Optimaliserte R-beregninger i skadeforsikring med vekt på feilreservering

av

FINN HARALD OPSJØN

MASTEROPPGAVE

for graden

Master i Modellering og Dataanalyse

(Master of Science)

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Mai 2013

Faculty of Mathematics and Natural Sciences
University of Oslo

Sammendrag

Beregninger innenfor skadeforsikring tar ofte utgangspunkt i modeller. Derfor er det interessant å analysere modellfeil, der virkningene av slike feil i stigende grad vil bli mer aktuelt i skadeforsikring. Slike analyser krever ofte nøstede simuleringer, som betyr Monte Carlo-simuleringer som avhenger av andre Monte Carlo-simuleringer. Dette fører til at tiden på simuleringene øker drastisk, noe som skaper et behov for raske simuleringer. Vi vil derfor optimalisere kjøretiden på simuleringer, der vi i denne oppgaven bruker den statistiske plattformen R, og anvendelsesområdet er reserveberegninger i skadeforsikring.

Vi introduserer en egenprodusert algoritme for reserveberegninger, og optimaliserer ved å vektorisere, parallellisere og kombinere C++- og R-kode, slik at vi kan beregne reserver opp mot 240 ganger raskere enn standardmetoden. Optimaliseringen gjør det mulig å øke presisjon og hastighet på simulering av reserver, og kan bidra til å gjøre beregninger i forsikringsselskaper mer effektive. Raskere simuleringer fører også til at vi kan gjøre brede analyser av problemer som krever nøstede simuleringer. Et slikt problem er feilreservering i skadeforsikring, og denne oppgaven inneholder en analyse av dette når vi velger feil modell for erstatningskravene.

Vi deler reserveringsfeil inn i to typer feil, systematisk og tilfeldig feil. Den systematiske feilen skyldes valg av feil modell, og den tilfeldige feilen skyldes estimeringsfeil av parametere innenfor valgt modell. Vi studerer to situasjoner, der underliggende korrekt modell er enten log-normal eller pareto, mens valgt modell er gamma i begge tilfellene. Vi undersøker hvordan reserveringsfeil avhenger av parametere i underliggende korrekt modell, antall observasjoner og skadebegrensning per polise.

Forord

Jeg vil takke min veileder, Erik Bølviken, for å gi meg en god og interessant oppgave, og takk for Fortran-kode som finner vekter og abskisser ved Gaussisk-Legendre kvadratur.

Takk til folket på lesesal B802 for godt arbeidsmiljø, og takk til Rebecca for hjelp med asymptotisk teori.

Jeg vil også takke min familie for god støtte, spesielt Solveig og Hartvig, og takk til Philip for teknisk hjelp i C++. Til slutt vil jeg takke min kjæreste, Maria.

Innhold

1 Innledning **1**

2 Skadeforsikringsteori **3**
2.1 Tapsmodeller . . . 3
2.1.1 Gamma-modellen . . . 3
2.1.2 Log-normal-modellen . . . 5
2.1.3 Pareto-modellen . . . 6
2.2 Reserve . . . 7
2.2.1 Definisjon . . . 7
2.2.2 Skadebegrensning . . . 8
2.3 Reserveringsfeil . . . 10
2.3.1 Definisjon . . . 10
2.3.2 Riktig modell . . . 10
2.3.3 Feil modell . . . 11
2.3.4 Systematisk og tilfeldig feil . . . 13
2.4 Sannsynlighet for valg av modell . . . 15

3 Optimalisering i R **17**
3.1 Innledning . . . 17
3.2 R-pakker . . . 17
3.2.1 Rbenchmark . . . 18
3.2.2 Compiler . . . 19
3.2.3 Rcpp/Inline . . . 19
3.2.4 Foreach/DoParallel . . . 20
3.2.5 Rmpi/Snow . . . 21
3.3 Optimalisert programmering . . . 22
3.3.1 Enkle eksempler . . . 22
3.3.2 Vektorisering og parallellisering . . . 26
3.4 Optimalisert reserveberegning . . . 28
3.4.1 Ny algoritme . . . 28
3.4.2 Hastighetstester . . . 29
3.5 Oppsummering . . . 31

4 Feilreservering **33**
4.1 Innledning 33
4.2 Gamma-modellen mot Log-normal-modellen 36
4.2.1 Systematisk feil 37
4.2.2 Relativ systematisk feil og total feil 38
4.2.3 Parameterfeil 43
4.2.4 Sannsynlighet for å velge riktig modell 45
4.3 Gamma-modellen mot Pareto-modellen 46
4.3.1 Systematisk feil 47
4.3.2 Relativ systematisk feil og total feil 48
4.3.3 Parameterfeil 53
4.3.4 Sannsynlighet for å velge riktig modell 54
4.4 Oppsummering 55

5 Konklusjon **57**

6 Videre arbeid **59**

A Appendix **61**
A.1 Figurer for relativ systematisk feil og total feil 61
A.1.1 Gamma-modellen mot Log-normal-modellen 61
A.1.2 Gamma-modellen mot Pareto-modellen 65
A.2 Tabeller 68
A.2.1 Optimalt parametersett under gamma-modellen 68
A.2.2 Optimal 99 %-reserve (ψ_0) under gamma-modellen 69
A.2.3 99 %-reserve (ψ) under korrekt modell 71
A.3 Programkode 72
A.3.1 Eksempel 1 i seksjon 3.3.1 72
A.3.2 Eksempel 2 i seksjon 3.3.1 73
A.3.3 Eksempel 3 i seksjon 3.3.1 74
A.3.4 Test 1 i seksjon 3.4.2 75
A.3.5 Test 2 i seksjon 3.4.2 77
A.3.6 Egenprodusert R-pakke: reserveRcpp 79
A.3.7 Test som viser at Algoritme 1 og 2 gjør samme beregninger 80
A.3.8 Finne vekter og abskisser for (2.30) 81
A.3.9 Finne optimalt parametersett under gamma-modellen 83
A.3.10 Beregne systematisk feil 85
A.3.11 Beregne relativ systematisk feil/total feil/parameterfeil 87
A.3.12 Sannsynlighet for å velge riktig modell 91
A.3.13 Figurer 93

Bibliografi **100**

Kapittel 1

Innledning

R er en gratis programvare og programmeringsplattform for statistiske beregninger. Tidligere har R vært for treg til å kjøre omfattende Monte Carlo-simuleringer (MC-simuleringer), men dette er i ferd med å endre seg. I dag er R i sterk utvikling og er blitt et seriøst alternativ for å kjøre tunge MC-simuleringer. Dette gjør R til et godt analyseverktøy i skadeforsikring, fordi det er praktisk å gjøre statistisk analyse av historiske data og MC-simuleringer i en og samme plattform.

Innenfor skadeforsikring vil man i stigende grad interessere seg for virkningen av modellfeil på størrelser og mål som skadeforsikringsberegninger bygger på. Dette gir ofte nøstede simuleringer, som vil si å gjøre MC-simuleringer som avhenger av andre MC-simuleringer. Nøstede simuleringer gjør at simuleringstiden øker kraftig, og dette virker negativt på presisjonen i beregningene våre fordi antall simuleringer må begrenses. Desto flere simuleringer vi kjører, desto bedre presisjon på beregningene får vi. Dette motiverer oss til å øke hastigheten på simuleringene, slik at vi kan øke presisjonen. Derfor vil vi i denne oppgaven optimalisere simuleringer i R, der det sentrale anvendelsesområdet er reserveberegninger i skadeforsikring.

Ved å optimere reserveberegninger gjør vi det mulig å studere utvidede analyser av problemstillinger som krever nøstede simuleringer. En slik aktuell problemstilling i skadeforsikring er feilreservering, og i denne oppgaven vil vi studere dette nærmere når vi velger feil modell for erstatningskravene. Det er godt begrunnet innenfor skadeforsikring at antall skader ofte er Poisson-fordelte, mens valg av modell for erstatningskravene varierer oftere fra situasjon til situasjon. Derfor vil vi fokusere på hvordan valg av feil modell for erstatningskravene påvirker reserveringsfeil, der reserveringsfeil er absoluttverdien av differansen mellom estimert reserve og korrekt reserve. Siden reserver beregnes ved MC-simuleringer, vil det oppstå MC-feil, men vi vil kjøre mange nok simuleringer slik at MC-feilen blir neglisjerbar.

I denne oppgaven deler vi reserveringsfeil inn i to typer feil, systematisk og tilfeldig feil. Systematisk feil er en konstant og skyldes valg av feil modell. Den tilfeldige feilen skyldes estimeringsfeil av parametere under valgt modell. Vi vil undersøke hvordan reserveringsfeil avhenger av parametere i underliggende korrekt modell, antall observasjoner og skadebegrensning per polise. Vi vil også se på sannsynligheten for å velge riktig modell.

I kapittel 2 beskrives skadeforsikringsteori, deriblant tapsmodeller, reserver, reserveringsfeil for både riktig og feil modell, samt sannsynlighet for å velge riktig modell. Kapittelet inneholder også algoritmer som beskriver hvordan de ulike beregningene i analysen av feilreservering gjøres.

I kapittel 3 introduserer vi R-utvidelser (R-pakker), som er nyttige å bruke ved optimalisering i R, samt hvordan vi optimaliserer i R. Siden statistikere som bruker R ofte kan lite avansert programmering, vil det beskrives hvordan vi optimaliserer relativt enkelt. Videre optimaliserer vi beregningen av reserver ved å vektorisere og parallellisere, samt kombinere C++-kode med R-kode. Vi introduserer også en egenutviklet algoritme for beregning av reserver.

I kapittel 4 studerer vi numeriske resultater for reserveringsfeil når vi velger feil modell for ertatningskravene i skadeforsikring. Vi studerer to situasjoner, der underliggende korrekt modell er enten log-normal- eller pareto-modellen, mens valgt modell er gamma-modellen i begge tilfellene.

Nye metoder som introduseres i denne oppgaven er en vektorisert algoritme for beregning av reserver i skadeforsikring, samt beskrivelser av hvordan reserveberegninger optimaliseres i R ved å integrere inn C++-kode og ved å parallellisere. Det er også en ny metode å dele opp reserveringsfeil i skadeforsikring inn i systematisk og tilfeldig feil, og det er ikke kjent at noen tidligere har analysert reserveringsfeil når valgt modell for erstatningskravene er gamma-modellen, mens korrekt modell er log-normal- eller pareto-modellen.

Kapittel 2

Skadeforsikringsteori

2.1 Tapsmodeller

I denne oppgaven vil vi bruke tre forskjellige tapsmodeller som er mye brukt i skadeforsikring. Det som er typisk for disse modellene er at vi som regel har mange små og noen store skadeutbetalinger (f. eks. brannskade, bilskade, naturskade osv.). Dermed får vi fordelinger som ofte har tunge haler i slike situasjoner. Denne seksjonen inneholder de viktigste resultatene innenfor hver modell som vi trenger for beregningene i denne oppgaven. Mer spesifikt vil det si tetthet, forventning, standardavvik, log-likelihood-funksjon og maximum likelihood-estimatorer (ML-estimatorer) for hver modell. Dette er velkjente uttrykk i statistikk.

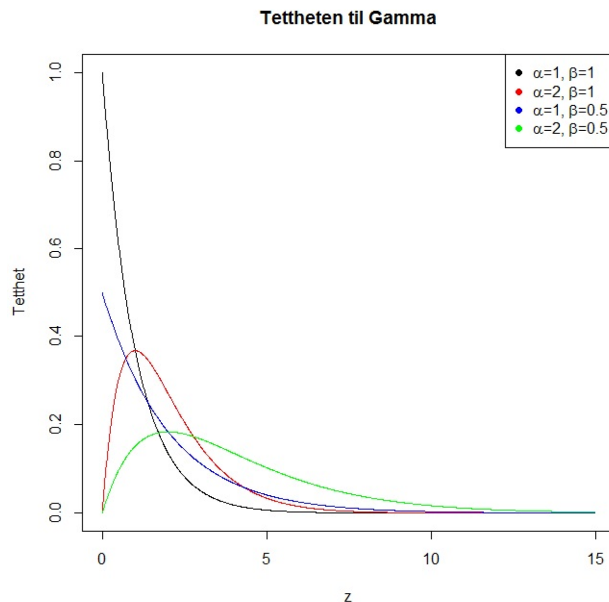
2.1.1 Gamma-modellen

Gamma-modellen er utfyllende beskrevet i [5].

Tetthetsfunksjonen til gamma-modellen er definert følgende

$$f(z;\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} z^{\alpha-1} \exp(-\beta z)\ ;\ z \in (0,\infty), \alpha > 0, \beta > 0 \tag{2.1}$$

der $\Gamma(\alpha) = \int_0^\infty z^{\alpha-1} \exp(-z)dz$, α er *shape*-parameter og β er *rate*-parameter. Når α=1, er gamma-modellen lik eksponensial-modellen med parameter β.



Figur 2.1: Tettheten til gamma-modellen for ulike parametere.

Forventning og standardavvik under gamma-modellen, uttrykt ved α og β, er følgende

$$\mathrm{E[Z]} = \frac{\alpha}{\beta} \tag{2.2}$$

$$\mathrm{sd[Z]} = \frac{\sqrt{\alpha}}{\beta} \tag{2.3}$$

R-kommando for å trekke tilfeldige gamma-fordelte variable (Z) er: **rgamma**(antall,α,β).

Log-likelihood-funksjonen til gamma-modellen

$$\log\ L(\alpha,\beta) = n\alpha\ \log(\beta) - n\ \log(\Gamma(\alpha)) + (\alpha-1)\sum_{i=1}^{n}\log(z_i) - \beta\sum_{i=1}^{n} z_i \tag{2.4}$$

der n er antall observasjoner.

ML-estimatene for α og β

$$\hat{\beta} = \frac{n\hat{\alpha}}{\sum_{i=1}^{n} z_i} \tag{2.5}$$

Dersom vi deriverer log-likelihood-funksjonen og løser for $\hat{\alpha}$, får vi ingen eksplisitt løsning for $\hat{\alpha}$. Derfor må $\hat{\alpha}$ finnes ved numerisk optimering av (2.4), der uttrykket for $\hat{\beta}$ er satt inn.

