik.at(0) - expLogLik.at(1))/abs(expLogLik.at(0));
            isOptimal = (rel_diff <= tolerance);

            if (!isOptimal){
                setTheta(true, getTheta(false));
                setTheta(false, 0.0);
                std::copy(currAlphas.data(), currAlphas.data() + M, prevAlphas.data());
                std::fill(currAlphas.data(), currAlphas.data() + M, 0.0);
            }
        }
    }
    return true;
}

void EMData::setParams(EMData* dat){
    setM(dat->getM());
    setTheta(true, dat->getTheta(false));
    setTheta(false, 0.0);

    // Resize the containers if they are not at least of size M.
    if (currAlphas.size() < M){
        currAlphas.resize(M);
        prevAlphas.resize(M);
        shapeParams.resize(M);
        logFactorials.resize(M);
        conditProbs.resize(M);
        jointProbs.resize(M);
    }

    setAlphas(true, dat->getAlphas(false));
    std::fill(currAlphas.data(), currAlphas.data() + M, 0.0);
    setLogLikelihood(0.0);
    std::copy(dat->getShapeParams().data(), dat->getShapeParams().data() + M,
        shapeParams.data());
    std::copy(dat->getLogFactorials().data(), dat->getLogFactorials().data() + M,
        logFactorials.data());
}

void EMData::copyParams(EMData* dat){
    setM(dat->getM());
    setTheta(false, dat->getTheta(false));
    // Resize the containers if they are not at least of size M.
    if (currAlphas.size() < M){
        currAlphas.resize(M);
        prevAlphas.resize(M);
        shapeParams.resize(M);
        logFactorials.resize(M);
        conditProbs.resize(M);
        jointProbs.resize(M);
    }

    setAlphas(false, dat->getAlphas(false));
    setLogLikelihood(dat->getLogLikelihood());
    std::copy(dat->getShapeParams().data(), dat->getShapeParams().data() + M,
        shapeParams.data());
    std::copy(dat->getLogFactorials().data(), dat->getLogFactorials().data() + M,
        logFactorials.data());
}

bool EMData::filterWeights(double tolerance){
    if (tolerance < pow(10.0, -2)){
        int i = 0;
        while (i < M){
            if (getAlphas(false)(i) < tolerance/M){
                if (!rebalanceAlphas(i, false)){
                    return false;
                }
                if (!removeBranch(i)){
                    return false;
                }
            }
            else{
                i++;
            }
        }
    }
    return true;
}
```

The main file - emalg.cpp

```
// This code uses the free Eigen library available at http://eigen.tuxfamily.org.
#include <iostream>
#include <fstream>
#include <limits>
#include <iomanip>
#include <vector>
#include <algorithm>
#include <time.h>
#include <Eigen/Dense>
#include <Eigen/StdVector>
#include <omp.h>
#include "EMData.h"
```

```
/**
 * Compares two pointers using the values they point to.
 * @param [in] i A pointer to a double.
 * @param [in] j A pointer to a double.
 * @return true if the value pointed to by 'i' is less than the value pointed to by 'j', false
 *         otherwise.
 */
inline bool compareByPointer(double* i, double* j){
    return (*i<*j);
}

// Rank or order the vector
// If ranked == 0, the order is returned

/**
 * Rank or order vector elements.
 * For a vector, say {1, 4, 5, 2}, the rank will be {1, 3, 4, 2} whereas the order will be
 * {1, 4, 2, 3}.
 * @param [in] ptr      A pointer to a vector of doubles.
 * @param M             The number of vector elements.
 * @param [out] ranks   A vector of integers that will store the ranks/order.
 * @param [out] pRanks  A vector of pointers of type double that will store the ranked/ordered
 *                      pointers.
 * @param ranked        If true, then the vector elements will be ranked, otherwise the
 *                      elements will be ordered.
 */
void rankVectorElements(double* ptr, int M, std::vector<int>* ranks,
    std::vector<double*>* pRanks, bool ranked){
        // Sort the pointers.
        for (int i = 0; i < M; i++){
            pRanks->at(i) = ptr + i;
        }
        double* p_init = pRanks->at(0);
        std::sort(pRanks->data(), pRanks->data() + M, compareByPointer);

        // Compute the ranks/order.
        if (ranked){
            for (int i = 0; i < M; i++){
                ranks->at(pRanks->at(i) - p_init) = i;
            }
        }
        else{
            for (int i = 0; i < M; i++){
                ranks->at(i) = pRanks->at(i) - p_init;
            }
        }
}

/**
 * Process the data.
 * Sort the sample data, remove any zero values, and compute the log of the sample data.
 * @param [in,out] sampleData   A reference to the vector of the sample data.
 * @param [out] logSampleData   A reference to a vector that will store the logged sample
 *                              data.
 */
void processData(Eigen::ArrayXd& sampleData, Eigen::ArrayXd& logSampleData){
    // Sort the sample data.
    int sampleSize = (int)sampleData.size();
    std::sort(sampleData.data(), sampleData.data() + sampleData.size());

    // Remove any zero values.
    int start = 0;
    while (!sampleData(start) && start < sampleSize){
        start++;
    }
    std::cout << start/sampleSize << " zero values were removed from the data.\n"
        << std::endl;

    if (!start){
        std::copy(sampleData.data() + start, sampleData.data() + sampleData.size(),
            sampleData.data());
        sampleData.conservativeResize(sampleData.size() - start);
    }
    sampleSize = (int)sampleData.size();

    // Compute the log of the sample data.
    logSampleData.resize(sampleSize);
    std::transform(sampleData.data(), sampleData.data() + sampleSize, logSampleData.data(),
        (double (*)(double))log);
}

/**
 * Initialize the parameters of the Erlang mixture, given an initial value of theta.
 * @param [in] sampleData   The vector of the sample data.
 * @param initTheta         The initial value of theta.
 * @return A pointer to an EMData object with the initial parameter estimates.
 */
EMData* initParams(Eigen::ArrayXd& sampleData, double initTheta){
    // Set initial value of M equal to the number of data points.
    int M = (int)sampleData.size();
    double meanSampleData = sampleData.mean();
    double theta = initTheta;
    int tempM = M; // Used to store the new value of M, after zero weights are removed.
    std::vector<double> tempInitAlphas(M);
    std::vector<int> tempInitShapeParams(M);
    int numRefined = 2;

    // Find initial values of the weights and "refine" the value of theta once.
    for (int m = 0; m < numRefined; m++){
        tempM = 0;
        int i = 1; // Used to iterate through the Erlang branches (1 <= i <= M).
        int j = 0; // Used to iterate through the sample data (0 <= j <= sampleSize - 1).
        int freq = 0; // Used to store the frequency of data points in an interval.

        // Compute the weights for i = 1 to M - 1.
        while (j < sampleData.size() && i <= sampleData.size()){
            if (sampleData(j) < i*theta){
                j++;
                freq++;
            }
            else{
                if (freq){
                    tempInitShapeParams.at(tempM) = i;
                    tempInitAlphas.at(tempM) = (double)freq/(double)sampleData.size();
                    tempM++;
                }
                i++;
                freq = 0;
            }
        }

        // Compute the weight for i = M.
        if (freq){
            tempInitShapeParams.at(tempM) = i;
            tempInitAlphas.at(tempM) = (double)freq/(double)sampleData.size();
            tempM++;
        }

        // Find the revised value of M.
        M = tempM;

        Eigen::Map<Eigen::VectorXi> mapTempInitShapeParams(&tempInitShapeParams.at(0), M);
        Eigen::Map<Eigen::VectorXd> mapTempInitAlphas(&tempInitAlphas.at(0), M);

        // Refine the value of theta.
        if (m != numRefined - 1){
            theta = meanSampleData/
                (mapTempInitShapeParams.cast<double>().dot(mapTempInitAlphas));
        }
    }