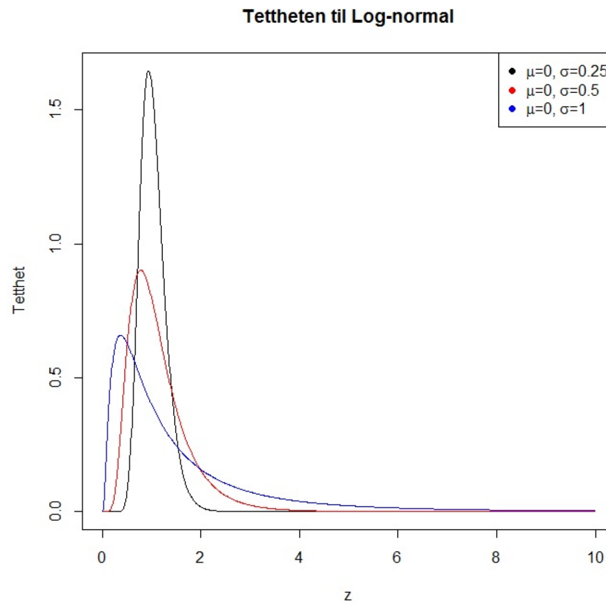
2.1.2 Log-normal-modellen

Log-normal-modellen er utfyllende beskrevet i [5].

Tetthetsfunksjonen til log-normal-modellen er definert følgende

$$f(z;\mu,\sigma) = \frac{1}{z\sigma\sqrt{2\pi}}\exp(-\frac{(\log z-\mu)^2}{2\sigma^2})\ ;\ z\in(0,\infty), \mu\in(-\infty,\infty), \sigma>0 \tag{2.6}$$

der μ og σ er parametere som henholdsvis beskriver forventning og standardavvik til log z $\sim$ normalfordelt.



Figur 2.2: Tettheten til log-normal-modellen for $\mu = 0$ og ulike σ.

Forventning og standardavvik til log-normal-modellen, uttrykt ved μ og σ, er følgende

$$\text{E[Z]} = \exp(\mu + \frac{\sigma^2}{2}) \tag{2.7}$$

$$\text{sd[Z]} = \sqrt{\exp(2\mu+\sigma^2)\exp(\sigma^2-1)} \tag{2.8}$$

R-kommando for å trekke tilfeldige log-normal-fordelte variable (Z) er: **rlnorm**(antall,μ,σ).

Log-likelihood-funksjonen til log-normal-modellen

$$\log\ L(\mu,\sigma) = -\log\sum_{i=1}^{n} z_i - \frac{n}{2}\log 2\pi - n\log\sigma - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(\log z_i-\mu)^2 \tag{2.9}$$

der n er antall observasjoner.

ML-estimatene for μ og σ

$$\hat{\mu} = \frac{\sum_{i=1}^{n} \log z_i}{n} \tag{2.10}$$

$$\hat{\sigma} = \sqrt{\frac{\sum_{i=1}^{n} (\log z_i - \hat{\mu})^2}{n}} \tag{2.11}$$

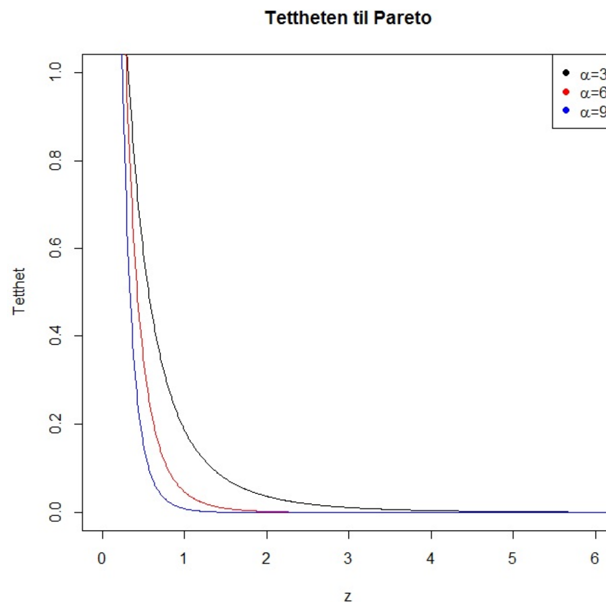
2.1.3 Pareto-modellen

Pareto-modellen er utfyllende beskrevet i [14].

Tetthetsfunksjonen til pareto-modellen er definert følgende

$$f(z;\alpha) = \frac{\alpha}{(1+z)^{\alpha+1}} \ ; \ z \in (0,\infty), \alpha > 0 \tag{2.12}$$

der α er *shape*-parameter. Vi får pareto-modellen ved å substituere x = log(1+z) i f(x), dersom f(x) er tetthetsfunksjonen til eksponensialfordelingen.



Figur 2.3: Tettheten til pareto-modellen for ulike α.

Forventning og standardavvik under pareto-modellen, uttrykt ved α, er følgende

$$\text{E[Z]} = \frac{1}{\alpha - 1} \ ; \ \alpha > 1 \tag{2.13}$$

$$\mathrm{sd}[\mathrm{Z}] = \sqrt{\frac{\alpha}{(\alpha-2)(\alpha-1)^2}}\ ;\ \alpha > 2 \tag{2.14}$$

R-kommando for å trekke tilfeldige pareto-fordelte variable (Z) er:
exp(**rexp**(antall,α))**-1**.

Log-likelihood-funksjonen til pareto-modellen

$$\log\ L(\alpha) = n\log\alpha - (\alpha+1)\sum_{i=1}^{n}\log(1+z_i) \tag{2.15}$$

der n er antall observasjoner.

ML-estimat for α

$$\hat{\alpha} = \frac{n}{\sum_{i=1}^{n}\log(1+z_i)} \tag{2.16}$$

2.2 Reserve

I denne seksjonen introduseres begrepene reserve og skadebegrensning, samt standardalgoritmen for beregning av reserve og algoritme for beregning av estimert reserve. Utfyllende bakgrunn for reserveberegninger er beskrevet i [3].

2.2.1 Definisjon

En reserve er et beløp et forsikringsselskap/foretak må avsette for å møte sine forpliktelser. Forsikringsselskapet må ha penger på konto til enhver tid som er nok til å dekke utbetalingene sine.

Matematisk kan dette skrives slik

$$Pr(X > q_\epsilon) = 1 - \epsilon \tag{2.17}$$

der X er samlede utbetalinger og q_ϵ er $(1-\epsilon)$-reserven, der ϵ uttrykkes i prosent/persentil. Dette betyr at sannynligheten for at samlede utbetalinger skal overstige $(1-\epsilon)$-reserven er ϵ %. Altså vil en 99 %-reserve bety at det er 1 % sannsynlighet for at samlede utbetalinger skal overstige 99 %-reserven.

I skadeforsikring er det helt sentralt å beregne reserver, og disse beregningene gjøres ved hjelp av simuleringer. For å forstå hvordan en reserve beregnes, må vi forstå hvordan samlede skadeutbetalinger beregnes. Vi antar antall skader (N) er Poisson-fordelt med parameter λ, $N \sim \textit{Poisson}(\lambda)$. Skadestørrelsene ($Z$) antas å tilhøre en bestemt modell, $Z \sim \textit{modell}$.

Samlede skadeutbetalinger (X) er summen av alle skadeutbetalinger (Z), og er definert følgende

$$X = \sum_{i=1}^{N} Z_i \tag{2.18}$$

For å finne reserven numerisk, regner vi ut samlede skadeutbetalinger (X) m antall ganger, sorterer X-vektoren i stigende rekkefølge og plukker ut element m*(1-ϵ), der ϵ bestemmer nivået på reserven. Videre vil vi definere skadebegrensninger, før vi ser hvordan reserveberegningene skal implementeres på datamaskinen ved hjelp av algoritmer.

2.2.2 Skadebegrensning

I skadeforsikring er det vanlig med skadebegrensninger for å sikre seg mot store krav. Denne skadebegrensningen kan være på både polisenivå eller porteføljenivå, men i denne oppgaven vil vi kun se på polisenivå. En skadebegrensningskontrakt på polisenivå går ut på at et forsikringsselskap (cedent) betaler maksimalt en viss størrelse b per krav (Z). Denne kontrakten inngås mellom cedent og re-assurandør, der re-assurandøren betaler det som måtte overstige skadebegrensningen b og cedenten betaler re-assurandør for å ta på seg denne risikoen. En slik kontrakt vil matematisk se slik ut

Polisenivå

$$Z = \begin{cases} Z & ; 0 \leq Z \leq b \\ b & ; \quad Z > b \end{cases} \tag{2.19}$$

Porteføljenivå

$$X = \begin{cases} X & ; 0 \leq X \leq b \\ b & ; \quad X > b \end{cases} \tag{2.20}$$

Reserver beregnes numerisk og standardmetoden for beregning av reserve kaller vi her Algoritme 1. Senere i oppgaven i seksjon 3.4.1 vil Algoritme 2 introduseres, som vil være en vektorisert utgave av Algoritme 1.

Algoritme 1

Beregning av $(1-\epsilon)$-reserve med skadebegrensning på polisenivå.

0. Input: m, b, θ, λ.
1. Repeteres m ganger:
  2. Trekk $\hat{N}^* \sim$ Poisson(λ)
  3. Trekk $\hat{Z}^*$ fra modell med parameter θ (antall trekk: $\hat{N}^*$)
  4. $\hat{Z}^*$ = min($\hat{Z}^*$, b)
  5. $\hat{X}^*$ = sum($\hat{Z}^*$)
6. Sortér $\hat{X}^*$ stigende
7. $(1-\epsilon)$-reserven = $(1-\epsilon)$-persentilen i $\hat{X}^*$

Her er m antall simuleringer, b er skadebegrensning per polise, θ er parametersett under aktuell modell for skadeutbetalingene og λ er Poisson-parameteren som beskriver forventet antall skader.

Linje 0 består av parametere som må defineres på forhånd, der Poisson-parameteren λ angir forventet antall skader i et scenario (f. eks. år). Linje 1 blir som regel programmert som starten på en for-løkke, som går m ganger. I linje 2 trekker vi tallet på antall skader i et scenario, før vi i linje 3 trekker skadeutbetalingene fra en bestemt modell. I linje 4 slår skadebegrensningen inn, før vi i linje 5 summerer alle skadene i en vektor $\hat{X}^*$. Deretter sorterer vi denne vektoren stigende i linje 6, før vi finner reserven til slutt. Linje 1-5 blir vanligvis programmert som en for-løkke.

Når et forsikringsselskap beregner reserver i skadeforsikring, må de først estimere et passende parametersett som er basert på historiske data. Dette parametersettet vil i denne oppgaven estimeres ved å finne verdien på parameterene som maksimerer likelihooden (ML) til valgt modell. Deretter brukes dette estimerte parametersettet ($\hat{\theta}$) som input i reserveberegningene (Algoritme 1). Dersom vi lar Ψ være en funksjon for reserve og $\hat{\psi}$ er estimert reserve, har vi at $\hat{\psi} = \Psi(\hat{\theta})$.

Algoritme for beregning av estimert reserve ($\hat{\psi}$)

0. Input: z.
1. Historiske data: z = $z_1, ..., z_n$
2. Finn $\hat{\theta}$ ved ML basert på $z_1, ..., z_n$
3. Finn $\hat{\psi}$ ved å kjøre Algoritme 1 med $\theta = \hat{\theta}$

2.3 Reserveringsfeil

I denne seksjonen definerer vi reserveringsfeil, og ser på hvordan reserveringsfeil avhenger av om modellen for skadestørrelsene er riktig eller ikke. Vi ser derfor på teori for parametere som fungerer som input i beregningen av reserver. Til slutt definerer vi uttrykk for ulike feilkilder til reserveringsfeil under feil modell, der dette er sentralt i denne oppgaven.

2.3.1 Definisjon

La $\hat{\psi}$ og ψ være henholdsvis estimert og korrekt reserve. Reserveringsfeilen er absoluttverdien av differansen mellom $\hat{\psi}$ og ψ

$$Reserveringsfeil = |\hat{\psi} - \psi| \tag{2.21}$$

Estimert reserve avhenger av ML-estimerte parametere som er basert på underliggende datasett. Videre vil vi derfor se på hvordan disse ML-estimatene avhenger av størrelsen på datasett (antall observasjoner), både når vi velger riktig og feil modell for skadestørrelsene.

2.3.2 Riktig modell

Her vil vi se på resultater som gjelder når vi har riktig modell for skadestørrelsene i skadeforsikring. Vi vil vise til velkjente resultater innenfor den asymptotiske teorien, der utfyllende bakgrunn for disse resultatene er beskrevet i [5]. Den asymptotiske teorien forteller oss hvordan ML-estimater av parametere oppfører seg når datamengden (n) øker. Dette vil vi bruke til å vise at reserveringsfeilen går mot 0 når antall observasjoner går mot uendelig ($n \to \infty$).

Asymptotisk teori I

Vi definerer først scorefunksjonen $s(\theta)$ følgende

$$s(\theta) = \frac{\partial}{\partial \theta} l(\theta) \tag{2.22}$$

der $l(\theta) = l(\theta; X)$ er log-likelihood-funksjonen til fordelingen $f(x;\theta)$. Dersom vi setter scorefunksjonen lik 0 og løser for θ, får vi ML-estimator $\hat{\theta}$, som er den parameteren/parametersettet som maksimerer likelihooden. En viktig egenskap ved scorefunksjonen er at hvis X har en sannsynlighetsfordeling $f(X;\theta)$, er $\mathrm{E}[\mathrm{s}(\theta;X)] = 0$. Scorefunksjonen vil i et fler-parametertilfelle være en vektor med dimensjon p, der p er dimensjonen på θ (antall parametere).

Videre definerer vi Fisher-informasjonen $I(\theta)$ slik

$$I(\theta) = V[s(\theta)] = -E\left[\frac{\partial^2}{\partial \theta^2} l(\theta)\right] \quad \text{(en parameter)} \tag{2.23}$$

$$I(\theta) = I_{ij}(\theta) = Cov[s_i(\theta;X), s_j(\theta;X)] = -E\left[\frac{\partial^2}{\partial \theta_i \partial \theta_j} l(\theta)\right] \quad \text{(flere parametere)} \tag{2.24}$$

der i = 1, ..., p og j = 1, ..., p. Antall parametere er p. Fisher-informasjonen, også kalt forventet informasjon, er et mål på hvor mye informasjon vi har om parameteren(e) θ. I et fler-parametertilfelle vil $I(\theta)$ være en p×p-matrise.