    // Store the initial parameter estimates in an EMData object.
    EMData* initEMData = new EMData(M);
    std::copy(tempInitShapeParams.begin(), tempInitShapeParams.begin() + M,
        initEMData->getShapeParams().data());
    std::copy(tempInitAlphas.begin(), tempInitAlphas.begin() + M,
        initEMData->getAlphas(false).data());
    initEMData->setTheta(false, theta);

    // Compute the log factorials based on the initial estimates.
    initEMData->computeLogFactorials();

    return initEMData;
}

/**
 * Computes an initial value of theta.
 * From the possible values of theta, there are two criteria for selecting an initial value:
 * 1) The CAIC should be relatively low, and
 * 2) The value of the difference between the 5th moment of the fitted model and that of the
 *    data should be relatively small and positive.
 * The value of theta that performs "best" in these two categories is chosen. Additionally,
   * information on other values of theta are output to the file "info.txt".
 * @param [in] sampleData    A vector of the sample data.
 * @param [in] logSampleData A vector of the logged sample data.
 * @return An initial value of theta.
 */
double computeInitTheta(Eigen::ArrayXd& sampleData, Eigen::ArrayXd& logSampleData){
    // Initialize the vectors of the selection criteria.
    int sampleSize = sampleData.size();
    std::vector<double> caics(sampleSize, std::numeric_limits<double>::infinity());
    std::vector<double> diffMoments(sampleSize, std::numeric_limits<double>::infinity());

    std::ofstream info;
    info.open("info.txt", std::ofstream::out);
    info << "theta \t M \t log_likelihood \t CAIC \t model_moment_minus_data_moment"
        << std::endl;

    // Compute the fifth moment of the data.
    double data_mom5 = sampleData.pow(5).sum()/sampleSize;

#pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < sampleSize; i++){
        // Estimate the initial parameters given a value of theta
        double theta = sampleData.mean()/(double)(i+1);
        EMData* initEMData = initParams(sampleData, theta);

        // Consider this value of theta only if the number of parameters is less than the
        // size of the sample data (to prevent over-fitting).
        if (initEMData->getM()*2 + 1 < sampleSize){
            // Execute the EM algorithm once to find an "optimal" set of parameters.
            initEMData->setParams(initEMData);
            bool isWithoutError = initEMData->emAlg(0.1, sampleData, logSampleData,
                sampleData.mean());

            // Continue only if the EM algorithm completes successfully.
            if (isWithoutError){
                initEMData->copyParams(initEMData);
                theta = initEMData->getTheta(false);
                int M = initEMData->getM();

                // Compute the fifth moment of the fitted model.
                double first = initEMData->getAlphas(false).matrix().
                    dot(initEMData->getShapeParams().cast<double>().matrix())*theta;
                double second = initEMData->getAlphas(false).head(M).matrix().
                    dot(initEMData->getShapeParams().head(M).cast<double>().pow(2).matrix())*
                    pow(theta, 2.0);
                double third = initEMData->getAlphas(false).head(M).matrix().
                    dot(initEMData->getShapeParams().head(M).cast<double>().pow(3).matrix())*
                    pow(theta, 3.0);
                double fourth = initEMData->getAlphas(false).head(M).matrix().
                    dot(initEMData->getShapeParams().head(M).cast<double>().pow(4).matrix())*
                    pow(theta, 4.0);
                double fifth = initEMData->getAlphas(false).head(M).matrix().
                    dot(initEMData->getShapeParams().head(M).cast<double>().pow(5).matrix())*
                    pow(theta, 5.0);
                double mod_mom5 = fifth + 10*fourth*theta + 35*third*pow(theta, 2.0) +
                    50*second*pow(theta, 3.0) + 24*first*pow(theta, 4.0);

                if (mod_mom5 > data_mom5){
                    caics.at(i) = initEMData->computeCaic(sampleSize);
                    diffMoments.at(i) = mod_mom5 - data_mom5;
                }

                // Store the information in the file "info.txt".
                #pragma omp critical
                {
                    info << std::setprecision(20) << sampleData.mean()/(double)(i+1) << "\t"
                        << initEMData->getM() << "\t"<< initEMData->getLogLikelihood()
                        << "\t" << initEMData->computeCaic(sampleSize) << "\t"
                        << mod_mom5 - data_mom5 << std::endl;
                }
            }
        }

        delete initEMData;
    }

    // Rank the model criteria.
    std::vector<double*> pCaics(caics.size());
    std::vector<int> ranksCaics(caics.size());
    std::vector<double*> pDiffMoments(diffMoments.size());
    std::vector<int> ranksDiffMoments(diffMoments.size());
    rankVectorElements(caics.data(), caics.size(), &ranksCaics, &pCaics, true);
    rankVectorElements(diffMoments.data(), diffMoments.size(), &ranksDiffMoments,
        &pDiffMoments, true);

    // Reuse the caics vector to find the average rank.
    for (int i = 0; i < (int)caics.size(); i++){
        caics.at(i) = (ranksCaics.at(i) + ranksDiffMoments.at(i))/2.0;
    }

    // Find the index of the theta with the best/lowest rank.
    int bestRankIndex = std::min_element(caics.begin(), caics.end()) - caics.begin();

    info.close();
    return (sampleData.mean()/(double)(bestRankIndex + 1));
}

/**
 * The stepwise algorithm for selecting the most optimal parameters.
 * @param [in] EMDataVector      The vector of EMData objects.
 * @param [in,out] finalEMData   The final EMData object which will store the most optimal
 *                               parameters.
 * @param [in] initEMData        The initial EMData object which stores the initial parameters.
 * @param tolerance              The tolerance of convergence of the likelihood function.
 * @param [in] sampleData        The vector of the sample data.
 * @param [in] logSampleData     The vector of the logged sample data.
 * @return True if the algorithm completes successfully, false otherwise. The algorithm
 *         will complete successfully if
 *         1) the log-likelihood converges,
 *         2) the scale parameter is non-zero, and
 *         3) the removal of any insignificant branches and rebalancing of the remaining
 *            weights are completed successfully.
 */
bool stepwise(std::vector<EMData>* EMDataVector, EMData* finalEMData, EMData* initEMData,
    double tolerance, Eigen::ArrayXd& sampleData, Eigen::ArrayXd& logSampleData){
        // Set the number of threads to the number of processors on the user's computer.
        int numThreads = omp_get_num_procs();
        omp_set_num_threads(numThreads);

        // Optimize the initial set of parameters and store in the finalEMData object.
        int sampleSize = sampleData.size();
        double meanSampleData = sampleData.sum()/(double)sampleSize;
```

```
finalEMData->setParams(initEMData);
finalEMData->emAlg(tolerance, sampleData, logSampleData, meanSampleData);
finalEMData->copyParams(finalEMData);
bool isOptimal = false;

// The first major step of the stepwise algorithm.
// This loop removes the smallest weight one at a time until the CAIC cannot be
// improved.
while (!isOptimal){
    // Find the index of the smallest weight.
    int minIndex = 0;
    for (int i = 1; i < finalEMData->getM(); i++){
        if (finalEMData->getAlphas(false)(i) <
            finalEMData->getAlphas(false)(minIndex)){
            minIndex = i;
        }
    }

    // Use the first element of EMDataVector to perform the computations.
    EMDataVector->at(0).setParams(finalEMData);

    // Rebalance the weights.
    if (!EMDataVector->at(0).rebalanceAlphas(minIndex, true)){
        return false;
    }

    // Remove the branch with the smallest weight.
    if (!EMDataVector->at(0).removeBranch(minIndex)){
        return false;
    }