Dersom vi har tilfeldige variable x_1, ..., x_n, som tilhører en fordeling $f(x;\theta)$, og antar at mengden av mulige verdier ikke avhenger av θ, vil en ML-estimator $\hat{\theta}$ være konsistent ($\Pr(\hat{\theta} \to \theta) = 1$ når $n \to \infty$) og tilnærmet normalfordelt med forventning θ og varians $\frac{1}{n}I^{-1}(\theta)$, når $n \to \infty$. Det vil si at

$$\sqrt{n}(\hat{\theta} - \theta) \sim N(0, I^{-1}(\theta)) \tag{2.25}$$

når $n \to \infty$. Dette gjelder både når θ er en eller flere dimensjoner.

Dersom en funksjon h er kontinuerlig, vil $h(\hat{\theta}) \to h(\theta)$ hvis $\hat{\theta} \to \theta$ når $n \to \infty$. Derfor har vi at estimert reserve, $\hat{\psi} = \Psi(\hat{\theta})$, går mot korrekt reserve, $\psi = \Psi(\theta)$, når $n \to \infty$, der Ψ er en funksjon for reserve. Det betyr at reserveringsfeilen går mot 0 når $n \to \infty$.

2.3.3 Feil modell

Når vi velger feil modell for skadeutbetalingene, er den asymptotiske teorien litt annerledes. I tillegg vil reserveringsfeilen bestå av enda en ekstra feilkilde i forhold til tilfellet med riktig modell. Dermed vil reserveringsfeilen avhenge av to faktorer. For å gå nærmere inn på dette trenger vi noen flere resultater, der resultatene for den asymptotiske teorien under feil modell er utfyllende beskrevet i [10].

Asymptotisk teori II

La x_1, ..., x_n være tilfeldige variable, som tilhører en fordeling g(x), og la ML-estimat $\hat{\theta}$ tilhøre en fordeling $f_\theta \neq g$. I et en-parameter-tilfelle gjelder følgende

$$\sqrt{n}(\hat{\theta} - \theta) \sim N(0, I_g^{-1}(\theta)) \tag{2.26}$$

når $n \to \infty$. Her er $I_g(\theta) = V_g[\frac{\partial}{\partial\theta}l(\theta)] = -E_g[\frac{\partial^2}{\partial\theta^2}l(\theta)]$ og E_g/V_g er forventning/varians under modell g. ML-estimatoren $\hat{\theta}$ er konsistent også i dette tilfellet, og den asymptotiske forventningen til $\sqrt{n}(\hat{\theta} - \theta)$ er lik som tilfellet under Asymptotisk teori I, mens vi ser at variansen $I_g^{-1}(\theta)$ er forskjellig.

For fler-parametertilfellet har Huber vist at når $n \to \infty$, er $\hat{\theta}$ konsistent og $\sqrt{n}(\hat{\theta}-\theta)$ tilnærmet normalfordelt med forventning 0 og kovariansmatrise $\Lambda^{-1}I_g(\theta)(\Lambda^T)^{-1}$ (dimensjon p × p)

$$\sqrt{n}(\hat{\theta} - \theta) \sim N(0, \Lambda^{-1}I_g(\theta)(\Lambda^T)^{-1}) \tag{2.27}$$

Her er $\Lambda = \lambda_{ij}(\theta)$ er en inverterbar matrise og er definert følgende

$$\lambda_{ij}(\theta) = \int_{-\infty}^{\infty} \left(\frac{\partial^2}{\partial\theta_i\partial\theta_j}l(\theta)\right) g(x)dx = E_g\left[\frac{\partial^2}{\partial\theta_i\partial\theta_j}l(\theta)\right] \tag{2.28}$$

der i = 1, ..., p og j = 1, ..., p. Dimensjonen til θ er p.

Siden E[$\sqrt{n}(\hat{\theta} - \theta)$]=0 når $n \to \infty$, betyr det at estimeringsfeilen av ML-estimatorene $\hat{\theta}$ går mot 0 når antall observasjoner går mot uendelig, selv om vi velger feil modell.

Kullback-Leibler-avstanden

I sannsynlighetsteori er Kullback-Leibler-avstanden et ikke-symmetrisk mål på avstanden mellom to sannsynlighetfordelinger. Vi kaller disse to fordelingene f og g, der g er fordelingen vi vil måle avstanden til. Kullback-Leibler-avstanden måler dermed avstanden fra f til g, der denne avstanden er ikke-negativ. I denne oppgaven er fordelingene vi jobber med kontinuerlige, og Kullback-Leibler-avstanden er definert følgende [9]

$$D_{KL}(f|g) = \int_{-\infty}^{\infty} g(x) \log\left(\frac{g(x)}{f(x)}\right) dx \tag{2.29}$$

En god tilnærming til Kullback-Leibler avstanden kan gjøres ved hjelp av Gaussisk-Legendre kvadratur. Denne metoden krever at funksjonene vi vil integrere over er glatte funksjoner. Metoden går ut på at vi har et gitt antall (N) sett med vekter (w_i) og abskisser ($x_i \sim g(x)$), der vi tilnærmer integralet (2.29) som en vektet sum [8]

$$D_{KL}(f|g) \approx \sum_{i=1}^{N} w_i \left[g(x_i) \log\left(\frac{g(x_i)}{f(x_i)}\right)\right] \tag{2.30}$$

Denne tilnærmingen konvergerer raskt når N går mot uendelig, men N trenger sjeldent å være veldig stor for å få en god tilnærming. Den raske konvergensen skyldes at vi benytter oss av vekter, og det konvergerer raskere enn en tilnærming uten vekter.

Optimalt parametersett (θ_0)

I skadeforsikring vil vi finne best mulig tilnærming til dataene gjennom en modell og tilhørende parameterestimater. Vi kaller det optimale parametersettet for θ_0. Dette parametersettet (tilhørende modell f) finner vi ved å minimere Kullback-Leibler-avstanden (2.30) med hensyn på parameter(e) under f. Dette vil være ekvivalent med å minimere

$$\sum_{i=1}^{N} -w_i [g(x_i) \log(f(x_i))] \tag{2.31}$$

fordi følgende uttrykk

$$\sum_{i=1}^{N} w_i [g(x_i) \log(g(x_i))] \tag{2.32}$$

minimert med hensyn på parameter(e) under f, er 0.

I denne oppgaven er valgt modell f_θ, mens korrekt modell er g. Det optimale parametersettet, θ_0, er den verdi som ML-estimat $\hat{\theta}$ konvergerer mot når $n \to \infty$. Dette er avgjørende for å beskrive de to feilkildene til reserveringsfeil ved valg av feil modell.

2.3.4 Systematisk og tilfeldig feil

La ψ være reserve basert på korrekt underliggende modell g. Videre lar vi f_θ være en modell ulik g, som avhenger av parameter θ, og θ_0 er den verdi som minimerer Kullback-Leibler-avstanden mellom f_θ og g. Dermed vil ψ_0 være reserve basert på underliggende modell f_{θ_0}. Både ψ_0 og ψ vil være faste størrelser, slik at også differansen mellom dem vil være en fast størrelse. Denne differansen beskriver reserveringsfeilen ved valg av feil modell f, mens den korrekte modellen er g. I denne oppgaven kaller vi absoluttverdien av differansen mellom ψ_0 og ψ for **systematisk feil**.

Reserveringsfeil avhenger ikke bare av valg av feil modell, men også av estimeringsfeil av parametere. I skadeforsikring er scenarioet typisk at dataene $X_1, ..., X_n$ stammer fra modell g(x), slik at korrekt modell er g, mens vi tror at modell f er den riktige modellen. Parametersettet $\hat{\theta}$ estimeres basert på datasettet X ved ML under modell f. Basert på modell $f_{\hat{\theta}}$ beregnes reserve $\hat{\psi}$. Differansen mellom $\hat{\psi}$ og ψ_0 avhenger av størrelsen n på datasettet, og den asymptotiske teorien gir at denne differansen vil gå mot 0 når n øker. Denne differansen beskriver reserveringsfeilen som følge av estimeringsfeil, og absoluttverdien av denne differansen kaller vi **tilfeldig feil**.

Når $n \to \infty$ vil $\hat{\theta} \to \theta_0$, slik at $\hat{\psi} \to \psi_0$ og tilfeldig feil vil gå mot 0. Det er verdt å nevne at dersom antall observasjoner er få, gjelder ikke den asymptotiske teorien og vi får forventningsskjeve parameterestimatorer. Dette fører til en ekstra feilkilde, som i denne oppgaven inngår i den tilfeldige feilen, for ikke å gjøre feilanalysen for kompleks. I denne oppgaven vil vi stort sett studere tilfeller med mange observasjoner, slik at den asymptotiske teorien gjelder.

Den totale reserveringsfeilen er absoluttverdien av differansen mellom $\hat{\psi}$ og ψ, og denne differansen kaller vi **total feil**. Total feil definerer vi følgende

$$\textit{Total feil} = |\hat{\psi} - \psi| = |\hat{\psi} - \psi_0 + \psi_0 - \psi| \leq |\hat{\psi} - \psi_0| + |\psi_0 - \psi| \tag{2.33}$$

Total feil inneholder to feilkomponenter, som beskriver tilfeldig og systematisk feil, der vi matematisk definerer tilfeldig og systematisk feil slik

$$\textit{Tilfeldig feil} = |\hat{\psi} - \psi_0| \tag{2.34}$$

$$\textit{Systematisk feil} = |\psi_0 - \psi| \tag{2.35}$$

Dersom vi har riktig modell og $n \nrightarrow \infty$, vil den systematiske feilen forsvinne, og vi står kun igjen med tilfeldig feil i (2.33). Om vi har feil modell og $n \to \infty$, vil tilfeldig feil forsvinne, og vi står igjen med kun systematisk feil i (2.33).

I denne oppgaven vil vi se på hvor stor den totale feilen er, samtidig som vi studerer forholdet mellom den systematiske og tilfeldige feilen for å se hvilken feil som er dominerende i ulike tilfeller. Vi måler forholdet mellom systematisk og tilfeldig feil ved å beregne **relativ systematisk feil**, som vi definerer følgende

$$\textit{Relativ systematisk feil} = \frac{|\psi_0 - \psi|}{|\psi_0 - \psi| + |\hat{\psi} - \psi_0|} \tag{2.36}$$

Vi bruker ikke *mean squared error* (MSE) som mål på tilfeldig og total feil, fordi det her ville gjort sammenligningen mer unøyaktig av de ulike feiltypene. Dersom vi f. eks. skulle målt hvor stor andel systematisk feil utgjør av MSE for total feil, kan vi i noen tilfeller få at denne andelen utgjør mer enn 100 %.

Ved simulering av reserveringsfeil må vi benytte oss av **nøstede simuleringer**. Det vil si at vi gjør MC-simuleringer som er avhengige av andre MC-simuleringer. Her er beregningen av hver reserve gjort ved m antall simuleringer, mens vi gjør m_b antall simuleringer av reserveringsfeil. Hver simulering m_b av reserveringsfeil inneholder dermed avstanden mellom simulerte reserver.

Tilfeldig og total feil avhenger av $\hat{\psi}$, slik at vi trenger et estimert parametersett $\hat{\theta}$ som input for hver $\hat{\psi}$ som beregnes. Dette parametersettet ($\hat{\theta}$), som tilhører antatt riktig modell, finner vi ved ML basert på et gitt antall data trukket fra korrekt modell. Total feil består dermed av m_b simuleringer av $\hat{\psi}$, som måles mot ψ. Det samme gjelder for tilfeldig feil, bortsett fra at $\hat{\psi}$ måles mot ψ_0. Det betyr at vi beregner *forventningen* til relativ systematisk feil og total feil ved MC-simuleringer. For m_b antall MC-simuleringer kalkuleres (forventet) relativ systematisk feil og total feil henholdsvis slik

$$\frac{|\psi_0 - \psi|}{|\psi_0 - \psi| + E[|\hat{\psi} - \psi_0|]} = \frac{|\psi_0 - \psi|}{|\psi_0 - \psi| + \frac{1}{m_b}\sum_{i=1}^{m_b} |\hat{\psi}_i - \psi_0|} \tag{2.37}$$

$$E[|\hat{\psi} - \psi|] = \frac{1}{m_b}\sum_{i=1}^{m_b} |\hat{\psi}_i - \psi| \tag{2.38}$$

Her er $|\psi_0 - \psi|$ systematisk feil (konstant), $E[|\hat{\psi} - \psi_0|]$ er forventet tilfeldig feil og $E[|\hat{\psi} - \psi|]$ er forventet total feil.

Algoritme for beregning av relativ systematisk feil og total feil

0. Input: m_b, m, n, θ_K, ψ_0^*, ψ^*.
1. Repeteres m_b ganger:
  2. Trekk data $z_1^*, ..., z_n^*$ fra korrekt modell med parameter θ_K
  3. Estimér parametersett $\hat{\theta}^*$ under valgt modell ved ML basert på $z_1^*, ..., z_n^*$
  4. Beregn $\hat{\psi}^*$ ved Algoritme 1 basert på $\hat{\theta}^*$ (m simuleringer)
5. Relativ systematisk feil = $\frac{|\psi_0^* - \psi^*|}{|\psi_0^* - \psi^*| + \frac{1}{m_b}\sum_{i=1}^{m_b} |\hat{\psi}_i^* - \psi_0^*|}$
6. Total feil = $\frac{1}{m_b}\sum_{i=1}^{m_b} |\hat{\psi}_i^* - \psi^*|$

Her er m_b antall simuleringer av reserveringsfeil, m er antall simuleringer per reserve, n er antall observasjoner, θ_K er parametersettet som tilhører korrekt modell, mens ψ_0^*, ψ^* og $\hat{\psi}^*$ er simulerte verdier av reservene ψ_0, ψ og $\hat{\psi}$.

2.4 Sannsynlighet for valg av modell

I skadeforsikring er det vanlig å velge modell for skadeutbetalingene etter hvilken modell som gir best tilpasning ved QQ-plott. Det vil si å velge den modellen som har mest mulig like persentiler sammenlignet med underliggende datasett. Når vi vil gjøre MC-simuleringer av dette, må vi ha et mål på hvilken modell som gir best tilpasning. Dette målet kaller vi Q, og er definert følgende

$$Q = \sum_{i=1}^{n} |p_i - z_{(i)}| \tag{2.39}$$

der $p_i = F^{-1}\left(\frac{i-0.5}{n}\right)$ og i = 1, ...,n [4, 6]. F er kumulativfunksjonen til aktuell modell, mens $z_{(1)}$, ..., $z_{(n)}$ er underliggende data sortert stigende. Dermed vil den modellen med lavest verdi av Q være den foretrukne modellen, fordi det gir minst differanse mellom persentilene i aktuell modell og underliggende datasett.

Vi kan ved MC-simuleringer se på sannsynligheten for å velge ulike modeller, dersom vi trekker nye datasett per simulering og summerer opp antall ganger aktuell modell gir lavest Q-verdi. Denne summen dividert på antall simuleringer er sannsynligheten for å velge aktuell modell. I denne oppgaven vil underliggende data trekkes fra en gitt modell, der både størrelse på datasett og parameterene som tilhører gitt modell, vil variere. Estimeringen av parametere under aktuelle modeller tilpasses ved ML.

Algoritme for beregning av sannsynlighet for valg av modell

0. Input: m_b, n, θ_K.
1. Repeteres m_b ganger:
  2. Trekk data $z_1^*, ..., z_n^*$ fra korrekt modell med parameter θ_K
  3. Estimér parametersett $\hat{\theta}^*$ under aktuelle modeller ved ML basert på $z_1^*, ..., z_n^*$
  4. Sortér $z_{(1)}^* \leq ... \leq z_{(n)}^*$
  5. Beregn $p_1^* \leq ... \leq p_n^*$ basert på $\hat{\theta}^*$ og aktuelle modeller
  6. Beregn $Q^* = \sum_{i=1}^{n} |p_i^* - z_{(i)}^*|$ for hver aktuelle modell
7. Sannsynlighet for valg av aktuell modell = Summen av antall ganger aktuell modell gir lavest Q-verdi dividert på m_b

Her er m_b antall simuleringer, n er antall observasjoner og θ_K er parametersett under korrekt modell.

Kapittel 3

Optimalisering i R

3.1 Innledning

R er en programmeringsplattform som utvikles av frivillige programmerere over hele verden. Innenfor statistikk er R et av de beste programmeringsspråkene, og det er ikke uvanlig å importere R-kode når det skal gjøres statistiske beregninger i andre programmeringsspråk. Simuleringshastigheten til R derimot, har vært altfor treg til å være et programmeringsspråk for beregningsintensive kjøringer. Dette er dog i ferd med å endre seg, og R er i senere tid blitt et seriøst alternativ til å kjøre tunge MC-simuleringer. Det skyldes god utvikling av programvaren R og nye R-pakker. Derfor vil vi i dette kapittelet vise hvordan vi kan bruke R-pakker til å øke simuleringshastigheten betraktelig.

Vi vil først introdusere noen R-utvidelser (R-pakker), som er hensiktsmessige å bruke når vi vil optimalisere kode i R. Deretter vil vi se på noen enkle eksempler på hvordan disse R-pakkene kan anvendes for å oppnå raskere simuleringer, samt hva det er viktig å tenke på når vi effektiviserer R-kode. Siden de fleste R-programmerere er statistikere, som kan lite avansert programmering, vil det beskrives relativt enkelt hvordan vi optimaliserer R-kode. Kapittelet avsluttes med optimalisering av reserveberegninger i skadeforsikring, der vi optimaliserer ved å vektorisere og parallellisere, samt kombinerer C++-kode med R-kode. Det vil også introduseres en ny algoritme for beregning av reserve, som vil være en vektorisert utgave av Algoritme 1.

3.2 R-pakker

En R-pakke er en utvidelse av R, som blir laget av frivillige utviklere. Hver utvidelse inneholder stort sett funksjoner som gjør det lettere å programmere eller gjøre beregninger i R. For å bruke disse R-pakkene må de installeres først. Dette trenger vi kun å gjøre én gang. Dersom vi skal benytte oss av pakkene i R, må vi laste inn pakkene for hver ny sesjon i R. Vi installerer R-pakker med kommandoen **install.packages("pakkenavn")**, og laster inn R-pakker med kommandoen **library(pakkenavn)**.

3.2.1 Rbenchmark

Først introduserer vi *rbenchmark*-pakken [11]. Denne pakken gjør det enkelt å teste forskjellig kode mot hverandre og måle tidsdifferanser på ulike kodesnutter ved bruk av funksjonen **benchmark**. Funksjonen gjør ikke R-kode noe raskere, men testing av kode blir veldig enkelt og vi får god oversikt. Her vil vi vise hvordan vi bruker *benchmark*-funksjonen på en enkel måte.

La oss ta for oss tre forskjellige kodesnutter vi vil sammenligne tiden på. I det følgende eksempelet trekker vi 100 ganger en million tilfeldige variable fra fordelingene uniform (R-kommando: **runif**), gamma (R-kommando: **rgamma**) og log-normal (R-kommando: **rlnorm**). For å teste dette eksempelet ved bruk av benchmark, skriver vi følgende R-kode

```
benchmark("Uniform" = { runif(1e6) },
          "Gamma" = { rgamma(1e6,1) },
          "Log-normal" = { rlnorm(1e6) },
          columns=c("test",
            "replications",
            "elapsed",
            "relative"
            ),
          order="relative",
          replications=100
          )
```

De tre første linjene består av hver kodesnutt vi vil teste med tilhørende navn skrevet innenfor anførselstegnene og selve R-koden innenfor klammeparantesene. Under *columns* skriver vi en vektor med den utskriften vi ønsker. *Test* angir navn på testen, *replications* angir antall repetisjoner av kodesnuttene, *elapsed* angir total tid i sekunder på kjøringene og *relative* er tiden på hver kjøring dividert på den beste tiden. I dette eksempelet vil vi sortere kjøringene etter relativ tid målt i forhold til den raskeste kjøringen. Derfor skriver vi at *order* skal være lik *relative*. Til slutt skriver vi antall (her: 100) kjøringer vi vil ha av de forskjellige kodesnuttene under *replications*. Dette gir følgende R-utskrift

```
        test replications elapsed relative
1    Uniform          100    2.65    1.000
3 Log-normal          100   16.02    6.045
2      Gamma          100   17.09    6.449
```

Vi ser at vi får en pen utskrift av kjøringene, der nummereringen av kjøringene står helt til venstre. Videre følger testnavn, antall repetisjoner, tid på kjøringene i sekunder og relativ tid målt mot beste tid. Dette illustrerer hvor god oversikt vi får ved å bruke *rbenchmark*-pakken til å teste tid på kjøringer mot hverandre. Ellers viser dette eksempelet at det tar mer enn 6 ganger så lang tid å trekke tilfeldige log-normal- og gammavariable enn uniforme variable i R.

3.2.2 Compiler

R er et interpreterende programmeringsspråk, også kjent som høynivå-språk. Det vil si at hver linje med kode som kjøres, oversettes fra høynivå-kode til maskinkode. Dette tar lenger tid å kjøre enn om koden er kompilert ned til maskinkode fra starten av, slik kompileringsspråkene C, C++ og Fortran er eksempler på.

Siden R er et interpreterende programmeringsspråk tar det tid for datamaskinen å oversette R-kode til maskinkode. R-pakken *compiler* gjør noe med dette ved å kompilere R-kode om til bytekode [12]. Bytekode går ikke like fort som maskinkode, men det kan være tid å spare i mange tilfeller. Denne pakken er veldig enkel å bruke. Den R-koden vi vil gjøre om til bytekode skriver vi først som en funksjon. Deretter skriver vi **cmpfun(f)** for å kompilere en funksjon f om til bytekode, og lagrer den nye kompilerte funksjonen som en ny funksjon. R-koden blir følgende for en funksjon x^2, der fc er den kompilerte funksjonen

```
f = function(x){ x*x }
fc = cmpfun(f)
```

Etter kompileringen vil den nye funksjonen i de fleste tilfeller kjøre raskere eller minst like fort som den opprinnelige funksjonen. De aller fleste innebygde funksjoner i R er byte-kompilerte. Vi vil senere se på effekten ved bruk av *compiler*-pakken.

3.2.3 Rcpp/Inline

Rcpp inneholder C++ klasser som gjør det enklere å bruke C++-kode kombinert med R-kode [7]. Dermed er det mulig å oppnå tilnærmet C++-hastighet i R. *Rcpp* gjør det også mulig å sende data fra C++ til R og fra R til C++. Det er viktig å være klar over at det tar (ørliten) tid å laste data mellom de to programmeringsspråkene, men gevinsten er ofte mye større enn dette (ørlite) tidstapet. I tillegg gjør *Rcpp* det mulig å bruke R-kode i C++. Dette gjør at vi trenger ikke å kunne mye C++ for å bruke *Rcpp*. Det krever riktignok at vi minimum kan gjøre noen enkle operasjoner i C++, som å definere heltall og vektorer osv. Dersom vi skal bruke *Rcpp* i Windows, må vi installere programmet (ikke R-pakke), *Rtools*.

Inline-pakken hører mer eller mindre sammen med *Rcpp* [13]. Stort sett bruker vi begge pakkene samtidig. *Inline* gjør det veldig enkelt å definere R-funksjoner som kjører kode i C/, C++ og Fortran. Videre vil vi holde oss til kombinasjonen av C++-kode og R-kode. Ved å skrive C++-kode som en *string* i R, og et kall på funksjonen **cxxfunction**, kan vi lage en R-funksjon med C++-kode. Dette forklares best ved et eksempel. Ved bruk av *inline* og *Rcpp* vil en funksjon x^2 defineres følgende i R

```
Cpp_kode <- "
double x_ = as<double>(x) ;
return wrap(x_*x_) ;
"

f_Rcpp <- cxxfunction(signature(x="numeric"),
                              body = Cpp_kode,
                              plugin = "Rcpp"
                              )
```

Vi ser C++-koden er av typen *string*, der $x_$ er definert som *double* og $x_^2$ returneres. **As**-funksjonen gjør R-objekter om til C++-objekter, mens **wrap**-funksjonen gjør det motsatte. C++-koden mates inn i *cxxfunction*-funksjonen under *body*. *Plugin* settes lik *Rcpp* for å bruke egenskapene til *Rcpp*. Under *signature* skriver vi alle variable som skal sendes fra R til C++, samt hvilken type disse variablene er. I dette tilfellet er variabelen x av typen *numeric*, som er det samme som typen *double* (R kaller *double* for *numeric*). Til slutt står vi igjen med en R-funksjon *f_Rcpp* som returnerer x^2 via C++.

Rcpp sin hovedegenskap er å gjøre det enkelt å bruke C++-kode sammen med R-kode, men det er også viktig å forstå når vi burde bruke *Rcpp*. *Rcpp* bør brukes til å løse opp flaskehalser i R, der treg R-kode erstattes med C++-kode som ofte er raskere. Senere skal vi se hvor effektivt det kan være å bruke *Rcpp* ved noen enkle eksempler.

3.2.4 Foreach/DoParallel

Siden standard R kun kjører på én prosessorkjerne, er vi avhengig av å benytte oss av pakker for å parallellisere. Å parallellisere vil si å kjøre simuleringer på flere prosessorkjerner samtidig. Blant de mest populære parallelliseringspakkene er *snow*, *multicore* (ikke tilgjengelig i Windows) og *foreach* [2]. Sistnevnte er en av de enkleste å sette seg inn i på grunn av enkel syntaks. Parallelliseringen krever at vi registrerer en klynge med prosessorkjerner. Derfor trenger vi *DoParallel*-pakken i tillegg [1]. Denne pakken bruker en såkalt *SOCKET*-klynge, som er den enkleste måten å koble sammen prosessorkjerner i en klynge. *SOCKET*-klynger fungerer stort sett kun på én maskin. Vi setter opp en slik klynge med funksjonen **makeCluster**, slik at kjernene kan kommunisere med hverandre. Når klyngen er satt opp, registrerer vi klyngen ved kommandoen **registerDoParallel**, og dermed er vi er klare for å parallellisere. Dersom vi er usikre på antall kjerner vi har på datamaskinen, vil kommandoen **detectCores()** returnere antall tilgjengelige prosessorkjerner.

Eksempel på bruk av *foreach* og *doParallel*, der m er antall simuleringer og f er funksjonen vi parallelliserer

```
#Setter opp klyngen med 8 kjerner:
klynge <- makeCluster(8)

#Registrerer klyngen med foreach-pakken:
registerDoParallel(klynge)

#Paralleliserer funksjonen f:
foreach(i=1:m) %dopar% f(i)
```

Dersom vi vil kjøre simuleringene sekvensielt i stedet for parallellt, erstatter vi %**dopar**% med %**do**%. Dette er nyttig dersom vi vil sammenligne og se hvor effektiv parallelliseringen er. Parallellisering med *foreach* fungerer bra som erstatning for en tidkrevende for-løkke, og funksjonen som evalueres trenger ikke å være avhengig av den tellende variabelen som sendes inn i funksjonen. Det er viktig å være klar over at vi stort sett kan parallellisere på kun én maskin med *foreach*. Dersom vi vil parallellisere over et nettverk av maskiner må vi bruke andre pakker som *Rmpi* og *snow*.

Det kan oppstå feilmeldinger om vi parallelliserer kode som inneholder *Rcpp*-funksjoner. Grunnen til det er at *Rcpp*-funksjoner må eksporteres ut til hver av prosessorkjernene som skal jobbe. Det anbefales å lage en egen lokal pakke [R-kommando: **Rcpp.package.skeleton()**], som inneholder de *Rcpp*-funksjonene som skal brukes. Deretter laster vi inn pakken på hver av kjernene. Dersom klyngen vi har satt opp heter *klynge* og den lokale pakken heter *minpakke*, laster vi inn pakken på hver kjerne med kommandoen: **clusterEvalQ(klynge, library(minpakke))**. Det å lage en egen pakke kan være krevende, men om vi sparer mye tid ved å bruke *Rcpp*-funksjoner, kan det være verdt å investere tid i å lage en pakke. Ved å skrive **vignette("Rcpp-package")** i kommandovinduet i R, kan vi lese mer om hvordan vi kan lage en pakke med *Rcpp*-funksjoner. Kort forklart skal C++-funksjoner i *src*-mappen og R-funksjoner i *R*-mappen (se eksempel på lokal R-pakke i appendix A.3.6). Dersom vi ikke vil lage en pakke, er det mulig å definere hver *Rcpp*-funksjon på hver kjerne, men det er ikke det mest tidseffektive.