    // Execute the EM algorithm on the mixture without the smallest weight.
    if (!EMDataVector->at(0).emAlg(tolerance, sampleData, logSampleData,
        meanSampleData)){
        return false;
    }

    if (finalEMData->computeCaic(sampleSize) <
        EMDataVector->at(0).computeCaic(sampleSize)){
        // If the removal of the smallest weight does not improve the CAIC, then we
        // proceed to the next major step of the algorithm. Before proceeding, we
        // resize the EMDataVector to the number of Erlangs in the finalEMData vector.
        isOptimal = true;
        for (int m = 1; m < finalEMData->getM(); m++){
            EMDataVector->push_back(EMData(finalEMData->getM()));
        }
    }
    else{
        // If the removal of the smallest weight improves the CAIC, continue the loop.
        finalEMData->copyParams(&EMDataVector->at(0));
    }
}

// The second major step of the stepwise algorithm.
// Find the "smallest" weight in the set of weights that can be removed so that the
// CAIC is improved.
```

```
    // Create vectors to rank the weights and store the ranks.
    std::vector<int> ranks;
    std::vector<double*> pRanks;
    ranks.resize(finalEMData->getM());
    pRanks.resize(finalEMData->getM());

    // The branch that, once removed, improves the CAIC will be referred to as an
    // insignificant branch.
    bool foundInsignificantBranch = false;
    // The number of branches checked until an insignificant branch was found.
    int numBranchesChecked = 0;
    isOptimal = false;

    while (!isOptimal){
        // Rank the weights and store in the vector "rank".
        rankVectorElements(finalEMData->getAlphas(false).data(), finalEMData->getM(),
            &ranks, &pRanks, false);
        foundInsignificantBranch = false;
        numBranchesChecked = 0;
        bool isWithoutError = true;

        // Perform parallel execution of the EM algorithm until the removal of a branch
        // improves the CAIC.
#pragma omp parallel for shared(foundInsignificantBranch, isWithoutError) schedule(dynamic)
        for (int m = 0; m < finalEMData->getM(); m++){
            // The EM algorithm is not executed if a previous branch was found to be
            // insignificant, or if there was an error in any previous executions of the
            // EM algorithm.
            if (!foundInsignificantBranch && isWithoutError){
                numBranchesChecked++;

                // Run the EM algorithm when the branch with the m-th smallest weight is
                // removed.
                EMDataVector->at(ranks.at(m)).setParams(finalEMData);

                // Rebalance the weights.
                if (!EMDataVector->at(ranks.at(m)).rebalanceAlphas(ranks.at(m), true)){
                    isWithoutError = false;
                }

                // Remove the branch with the m-th smallest weight.
                if (!EMDataVector->at(ranks.at(m)).removeBranch(ranks.at(m))){
                    isWithoutError = false;
                }

                // Execute the EM algorithm on the mixture without the m-th smallest
                // weight.
                if (!EMDataVector->at(ranks.at(m)).emAlg(tolerance, sampleData,
                    logSampleData, meanSampleData)){
                    isWithoutError = false;
                }

                // Check if the branch is the first insignificant branch.
                if (!foundInsignificantBranch && isWithoutError){
                    foundInsignificantBranch = finalEMData->computeCaic(sampleSize) >
                        EMDataVector->at(ranks.at(m)).computeCaic(sampleSize);
                }
                }
            }

            // Return false if any errors occur.
            if (!isWithoutError){
                return false;
            }

            if (foundInsignificantBranch){
                // If an insignificant branch is found, store the new optimal parameters in
                // the finalEMData object and continue the loop.
                int minCaicIndex = ranks.at(0);
                double minCaicValue = EMDataVector->at(minCaicIndex).computeCaic(sampleSize);

                // Since the algorithm was executed in parallel, there may be multiple
                // insignificant branches. Find the most insignificant branch.
                for (int m = 1; m < numBranchesChecked; m++){
                    double temp = EMDataVector->at(ranks.at(m)).computeCaic(sampleSize);
                    if (temp < minCaicValue){
                        minCaicValue = temp;
                        minCaicIndex = ranks.at(m);
                    }
                }