3.2.5 Rmpi/Snow

Dersom vi vil parallellisere og har tilgang på flere maskiner koblet sammen i en klynge, fungerer det ikke lenger med *foreach/DoParallel*. For maskiner i klynge anbefales R-pakkene *Rmpi* og *snow*, der *MPI* står for *Message Passing Interface* [15, 16]. Kort fortalt gjør *MPI* det mulig for prosessorer på forskjellige maskiner å kommunisere sammen. *Snow*-pakken inneholder funksjonene som registrerer klyngen med prosessorkjerner, samt funksjoner som kjører parallelliseringen. *Rmpi*-pakken ligger i bakgrunnen og sørger for at kjernene i klyngen kan kommunisere over et nettverk av flere maskiner.

Parallellisering over nettverk er for de som vil ta parallellisering til et nytt nivå, og det er store muligheter til å få kjørt programmene raskere ved å utnytte mye datakraft i en dataklynge. Et eksempel på en slik klynge er Universitetet i Oslos super-PC, kalt *Abel*. *Abel*-klyngen består av 615 maskiner, der hver maskin har 16 fysiske prosessorer og hver prosessor er to-kjernet. Altså, hver maskin har 32 kjerner.

Eksempel på parallellisering med *Rmpi/snow*

```
#Setter opp klyngen med 100 kjerner
cl <- makeMPIcluster(100)

#Parallelliserer funksjonen f
parSapply(cl, 1:m, f)

#Avslutter klynge og MPI
stopCluster(cl)
mpi.exit()
```

Her er m antall simuleringer, der vi sender inn argument x = 1, ..., m inn i funksjonen f(x). Effekten av å parallellisere ser vi nærmere på i slutten av kapittelet.

3.3 Optimalisert programmering

Når vi skal programmere effektivt, gjelder det å forstå hvordan en datamaskin jobber. Det er lett å gå i feller, dersom vi ikke er bevisst på koden bak innebygde funksjoner. F. eks. inneholder **apply**-funksjonen i R en treg for-løkke. Derfor kan vi ofte få trege programmer som kan effektiviseres kraftig ved å bytte ut noen linjer med kode. Vi kan analysere kode i R med kommandoene **Rprof** og **summaryRprof**. Disse kommandoene viser en detaljert oversikt over kjøretiden til hver kodelinje, som gjør det enkelt å identifisere eventuelle flaskehalser.

For å illustrere best hvordan man kan øke hastigheten på R-programmer, skal vi se på enkle eksempler og se på kjøretid på programmene. Eksemplene er kjørt på en vanlig bærbar-PC med Windows 7 (64-bit) og Intel® Core™ i7 X920-prosessor med 4 GB minne. R-kode til eksempel 1, 2 og 3 ligger i appendix.

3.3.1 Enkle eksempler

Eksempel 1: Lage en vektor

Vi tar for oss 4 ulike metoder for å lage en vektor med heltall fra 1 til 10 000. Dette eksempelet er ment å illustrere hvor viktig det er å allokere minne før simuleringene kjøres for fullt, samt hvor effektivt vektorisering er. R-kode er som følger

```
#Metode 1
x = 0
for (i in 1:1e4){ x[i] = i }

#Metode 2
x = 1
for (i in 2:1e4){ x = c(x,i) }

#Metode 3
x = rep(0,1e4)
for (i in 1:1e4){ x[i] = i }

#Metode 4
x = 1:1e4
```

I metode 1 definerer vi x før for-løkken, og i løkken blir hvert element lik i for vektoren x. Dermed består vektoren x av heltall fra 1 til 10 000, fordi for-løkken løper fra 1 til 10 000 (1e4). I metode 2 definerer vi $x = 1$. For-løkken begynner fra 2 og går opp til i lik 10 000. Inne i løkken legger vi til et ekstra tall i vektoren x for hver gang løkken går. Metode 3 er den samme som metode 1, bortsett fra at vi allokerer minne for vektoren x før for-løkken kjøres. Metode 4 utspenner en vektor x fra 1 til 10 000 direkte uten løkke.

Kjøreeksempel fra R for 1000 kjøringer

```
      test replications elapsed relative
4 Metode 4         1000    0.02      1.0
3 Metode 3         1000   22.23   1111.5
2 Metode 2         1000   85.10   4255.0
1 Metode 1         1000  172.68   8634.0
```

Vi ser at det tar mye tid å legge til ett og ett element slik som i metode 1 og 2, men det tar ca. 50 % kortere tid ved metode 2. Det som er interessant er at hvis vi først allokerer minne for den vektoren vi vil lage, tar det mye kortere tid (se tiden for metode 3). Den aller mest effektive måten er metode 4. Vektoren blir laget direkte, og det tar kun tohundredelssekund.

Dette eksempelet viser veldig enkelt hvordan vi kan ende opp med en ineffektiv R-kode, dersom vi ikke er bevisst på hvilke metoder vi bruker. Det er viktig å allokere minne på forhånd når vi vil programmere effektivt. I tillegg ser vi hvor effektivt det er å vektorisere i motsetning til trege for-løkker.

Eksempel 2: Beregning av x^5

Foreløpig har vi kun sett på et banalt eksempel. I dette eksempelet vil vi gå et steg videre, og benytte oss av pakkene *Rcpp/inline* og *compiler* for å oppnå raskere R-kode. Vi skal teste hvor lang tid det tar å regne ut x^5 10 millioner ganger, der vi gjentar en for-løkke 1 000 ganger, hvorav hver for-løkke evaluerer x^5 10 000 ganger. Grunnen til at vi deler opp simuleringene i en for-løkke som repeteres, i stedet for å kjøre 10 millioner simuleringer direkte, er at vi skal se på effekten av for-løkker ved *Rcpp*. Vi har følgende 10 metoder vi vil teste

- **Standard R (Metode 1-3):** $x**5$, $x\hat{}5$ og $x*x*x*x*x$
- **Compiler (Metode 4-6):** Byte-kompilerte versjoner av metode 1-3
- **Rcpp (Metode 7-8), for-løkke i R:** $x*x*x*x*x$ og $x\hat{}5$
- **Rcpp (Metode 9-10), for-løkke i C++:** $x*x*x*x*x$ og $x\hat{}5$

Kjøreeksempel fra R for x = 2 og 1 000 kjøringer

	test	replications	elapsed	relative
10	x*x*x*x*x for Rcpp	1000	0.08	1.000
9	x^5 for Rcpp	1000	0.67	8.375
6	x*x*x*x*x compiler	1000	7.29	91.125
4	x^5 compiler	1000	8.46	105.750
5	x**5 compiler	1000	8.49	106.125
8	x*x*x*x*x Rcpp	1000	10.36	129.500
2	x**5 R	1000	10.56	132.000
1	x^5 R	1000	10.72	134.000
7	x^5 Rcpp	1000	11.78	147.250
3	x*x*x*x*x R	1000	17.97	224.625

Vi ser metode 3 med ren R-kode er den aller tregeste av alle de 10 metodene. Metode 3 tar hele 224 ganger så lang tid som den raskeste metoden. Riktignok er det ikke naturlig å programmere $x*x*x*x*x$, men denne metoden er tatt med for å vise effekten av byte-kompilering og kompilering ved C++. Metode 6 er den raskeste byte-kompilerte versjonen, og kjører ca. 2.5 ganger raskere enn metode 3. Vi ser samtidig at det er liten forskjell på om vi skriver $x**5$ eller $x\hat{}5$, uansett om vi byte-kompilerer eller kjører standard R-kode. Fra disse resultatene er det klart at *compiler*-pakken har en viss effekt, men ikke på langt nær like effektiv som de beste *Rcpp*-metodene.

Metode 9 og 10 er de klart raskeste, mens metode 7 og 8, ikke er i nærheten av samme hastighet. Det er fordi det er forskjell på hvor ofte vi overfører data mellom C++ og R. Dersom vi overfører data mellom de to programmeringsspråkene for hver simulering (metode 7 og 8), ser vi at *Rcpp* ikke har noe særlig effekt. Om vi derimot overfører mer arbeid (for-løkken) over i C++, og utnytter farten C++ har, ser vi styrken til *Rcpp*. Ved metode 9 og 10 trenger vi kun å overføre data mellom C++ og R 1 000 ganger, i stedet for 10 millioner ganger. Dette gjør at metode 9 og 10 utklasser de andre metodene på tid, hvorav metode 10 er 8 ganger raskere enn metode 9.

En interessant observasjon er at $x*x*x*x*x$-metoden er hurtigere enn å skrive $x\hat{}5$, både for *Rcpp* og *compiler*. Her ser vi forskjellen på at C++ er et kompileringsspråk og R et interpreterende språk. Dette eksempelet viser at *compiler*- og *Rcpp*-pakken er nyttige for å øke hastigheten i R, men vi må være bevisst på hvordan de skal brukes mest effektivt. Vi ser at *compiler*-pakken er effektiv for å optimalisere kode enkelt og raskt, mens vi får enda større effekt av å programmere mer avansert ved bruk av *Rcpp*.

Eksempel 3: Beregning av gjennomsnitt

I dette eksempelet vil vi vise at standardfunksjoner i R kan simuleres raskere. Vi vil se på alternative metoder for beregning av gjennomsnitt i R, der vi beregner gjennomsnittet av 100 standard uniformt fordelte variable. Vi tester den innebygde gjennomsnittsfunksjonen i R, **mean**, mot standardformelen for gjennomsnitt ($\sum_{i=1}^{n} x_i$), der x er en vektor og n er lengden til x. Testen går ut på å beregne gjennomsnittet av vektoren x 10 millioner ganger, der vi gjentar en for-løkke 1 000 ganger og for-løkken består i å beregne gjennomsnitt 10 000 ganger.

De 7 ulike metodene vi tester er som følger

- **Standard R (Metode 1-2):** mean(x) og $\sum_{i=1}^{n} x_i$
- **Compiler (Metode 3):** $\sum_{i=1}^{n} x_i$
- **Rcpp (Metode 4-5), for-løkke i R:** mean(x) og $\sum_{i=1}^{n} x_i$
- **Rcpp (Metode 6-7), for-løkke i C++:** mean(x) og $\sum_{i=1}^{n} x_i$

Kjøreeksempel fra R for 1 000 kjøringer

```
                         test replications elapsed relative
7  sum(x)/x.size() for Rcpp         1000    0.93    1.000
6          mean(x) for Rcpp         1000    3.03    3.258
3 sum(x)/length(x) compiler         1000   12.07   12.978
4              mean(x) Rcpp         1000   13.85   14.892
5      sum(x)/x.size() Rcpp         1000   13.87   14.914
2       sum(x)/length(x) R          1000   17.07   18.355
1                mean(x) R          1000   97.02  104.323
```

Vi ser *mean*-funksjonen i R (metode 1) er over fem ganger så treg som om vi beregner gjennomsnittet direkte (metode 2). Vi har også samme trend i dette eksempelet som i eksempel 2, at *compiler*-metoden er raskere enn *Rcpp*-metodene med for-løkke i R. Metodene 3, 4 og 5 gir en liten økning i simuleringshastighet i forhold til metode 2, men dersom vi kjører for-løkkene i C++, ser vi igjen at *Rcpp*-metodene er klart raskest, hvorav metode 7 er tre ganger raskere enn metode 6. Det lønner seg å gjøre mest mulig av jobben i C++, og det å bruke en innebygd R-funksjon i C++ (metode 6), er ikke fullt så effektivt som å beregne gjennomsnittet i C++ direkte (metode 7).

Fra utgangspunktet vårt (metode 1) ser vi at det er mulig å beregne gjennomsnitt over 100 ganger raskere om vi benytter oss av *Rcpp*. Dersom vi hadde overført enda mer av beregningene i C++ og færre overføreringer av data mellom C++ og R, hadde det gått enda fortere. Det er derfor viktig å ikke ta for gitt at standardfunksjonene i R er de mest tidsoptimale, fordi det ofte er fullt mulig å gjøre små modifikasjoner som er raskere. Mest sannsynlig er *mean*-funksjonen treg fordi den er tilpasset flere forskjellige tilfeller, som f. eks. gjennomsnittet av vektorer, matriser, 3D-matriser osv. Derfor kan ofte egenproduserte funksjoner være det mest tidseffektive, fremfor en innebygd R-funksjon. Samtidig ser vi at *compiler* og spesielt *Rcpp* gir signifikant raskere simuleringer i R.

3.3.2 Vektorisering og parallellisering

Når vi simulerer, vil vi ha best mulig presisjon på beregningene våre. Dette krever ofte mange simuleringer. Jo flere simuleringer, jo bedre presisjon. Samtidig ønsker vi å få resultatene så fort som mulig. Kombinasjonen av dette gjør at presisjonen blir begrenset, fordi vi kan ikke vente for evig på at simuleringene blir ferdig. Derfor er det viktig å programmere smart og effektivt, slik at vi oppnår best mulig presisjon og minst mulig ventetid. Dette motiverer oss til å øke hastigheten på simuleringene.

Det er ikke alltid like lett å gjøre et program raskere, men dersom vi ikke er bevisst på å programmere tidseffektivt, er det stor sannsynlighet for at det er muligheter for effektivisering. Vi har sett fra eksemplene tidligere at det ofte ikke skal mye til for å gjøre et program raskere. Hvorvidt en effektivisering av et program er verdt det eller ikke, avhenger mye av hvor lang kjøretid det er på programmet. La oss si vi har to programmer, som gjør samme beregninger, der det ene programmet er dobbelt så raskt som det andre. Dobbelt så kort tid er mye når det er snakk om programmer som kjører i flere timer eller dager, mens for et kortere program er det ikke så nøye om man må vente f. eks. 1 eller 2 sekunder. Jo lengre kjøretid på et program, jo mer tid er det å spare inn ved å effektivisere kode.