                // Store the most optimal parameters in the finalEMData object.
                finalEMData->copyParams(&EMDataVector->at(minCaicIndex));
            }
            else{
                // If no insignificant branch is found, then the finalEMData object contains
                // the most optimal set of parameters. Exit the loop.
                isOptimal = true;
            }
        }

        return true;
}

/**
 * Main method for this application. Most of the code in this method is used to communicate to
 * the user. As such, comments are only included for the most important code.
 * @return The exit code for the process - 0 for success, otherwise an error code.
 */
int main(){
    // Read the sample data.
    std::ifstream data;
    data.open("data.txt", std::ofstream::in);
    std::vector<double> tempSampleData;

    if (data.is_open()){
        double temp = 0.0;

        while (!data.eof()){
            data >> temp;
            tempSampleData.push_back(temp);
        }
    }
    else{
```

```
    std::cout << "Data file could not be opened!" << std::endl;
    return 0;
}

Eigen::ArrayXd sampleData, logSampleData;
double tolerance = pow(10.0, −3);
sampleData.resize((int)tempSampleData.size());
std::copy(tempSampleData.begin(), tempSampleData.end(), sampleData.data());

std::cout << "\nPlease specify the tolerance of convergence (e.g. 0.0001): ";
std::cin >> tolerance;
std::cout << "\n" << std::endl;

// Get the number of threads on the user's computer.
int numThreads = omp_get_num_procs();
omp_set_num_threads(numThreads);
std::cout << "This program will use " << numThreads << " threads. \n" << std::endl;

// Process the sample data.
processData(sampleData, logSampleData);

time_t start, end;
double bestInitTheta = 0.0;
int shouldFindInitTheta = 1;

std::cout << "To provide an initial value, type 0. " << std::endl;
std::cout << "Otherwise, type 1 if you have no preference." << std::endl;
std::cout << "Note that this process my take a long time to complete if the" << std::endl;
std::cout << "sample size is very large" << std::endl;
std::cout << "Enter 1 or 0 here: " << std::endl;
std::cin >> shouldFindInitTheta;
std::cout << std::endl;

if (shouldFindInitTheta){
    time(&start);
    // If the user has no preference, find an initial value of theta based on the
    // previously mentioned criteria.
    bestInitTheta = computeInitTheta(sampleData, logSampleData);

    time(&end);
    std::cout << std::setprecision(20) << "It took " << difftime(end, start)
        << " seconds to compute the initial estimates.\n" << std::endl;
}
else{
    std::cout << "Enter theta here: " << std::endl;
    std::cin >> bestInitTheta;
    std::cout << std::endl;
}

std::ofstream results;
results.open("results.txt", std::ofstream::out);
int isFirstRun = 1;
int quit = 0;
int hasRunOnce = 0;

while (!quit){
    if (!isFirstRun){
        std::cout << "If the results are satisfactory, please enter −1 to quit."
            << std::endl;
        std::cout << "Otherwise, enter an initial value for the scale parameter, theta."
            << std::endl;
        if (!hasRunOnce && shouldFindInitTheta){
            std::cout << "The file info.txt may aid in your initial choice of theta."
                << std::endl;
            std::cout << "(The file is better viewed using a spreadsheet program)."
                << std::endl;
            hasRunOnce = 1;
        }
        std::cout << "\n Enter theta (or −1) here: " << std::endl;
        std::cin >> bestInitTheta;
        std::cout << std::endl;
        if (bestInitTheta == −1){
            quit = 1;
            return 0;
        }
        std::cout << "\nPlease specify the tolerance of convergence (e.g. 0.0001): ";
        std::cin >> tolerance;
        std::cout << "\n" << std::endl;
    }
    else{
        isFirstRun = 0;
    }

    time(&start);