For å øke hastigheten på programmer gjelder det først og fremst å få datamaskinen til å jobbe med vektorer og unngå løkker. Stort sett er det mulig å unngå løkker og programmere det samme med bare vektorer. Løkker går tregt, mens vektorer er lette å jobbe med for en datamaskin. Vektorisering kan ofte føre til at vi tar opp mer minne i datamaskinen, og derfor er det viktig at vektorene ikke blir for lange. Dersom vektorene tar opp mye minne fører det til at farten på simuleringene kjører tregt, men hvis vektorene er for korte får vi ikke full effekt av å vektorisere. Datamaskiner jobber raskest når vi bruker lite minne, men det betyr ikke at vektorene skal være kortest mulig. Det gjelder å jobbe med passe lange vektorer, slik at vi ikke sprenger minnet, men samtidig har en viss lengde på vektorene for å få full effekt av vektoriseringen.

Fordi minnebruken påvirker simuleringshastigheten, er det greit å være oppmerksom på hvor mye minne vi bruker når vi kjører simuleringer. Et standard tall i R tar opp 8 byte i minnet på en maskin. Siden R 64-bit kun takler å jobbe med objekter som tilsammen tar opp ca. 8 GB minne, kan lengden på en vektor maksimalt være $\frac{8*1024^3(\text{1 GB i bytes})}{8(bytes)} \approx 1073$ millioner uten at R kræsjer. For R 32-bit vil maksimalt minne være halvparten av dette. Dersom vi først har en vektor som tar opp alt minnet, vil det være veldig vanskelig å gjøre ytterligere beregninger, fordi det ikke er noe ekstra minne å gå på. Derfor vil det i praksis være lurt å jobbe med vektorer som ikke har noe særlig mer enn 100 millioner elementer, dersom vi skal kunne gjøre beregninger uten å sprenge minnet.

I tillegg til å vektorisere kan vi spare tid på å parallellisere simuleringene. Det vil si å kjøre simuleringer samtidig, som er **uavhengige** av hverandre. Parallellisering kan foregå på flere forskjellige måter. Vi kan f. eks. dele opp et program og kjøre det på mange ulike maskiner, eller kjøre et program på flere forskjellige prosessorkjerner på en maskin. En vanlig bærbar-PC har i dag normalt 2 eller 4 prosessorkjerner, og vi vil med en slik maskin ha mulighet til å gjøre simuleringer ca. 2-4 ganger så fort ved å parallellisere. Vi kan også parallellisere ved å

utnytte et nettverk av prosessorkjerner. Parallellisering handler om å utnytte datakraften som er tilgjengelig, og siden standard R kun kjører på én prosessorkjerne, er det store muligheter for å oppnå betraktelig raskere simuleringer. De fleste simuleringsproblemer kan simuleres, spesielt MC-simuleringer, siden disse som oftest består av uavhengige simuleringer.

Kjøretiden på et parallellisert program avhenger av hvor mange prosessorkjerner vi har tilgjengelig. Jo flere kjerner vi har, jo fortere går simuleringene. Teoretisk sett vil tiden på et program gå ned med en faktor på antall kjerner vi parallelliserer på, om vi ser bort fra kommunikasjonstid mellom kjernene. Dermed er det antall kjerner tilgjengelig som begrenser hvor fort simuleringene kan kjøres ved parallellisering. Dersom vi lar T_p være kjøretiden på et parallelliserbart program, som kjøres på p prosessorkjerner, vil kjøretiden være følgende

$$T_p = \frac{T_1}{kp}, \quad 0 \leq k \leq 1 \tag{3.1}$$

Her er kp hvor mange ganger raskere programmet kjører (optimaliseringsfaktoren) ved parallellisering, slik at k angir hvor stor andel av kjernene som utgjør optimaliseringsfaktoren. Dermed beskriver *1-k* kommunikasjonstid mellom kjernene, som er kjøretiden av programmet som ikke fører til raskere kjøring av programmet. Når vi ikke parallelliserer, er k=1. Det er naturlig at det går mer tid på kommunikasjon mellom kjernene, jo flere kjerner det er i en klynge. Derfor vil k minke, desto flere kjerner vi parallelliserer på.

3.4 Optimalisert reserveberegning

Algoritme 1 er standardalgoritmen for reserveberegninger i skadeforsikring. Ved å benytte oss av optimaliseringsmetoder, som er nevnt tidligere i kapittelet, kan vi optimalisere Algoritme 1. Først og fremst er det mulig å unngå for-løkken i Algoritme 1 ved å vektorisere. Den vektoriserte utgaven av Algoritme 1 kaller vi Algoritme 2.

3.4.1 Ny algoritme

Algoritme 2

Beregning av $(1-\epsilon)$-reserve med skadebegrensning på polisenivå.

0. Input: m, b, θ, λ.
1. Trekk m antall $\hat{N}^* \sim$ Poisson(λ)
2. Trekk sum($\hat{N}^*$) antall $\hat{Z}^* \sim$ modell med parameter θ
3. $\hat{Z}^*$ = min($\hat{Z}^*$,b)
4. $\hat{N}^*_{kum}$ = cumsum($\hat{N}^*$)
5. $\hat{Z}^*_{kum}$ = cumsum($\hat{Z}^*$)
6. x1 = $\hat{Z}^*_{kum}[\hat{N}^*_{kum}]$
7. x2 = (0, x1[-m])
8. $\hat{X}^*$ = x1-x2
9. Sortér $\hat{X}^*$ stigende
10. $(1-\epsilon)$-reserven = $(1-\epsilon)$-persentilen i $\hat{X}^*$

Her er m antall simuleringer, b er skadebegrensning per polise, θ er parametersett under aktuell modell for skadeutbetalingene og λ er Poisson-parameteren som beskriver forventet antall skader.

Parameterene som må defineres på forhånd er de samme som i Algoritme 1. I linje 1 trekker vi tallet på antall skader i alle m scenarioer, før vi i linje 2 trekker alle skadeutbetalingene i alle scenarioene på en gang. Deretter slår skadebegrensningen inn i linje 3. I linje 4 og 5 kumulativsummeres de to vektorene som inneholder antall skader i hvert scenario ($\hat{N}^*$) og alle skadeutbetalingene ($\hat{Z}^*$). Dermed vil vektoren $\hat{N}^*_{kum}$ inneholde indeksene til siste skade i hvert scenario (antall: m).

I linje 6 plukker vi ut verdiene i $\hat{Z}^*_{kum}$-vektoren som tilsvarer indeksene til siste skade i hvert scenario, og lagrer disse verdiene som *x1*. Deretter lager vi en vektor *x2* som er lik *x1*-vektoren, bortsett fra at siste element i *x1* ikke er med og første element i *x2* er lik 0. Dersom vi tar differansen mellom *x1* og *x2*, får vi summen av samlede skadeutbetalinger for hvert scenario. I linje 8 gjør vi nettopp dette, før vi i linje 9 sorterer alle samlede skadeutbetalinger stigende, og kaller dette for $\hat{X}^*$-vektor. Til slutt plukker vi ut (1-ϵ)-reserven fra $\hat{X}^*$, og dermed har vi en vektorisert algoritme for beregning av reserve.

3.4.2 Hastighetstester

Videre skal vi teste hvor mye raskere Algoritme 2 er enn Algoritme 1 i R, under følgende forutsetninger

- m = 100 000 (antall MC-simuleringer per reserve)
- $\lambda = 50$ (antall skader per scenario)
- Z $\sim$ Gamma(α, β)
- $\alpha = \beta = 1$

Skadebegrensningen b vil vi se bort i fra i testene. Et scenario er f. eks. 1 år.

Vi vil gjøre to tester, der vi måler tiden på kjøringene for beregning av 99 %-reserver. Den første går ut på å teste Algoritme 1 og 2 mot hverandre, og samtidig anvende pakkene *Rcpp* og *compiler* for å øke hastigheten på simuleringene. Vi vil ta tiden på 100 repetisjoner av hver testmetode for å gjøre resultatene mindre tilfeldige. Testmetodene er nummerert slik at det blir lettere å referere til hver testmetode senere, og nummereringen har ingen annen funksjon enn dette.

Den andre testen vil bestå av å teste Algoritme 1 og 2 mot hverandre for 160 reserveberegninger (160 repetisjoner av algoritmene), der vi vil inkludere parallellisering på både en maskin (*foreach/doParallel*) og flere maskiner (*Rmpi/snow*). Her vil vi ta tiden på 10 repetisjoner av hver testmetode. Begge testene er kjørt i Linux Centos 5 (64 bit) på Abel-klyngen ved Universitetet i Oslo, der vi maksimalt har hatt 160 tilgjengelige prosessorkjerner (32 per maskin). Hver maskin har 16 to-kjernede prosessorer av typen Intel E5-2670 med GB minne per prosessor. R-kode til testene er i appendix.

Test 1

Metodenr.	Metodenavn	Tid (s)	Relativ tidsdifferanse
3	Algoritme 2 (Rcpp)	61.52	1.00
5	Algoritme 2 (compiler)	70.33	1.14
2	Algoritme 2	71.23	1.16
6	Algoritme 1 (Rcpp)	90.93	1.48
4	Algoritme 1 (compiler)	127.05	2.07
1	Algoritme 1	150.15	2.44

Tabell 3.1: 100 repetisjoner, hvorav en repetisjon inneholder en reserveberegning. Tid målt i sekunder. Relativ tidsdifferanse er målt mot raskeste metode.

Vi ser fra tabell 3.1 at *compiler*-pakken gir en tidsforbedring, der det er best effekt på Algoritme 1. Pakken for de mer avanserte programmerere *Rcpp*, har en enda sterkere effekt enn *compiler*. Algoritme 2 i standard R er over dobbelt så rask som Algoritme 1. Dersom vi bruker *Rcpp* på Algoritme 2, får vi ytterligere forbedring på ca. 13-14 %. Her ser vi at den raskeste reserveberegningsmetoden er Algoritme 2 sammen med *Rcpp*-pakken. Totalt sett er utgangspunktet vårt, Algoritme 1 i standard R, 2.44 ganger tregere enn den beste metoden

(metode 3). Denne testen viser at Algoritme 2 er klart raskere enn Algoritme 1, og en slik forbedring er nyttig når reserver beregnes i stor skala.

Videre i test 2 vil vi se nærmere på effekten ved parallellisere reserveberegninger. Metode 4, 5 og 6 fra test 1 vil vi se bort i fra i test 2 for å gjøre testen mer oversiktlig. Vi tar kun med de mest effektive/relevante metodene med videre.

Test 2

Metodenr.	Metodenavn	Tid (s)	Relativ tidsdifferanse
5	Algoritme 2 (Rcpp), 160 kjerner	9.03	1.00
4	Algoritme 2 (Rcpp), 32 kjerner	47.84	5.30
3	Algoritme 2 (Rcpp), 1 kjerne	895.48	99.16
2	Algoritme 2, 1 kjerne	1010.39	111.89
1	Algoritme 1, 1 kjerne	2162.82	239.51

Tabell 3.2: 10 repetisjoner, hvorav en repetisjon inneholder 160 reserveberegninger. Tid målt i sekunder. Relativ tidsdifferanse er målt mot raskeste metode.

Metode 1, 2 og 3 i test 2 er kjørt på kun én prosessorkjerne, slik det gjøres i standard R. Hver av disse metodene består av en for-løkke som kjører de ulike algoritmene 160 ganger. Metode 4 er parallellisert på én maskin med 32 tilgjengelige prossessorkjerner ved bruk av R-pakkene *foreach/doParallel*, mens metode 5 er parallellisert over flere maskiner med 160 tilgjengelige prosessorkjerner ved bruk av R-pakkene *Rmpi/snow*.

Vi ser fra tabell 3.2 at den relative tidsforskjellen mellom metode 1, 2 og 3 er mer eller mindre lik som i test 1 for tilsvarende metoder. Det mest interessante er hvor mye tid vi sparer ved å parallellisere. Ved å parallellisere på en maskin og 32 prosessorkjerner får vi en kraftig økning i hastighet på reservesimuleringene. En enda kraftigere økning får vi ved å parallellisere over 160 prosessorkjerner. Metode 5 er hele 239 ganger raskere enn utgangspunktet, metode 1. Metode 1 tar ca. 36 minutter, mens metode 5 tar kun 9 sekunder. Det er en kraftig forbedring. Folk flest har dog stort sett ikke mer enn én maskin å parallellisere på, men dette viser potensialet ved å parallellisere. Selv om vi hadde hatt få prosessorkjerner å parallellisere på, vil det være tidsbesparende å parallellisere. Parallellisering handler om å utnytte datakraften som er tilgjengelig.

Siden vi parallelliserer over 160 prosessorkjerner, kunne vi potensielt ha oppnådd at programmet vårt ble 160 ganger raskere. I praksis skjer ikke dette fordi det går bort tid på kommunikasjon mellom kjernene. Metode 3, som er kjørt på én prosessorkjerne, tar hele 99 ganger lenger tid enn metode 1, som er parallellisert på 160 prosessorkjerner. Det betyr at tiden på programmet vårt går ned med en faktor på ca. 62 % av antall prosessorkjerner vi benytter oss av. Når vi parallelliserer på kun én maskin (metode 4), går tiden på programmet ned med en faktor på ca. 58 % av de 32 prosessorkjernene vi bruker. Generelt er det slik at jo færre prosessorkjerner det er i en klynge, jo mindre relativ kommunikasjonstid er det mellom kjernene. Grunnen til at det ikke er slik i dette tilfellet (32 vs. 160 kjerner) skal vi se nærmere på.

Parallelliseringen på én maskin (metode 4) ser vi tar 5.3 ganger lenger tid enn å parallellisere over 160 kjerner (metode 5). Metode 4 bruker 5 ganger færre prosessorkjerner enn metode 5. Derfor skulle det teoretisk sett vært en relativ tidsdifferanse på 5 mellom disse metodene. I tillegg ville det vært naturlig å tro at kommunikasjonstiden mellom prosessorkjernene var mindre på én maskin enn over flere maskiner, slik at den relative tidsdifferansen burde være mindre enn 5. Grunnen til at det ikke er slik er at metode 4 parallelliserer ved bruk av en *SOCKET*-klynge. Når vi parallelliserer over flere maskiner, bruker vi en *MPI*-klynge som er mer avansert og effektiv. Derfor får vi en relativ tidsdifferanse mellom metode 4 og 5 på mer enn 5, nærmere bestemt 5.3 i dette tilfellet.