    // Construct the initial EMData object based on the initial value of theta.
    EMData* initEMData = initParams(sampleData, bestInitTheta);

    int M = initEMData−>getM();
    std::cout << "The number of Erlangs that will be used to fit this data is "
        << M << std::endl;
    std::cout << "The initial scale parameter is " << bestInitTheta << std::endl;

    // A vector of EMData objects to be used for parallel processing.
    std::vector<EMData> EMDataVector;
    EMDataVector.push_back(EMData(M));

    // Construct the final EMData object to be used for storing the most optimal
    // parameters.
    EMData* finalEMData = new EMData(M);

    // Perform the stepwise algorithm for finding the most optimal parameters
    bool isWithoutError = stepwise(&EMDataVector, finalEMData, initEMData, tolerance,
        sampleData, logSampleData);

    // Output results to the file "results.txt" if the algorithm completed without error.
    if (isWithoutError){
        M = finalEMData−>getM();

        time(&end);

        std::cout << std::setprecision(20) << "It took " << difftime(end, start)
            << " seconds to complete the algorithm.\n" << std::endl;
            if (results.is_open()){
                for (int i = 0; i < M; i++){
                    results << std::setprecision(20) << finalEMData->getShapeParams()(i)
                        << "\t" << finalEMData->getAlphas(false)(i) << std::endl;
                }
                results << std::setprecision(20) << finalEMData->getTheta(false) << "\t"
                    << finalEMData->getLogLikelihood() << "\n" << std::endl;

                std::cout << "Results have been written." << std::endl;
                std::cout << "All but the last number in the first column ";
                std::cout << "are the shape parameters." << std::endl;
                std::cout << "All but the last number in the second column ";
                std::cout << "are the weights (alpha)." << std::endl;
                std::cout << "In the last line is the scale parameter, theta ";
                std::cout << "followed by the log-likelihood.\n" << std::endl;
            }
            else{
                std::cout << "Results could not be written! \n" << std::endl;
            }
        }
        else{
            std::cout << "The algorithm cannot be executed for theta = " << bestInitTheta
                << "." << std::endl;
        }