3.5 Oppsummering

I dette kapittelet har vi introdusert nyttige R-pakker som kan brukes til å optimalisere R-kode. Videre har vi vist noen enkle eksempler på bruk av disse R-pakkene, og beskrevet optimaliseringsmetoder som vektorisering og parallellisering. Disse metodene har vi brukt for å optimalisere simuleringen av reserver i skadeforsikring, i tillegg til å kombinere C++-kode med R-kode.

Ved å vektorisere Algoritme 1 fikk vi en ny og egenutviklet algoritme, som vi kalte Algoritme 2. Videre programmerte vi Algoritme 2 i C++-kode ved bruk av *Rcpp*-pakken og fikk ytterligere raskere kode. Til slutt parallelliserte vi C++-versjonen av Algoritme 2 både på én og flere maskiner. Dette ga oss ca. 240 ganger raskere reserveberegninger enn utgangspunktet, der optimaliseringen steg for steg ser slik ut (optimaliseringsfaktor er ytterligere relativ tidsforbedring)

I. Algoritme 1 (utgangspunktet)

II. Algoritme 2, **Optimaliseringsfaktor: 2.11**

III. Algoritme 2 *Rcpp*, **Optimaliseringsfaktor: 1.16**

IV. Algoritme 2 *Rcpp* 32 kjerner, **Optimaliseringsfaktor: 18.72**

V. Algoritme 2 *Rcpp* 160 kjerner, **Optimaliseringsfaktor: 5.30**

Her er optimaliseringsfaktorene under II og III hentet fra test 1, mens optimaliseringsfaktorene under IV og V er fra test 2.

Fra test 1 har vi at Algoritme 2 er 2.11 ganger raskere enn Algoritme 1, og *Rcpp*-versjonen av Algoritme 2 er ytterligere 1.16 ganger raskere. Test 2 viste oss at parallellisering på 32 og 160 prosessorkjerner gjorde *Rcpp*-versjonen av Algoritme 2 henholdsvis 18.72 og 99.16 ganger raskere. Dermed blir den totale optimaliseringsfaktoren for parallellisering på 160 kjerner: 2.11 * 1.16 * 99.16 $\approx$ 242. Dette stemmer godt overens med den relative tidsdifferansen mellom metode 1 og 5 i test 2 (optimaliseringsfaktor: 239).

Vi ser at parallelliseringen utgjør den største optimaliseringsfaktoren, mens selve Algoritme 2 er drøye to ganger raskere enn Algoritme 1. Ved å bruke *Rcpp*-pakken får vi en ørliten tidsbesparelse, men nok til å gjøre Algoritme 2 over 10 % raskere. Dette viser oss at parallellisering

er et kraftig optimaliseringsverktøy, og det kan brukes i utrolig mange situasjoner, så fremt simuleringene er uavhengige av hverandre.

Kapittel 4

Feilreservering

4.1 Innledning

Feilreservering er et tema i skadeforsikring som blir mer og mer aktuelt. Det er vanlig i skadeforsikring å ta utgangspunkt i en bestemt statistisk modell for å estimere fremtidige skadeutbetalinger og beregne nødvendige reserver. Her vil vi se på reserveringsfeil ved beregning av 99%-reserver når vi velger feil modell for erstatningskravene. De underliggende datasettene vil bli trukket fra en bestemt korrekt modell, som er forskjellig fra modellen vi velger. Vi vil studere to situasjoner, der valgt modell er **gamma**-modellen i begge situasjonene, mens korrekt modell er enten **log-normal**-modellen (4.2) eller **pareto**-modellen (4.3).

For å analysere reserveringsfeil må vi først beregne tre ulike typer 99 %-reserver

- ψ, basert på korrekt modell, Log-normal(μ,σ)/Pareto(α_p)
- ψ_0, basert på valgt modell, Gamma, og optimalt parametersett (α_0, β_0)
- $\hat{\psi}$, basert på valgt modell, Gamma, og estimert parametersett ($\hat{\alpha}$, $\hat{\beta}$)

Det optimale parametersettet (α_0, β_0) minimerer avstanden (2.29) mellom gamma (valgt modell) og log-normal/pareto (korrekt modell), slik at ψ_0 er optimal reserve under gamma-modellen. Verdier på α_0 og β_0 er i appendix A.2.1. Estimeringen av parametersettet ($\hat{\alpha}$, $\hat{\beta}$) gjøres ved ML under gamma-modellen, basert på data trukket fra korrekt modell, slik at $\hat{\psi}$ er estimert reserve under gamma-modellen.

Fra seksjon 2.3.4 vet vi at **total** reserveringsfeil består av to komponenter, **systematisk feil** og **tilfeldig feil**, der de tre kalkulerte reservene ψ, ψ_0 og $\hat{\psi}$ brukes til å bestemme størrelsene på de ulike typene feil. Det er interessant å se hvor stor total feil er i ulike situasjoner, og i tillegg måle forholdet mellom den systematiske og tilfeldige feilen. Forholdet mellom systematisk og tilfeldig feil måler vi ved å se på **relativ systematisk feil**. De fire ulike typene feil er definert slik

- Total feil $= |\hat{\psi} - \psi| = |\hat{\psi} - \psi_0 + \psi_0 - \psi|$
- Systematisk feil $= |\psi_0 - \psi|$

- Tilfeldig feil = $|\hat{\psi} - \psi_0|$
- Relativ systematisk feil = Systematisk feil/(Systematisk feil + Tilfeldig feil)

Den systematiske feilen er en konstant og kommer av at vi velger feil modell (gamma). Dersom vi velger riktig modell, får vi ikke systematisk feil, og den totale feilen består dermed kun av tilfeldig feil.

Den tilfeldige feilen skyldes estimeringsfeil av parametere ($\hat{\alpha}$, $\hat{\beta}$) under gamma-modellen. Tilfeldig feil avhenger av antall observasjoner n, og den asymptotiske teorien gir at tilfeldig feil går mot 0 når $n \to \infty$. Når tilfeldig feil er 0, vil total feil kun bestå av systematisk feil, gitt at vi velger feil modell.

I dette kapittelet studerer vi total, systematisk og tilfeldig feil i ulike situasjoner ved å beregne total feil og relativ systematisk feil. Den relative systematiske feilen beskriver forholdet mellom systematisk og tilfeldig feil, og ved å studere relativ systematisk feil sammen med total feil, studerer vi alle de tre typene feil samtidig (total/systematisk/tilfeldig feil). Videre ser vi på parameterfeil, som beskriver den tilfeldige feilen ytterligere, der vi undersøker om vi overestimerer eller underestimerer parametere, samt standardavvik i parameterestimatene. Til slutt ser vi på sannsynligheten for å velge riktig modell for å se hvor realistisk det er at vi velger feil modell. Vi ser kun på to aktuelle modeller i hver seksjon. Disse modellene er gamma- og log-normal-modellen i (4.2.4), og gamma- og pareto-modellen i (4.3.4).

Seksjonene i kapittelet er bygd opp på følgende måte

I. Systematisk feil

II. Relativ systematisk feil og total feil

III. Parameterfeil

IV. Sannsynlighet for å velge riktig modell

Vi studerer ulike situasjoner, der skadebegrensning per polise (b), antall observasjoner (n) og parametere under korrekt modell (σ/α_p) varierer.

Vi gjør MC-simuleringer av reserver og reserveringsfeil under følgende forutsetninger

- m = 100 000 (antall MC-simuleringer per reserve)
- m_b = 10 000 (antall MC-simuleringer av reserveringsfeil)
- $\lambda = 50$ (forventet antall skader per scenario)
- b = 1, 2, ..., 10, ∞ (11 ulike skadebegrensninger per polise)
- n = 20, 40, ..., 200, 400, ..., 1000, 10 000 (15 ulike antall observasjoner)
- Log-normal: $\mu = 0$, $\sigma = 0.1, 0.2, ..., 1.0$ (10 ulike parametersett)
- Pareto: $\alpha_p = 3, 4, ..., 10$ (8 ulike parametere)

Dette gir oss henholdsvis 1650 og 1320 ulike situasjoner når korrekt modell er henholdsvis log-normal og pareto. Simuleringen av disse situasjonene ville vært tilnærmet umulig uten

optimalisering av reserveberegninger i kapittel 3.

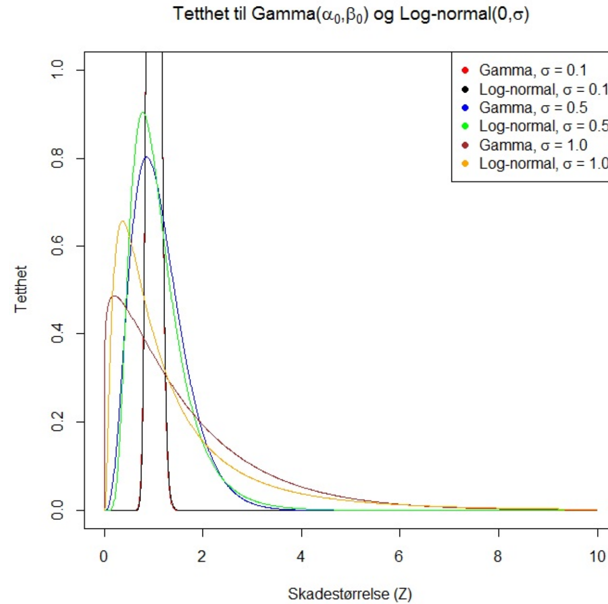
Det er valgt $m = 100\ 000$ simuleringer per reserve, siden vi måler differanser mellom to reserver og det er viktig med best mulig presisjon i reservene. Videre har vi $m_b = 10\ 000$ simuleringer av ulike typer feil for å sikre god presisjon i sluttresultatet. Poisson-parameteren, $\lambda = 50$, gir uttrykk for forventet antall skader per scenario, som f. eks. tilsvarer en liten brannskadeportefølje med forventet 50 skader i året. Skadebegrensningsnivåene er valgt slik at de stort sett dekker hele fordelingen til skadeutbetalingene. Antall observasjoner spenner seg fra 20 og opp til 10 000 for å dekke intervallet der tilfeldig feil går fra å være veldig stor til å være forsvinnende liten.

Parameterene under log-normal- og pareto-modellen er valgt for å få standardavvik i underliggende data til å variere fra relativt lite til relativt store standardavvik. Under log-normal-modellen er forventningsparameteren valgt fast for å forenkle feilanalysen og begrense antall situasjoner. De to ulike situasjonene der vi har underliggende modell, log-normal og pareto, er ikke direkte sammenlignbare i form av samme forventning og standardavvik for skadeutbetalingene. Hensikten med valg av parameterene, er å se hvor følsomme de ulike typene feil er i forhold til parametere i underliggende korrekt modell.

Beregningen av sannsynlighet for å velge riktig modell er uavhengig av parametere som inngår i beregningen av reserver (m, λ og b). Ellers er disse beregningene gjort under samme forutsetninger som ovenfor, og består av $\mathrm{m}_b = 10\ 000$ MC-simuleringer.

4.2 Gamma-modellen mot Log-normal-modellen

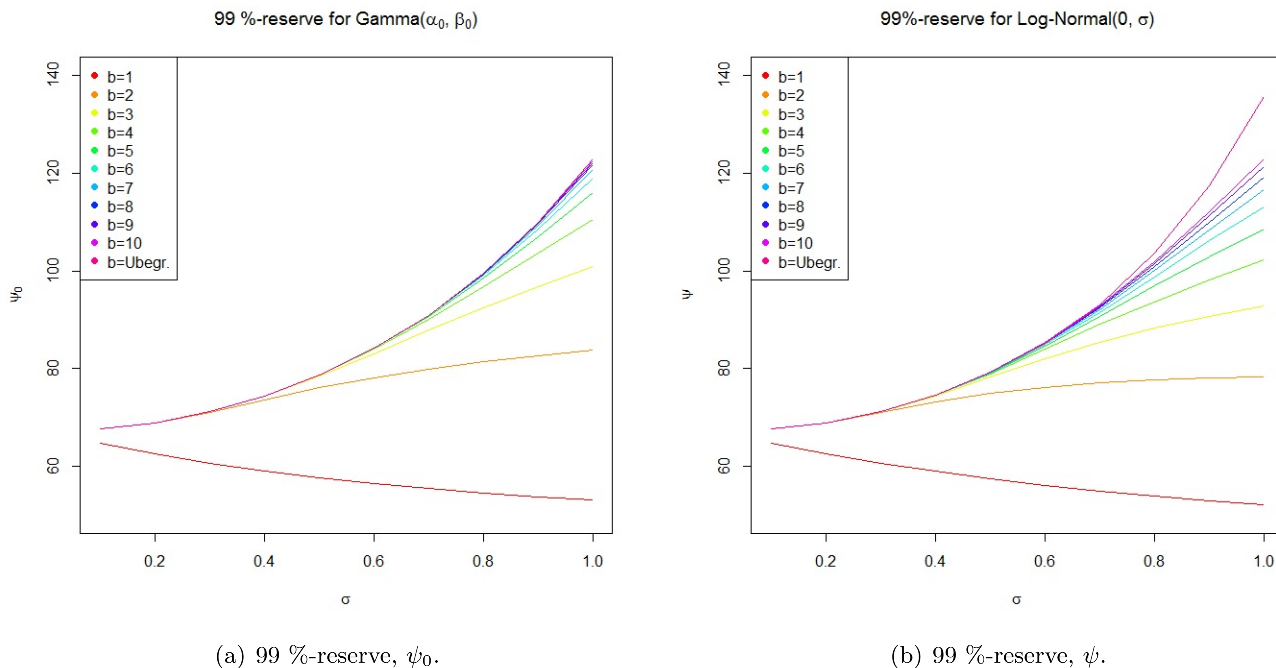
Vi begynner med å se på fordelingene til Gamma(α_0, β_0) og Log-normal(0, σ) for utvalgte σ, slik at vi får et inntrykk av størrelsen på skadeutbetalingene vi trekker.