        // Clean-up.
        delete initEMData;
        delete finalEMData;
    }

    system("PAUSE");
    return 0;
}
```

References

[1] Mark J. Ablowitz and Athanassios S. Fokas. *Complex variables: introduction and applications*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, second edition, 2003.

[2] Hansjörg Albrecher and Stefan Thonhauser. Optimality results for dividend problems in insurance. *Rev. R. Acad. Cienc. Exactas Fís. Nat. Ser. A Mat. RACSAM*, 103(2):295–320, 2009.

[3] Søren Asmussen and Hansjörg Albrecher. *Ruin probabilities*. Advanced Series on Statistical Science & Applied Probability, 14. World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, second edition, 2010.

[4] Florin Avram, Zbigniew Palmowski, and Martijn R. Pistorius. On the optimal dividend problem for a spectrally negative Lévy process. *Ann. Appl. Probab.*, 17(1):156–180, 2007.

[5] Florin Avram, Zbigniew Palmowski, and Martijn R. Pistorius. On the optimal dividend problem for a spectrally negative Lévy process. *Ann. Appl. Probab.*, 17(1):156–180, 2007.

[6] Pablo Azcue and Nora Muler. Optimal reinsurance and dividend distribution policies in the cramÉr-lundberg model. *Mathematical Finance*, 15(2):261–308, 2005.

[7] Jean Bertoin. Exponential decay and ergodicity of completely asymmetric Lévy processes in a finite interval. *Ann. Appl. Probab.*, 7(1):156–169, 1997.

[8] J Chen, K H Lundberg, D E Davison, and D S Bernstein. The final value theorem revisited - infinite limits and irrational functions. *Control Systems Magazine IEEE*, 27(3):97–99, 2007.

[9] Eric C. K. Cheung. On optimal dividend strategies in the compound Poisson model by Elias S. W. Shiu and Hans U. Gerber, April 2006 [mr2328638]. *N. Am. Actuar. J.*, 11(1):158–161, 2007.

[10] Bruno De Finetti. Su un'impostazione alternativa della teoria colletiva del rischio. *Transactions of the XVth International Congress of Actuaries, vol. 2.*, pages 433–443, 1957.

[11] Olivier Deprez and Hansjörg Albrecher. "Optimal dividends: analysis with Brownian motion," Hans U. Gerber and Elias S. W. Shiu, January 2004. *N. Am. Actuar. J.*, 8(2):111–115, 2004.

[12] J. D. Esary and A. W. Marshall. Shock models and wear processes. *The Annals of Probability*, 1(4):pp. 627–649, 1973.

[13] Edward Furman and Zinoviy Landsman. Risk capital decomposition for a multivariate dependent gamma portfolio. *Insurance Math. Econom.*, 37(3):635–649, 2005.

[14] T.W. Gamelin. *Complex analysis*. Springer Verlag, 2001.

[15] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[16] A. Kuznetsov, A. E. Kyprianou, and V. Rivero. The theory of scale functions for spectrally negative Le vy processes. *ArXiv e-prints*, April 2011.

[17] A. E. Kyprianou, V. Rivero, and R. Song. Convexity and smoothness of scale functions and de finetti's control problem, 2008.

[18] Andreas E. Kyprianou. *Introductory lectures on fluctuations of Lévy processes with applications*. Universitext. Springer-Verlag, Berlin, 2006.

[19] Andreas E. Kyprianou and Zbigniew Palmowski. A martingale review of some fluctuation theory for spectrally negative Lévy processes. In *Séminaire de Probabilités XXXVIII*, volume 1857 of *Lecture Notes in Math.*, pages 16–29. Springer, Berlin, 2005.

[20] S. Lee and S. Lin. Modelling insurance losses and calculating risk measures via a mixture of erlangs. 2009.

[21] S.C.K. Lee and X.S. Lin. Modeling and evaluating insurance losses via mixtures of erlang distributions. *North American Actuarial Journal*, 2010.

[22] Alan L. Lewis and Ernesto Mordecki. Wiener-hopf factorization for l'evy processes having negative jumps with rational transforms, 2005.

[23] R. L. Loeffen. On optimality of the barrier strategy in de Finetti's dividend problem for spectrally negative Lévy processes. *Ann. Appl. Probab.*, 18(5):1669–1680, 2008.

[24] Ronnie L. Loeffen and Jean-François Renaud. De finetti's optimal dividends problem with an affine penalty function at ruin. *Insurance: Mathematics and Economics*, 46(1):98 – 108, 2010. Gerber-Shiu Functions / Longevity risk and capital markets.

[25] Geoffrey J. McLachlan and Thriyambakam Krishnan. *The EM algorithm and extensions*. Wiley Series in Probability and Statistics. Wiley-Interscience [John Wiley & Sons], Hoboken, NJ, second edition, 2008.

[26] J. Paulsen. Optimal dividend payouts for diffusions with solvency constraints. *Finance and Stochastics*, 7(4):457–473, 2003.

[27] Jean-François Renaud and Xiaowen Zhou. Distribution of the present value of dividend payments in a Lévy risk model. *J. Appl. Probab.*, 44(2):420–427, 2007.

[28] B. A. Surya. Evaluating scale functions of spectrally negative Lévy processes. *J. Appl. Probab.*, 45(1):135–149, 2008.

[29] Axel Thümmler, Peter Buchholz, and Miklós Telek. A novel approach for phase-type fitting with the em algorithm. *IEEE Transactions on Dependable and Secure Computing*, 3:245–258, 2006.

[30] Henk C. Tijms. *Stochastic models*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons Ltd., Chichester, 1994. An algorithmic approach.

[31] David Vernon Widder. *The Laplace Transform*. Princeton Mathematical Series, v. 6. Princeton University Press, Princeton, N. J., 1941.