Figur 4.1: Tettheten til Gamma(α_0, β_0) og Log-normal(0, σ) for $\sigma = 0.1,\ 0.5,\ 1.0$, der skadestørrelsene f.eks. er i millioner NOK.

Figur 4.1 viser oss at gamma-modellen treffer log-normal-modellen ganske presist når $\sigma = 0.1$. Det er dermed liten grunn til å tro at det vil være store avvik mellom reservene, ψ_0 og ψ, for lav σ. Vi ser at log-normal-modellen har en tyngre hale enn gamma-modellen for større σ, mens det samtidig ser ut som det er større sannsynlighet for å trekke de minste skadeutbetalingene under log-normal-modellen enn gamma-modellen. Videre vil vi se på systematisk feil og reservene, ψ_0 og ψ, og se hvordan disse reservene avhenger av skadebegrensning b og standardavvik σ.

4.2.1 Systematisk feil

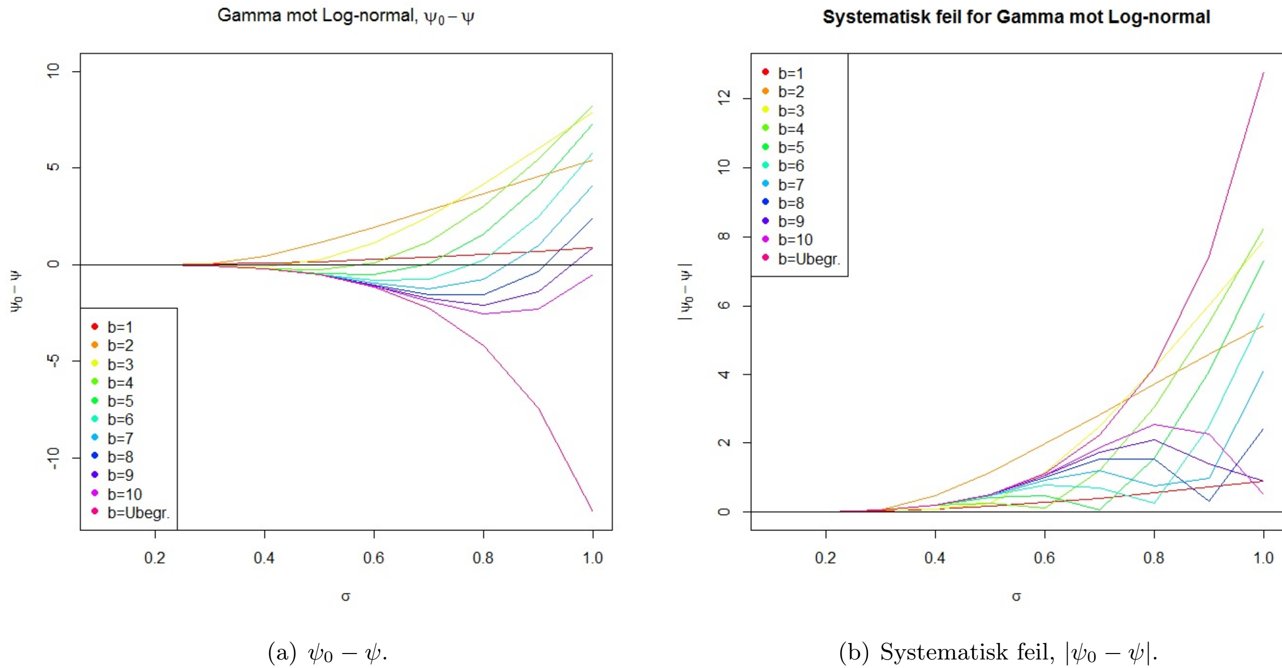


(a) 99 %-reserve, ψ_0.

(b) 99 %-reserve, ψ.

Figur 4.2: 99 %-reserve for ulike skadebegrensninger (b) under modell Gamma(α_0, β_0) og Log-normal(0, σ).

Vi ser fra figur 4.2 at 99 %-reserven er ganske lik for både gamma- (a) og log-normal-modellen (b). En av forskjellene mellom de to modellene, er at reserven ser ut til å flate ut mer i diagrammet under log-normal enn gamma for lave skadebegrensningsnivåer (fra b=1 til b=4), når standardavviket øker. Det er meget små forskjeller på reservene under gamma-modellen for skadebegrensningsnivå 6 og høyere, mens under log-normal-modellen for tilsvarende skadebegrensningsnivåer, er det større avvik i reserve når standardavvik og skadebegrensningsnivå øker. Det er verdt å merke seg at for skadebegrensningsnivå 1 er reserven fallende når standardavviket øker. Grunnen til det er at både gamma- og log-normal-modellen for lav σ, har større sannsynlighet for å trekke skader tett opptil 1 enn for stor σ. Videre vil vi se nærmere på den systematiske feilen.



(a) $\psi_0 - \psi$.

(b) Systematisk feil, $|\psi_0 - \psi|$.

Figur 4.3: $\psi_0 - \psi$ (a) og systematisk feil (b) for Gamma mot Log-normal for ulike skadebegrensninger per polise.

I figur 4.3 (a) ser vi på ψ_0 trukket fra ψ, mens figur 4.3 (b) viser den systematiske feilen. Den systematiske feilen blir stort sett større og større for økende standardavvik (σ), men skadebegrensningen gjør at vi får et blandet bilde. Vi ser at ψ_0 er større enn ψ for lavt skadebegrensningsnivå og σ større enn 0.3. For skadebegrensningsnivå mellom 4 og 9 derimot, går ψ fra å være større enn ψ_0 for mellomstor σ, til å være mindre enn ψ_0 for stor σ. Vi ser at når skadebegrensningsnivået er 10 eller mer, er ψ større enn ψ_0 for alle σ. Når ψ_0 er større enn ψ, betyr det at vi overestimerer reserven under gamma-modellen, mens vi underestimerer dersom ψ_0 er mindre enn ψ.

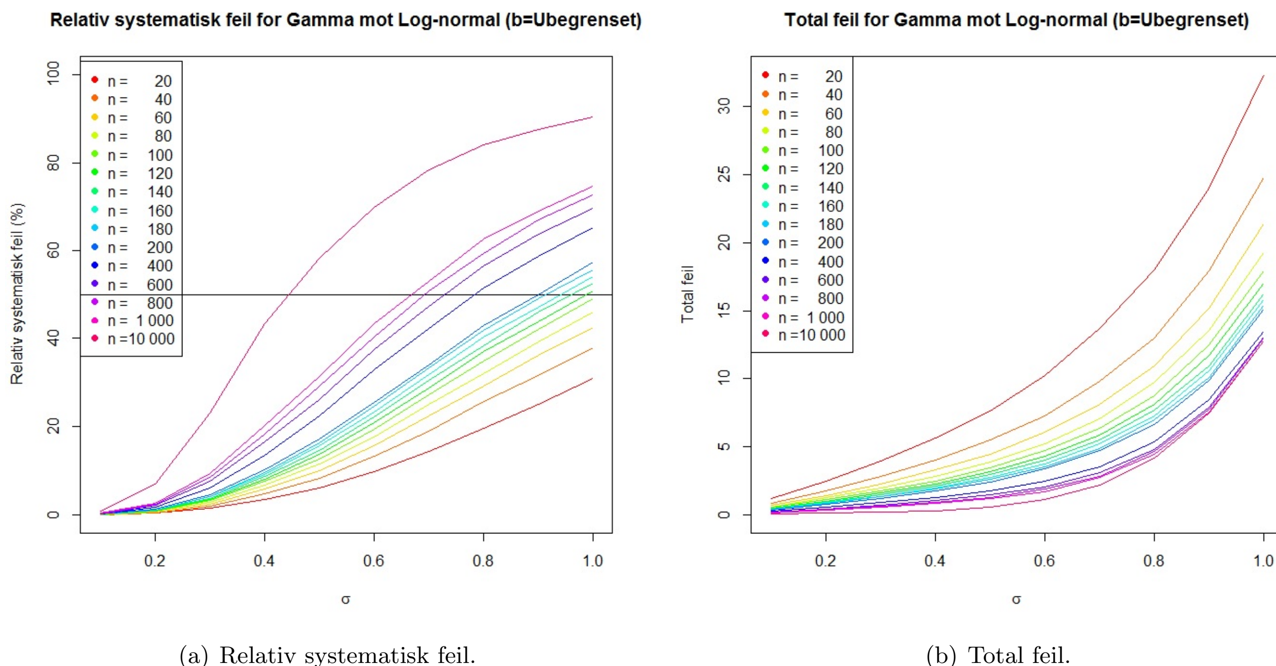
Grunnen til at den systematiske feilen i figur 4.3 (b) går fra å øke til å minke for noen bestemte skadebegrensningsnivåer (fra b=4 til b=10) når σ øker, er at log-normal-modellen har en tyngre hale enn gamma-modellen. Dette ser vi i figur 4.1. Jo større σ, jo tyngre hale får vi for begge modellene, og jo høyere må skadebegrensningen være for at ψ skal være større ψ_0. Det er forøvrig verdt å merke seg at den systematiske feilen er tilnærmet lik 0 for σ under 0.3. Det betyr at gamma-modellen er kapabel til å gjøre gode reserveberegninger når dataene er log-normal-fordelte med lav σ, som vi også påpekte tidligere under figur 4.1.

4.2.2 Relativ systematisk feil og total feil

Hittil har vi kun sett på den systematiske feilen isolert sett. Videre vil vi se på hvordan den systematiske feilen oppfører seg i forhold til total og tilfeldig feil, når ulike variable svinger. Vi vil la antall observasjoner n, skadebegrensning b og σ variere. Vi ser først på et tilfelle uten

skadebegrensning før vi ser på tre tilfeller med lavere og lavere skadebegrensningsnivå (b = 10, 5, 1). Figurene for tilfellene med skadebegrensningsnivå 2, 3, 4, 6, 7, 8 og 9 er i appendix.

Ingen skadebegrensning



(a) Relativ systematisk feil.

(b) Total feil.

Figur 4.4: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal uten skadebegrensning.

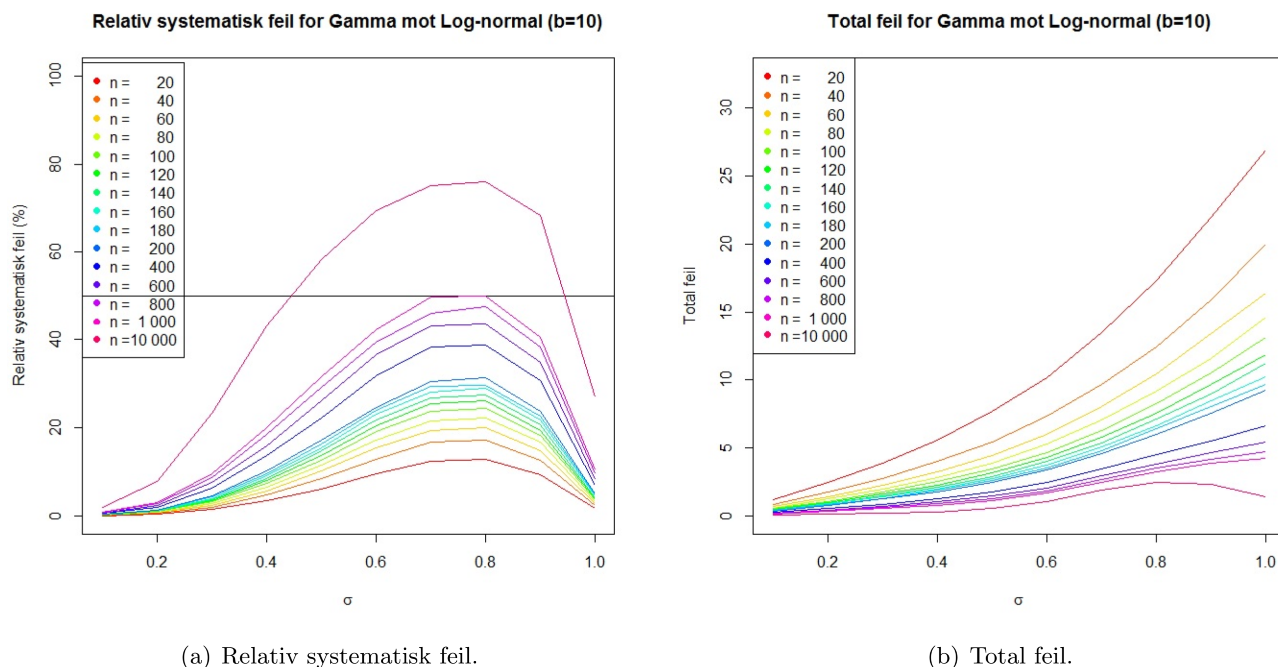
Figur 4.4 (b) viser oss at total feil øker for økende σ, mens total feil minker for økende antall observasjoner. Økningen i total feil ser ut til å være tilnærmet eksponensiell i figur 4.4 (b), fra venstre mot høyre i plottet. Total feil minker for økende antall observasjoner, siden den tilfeldige feilen minker når antall observasjoner øker, samtidig som den systematiske feilen er konstant for gitt σ. Systematisk feil øker derimot når σ øker, og dette forklarer mye hvorfor total feil øker når σ øker.

I figur 4.4 (a) viser den sorte linjen midt i diagrammet 50 %-nivået, der systematisk feil over denne sorte linjen betyr at systematisk feil er større enn tilfeldig feil, og motsatt dersom vi befinner oss under den sorte linjen. Når vi befinner oss på den sorte linjen, er systematisk og tilfeldig feil like store.

Den systematiske feilen utgjør en større og større andel av total feil når σ og antall observasjoner øker. Grunnen til at relativ systematisk feil øker for voksende σ, er at gamma-modellen tilnærmer log-normal-modellen dårligere og dårligere når σ øker. Dermed får vi større systematisk feil. I tillegg øker relativ systematisk feil når antall observasjoner øker, fordi den tilfeldige feilen minker når antall observasjoner øker. Dermed vil den systematiske feilen gå mot 100 % av total feil, og dominere mer og mer når antall observasjoner øker. Altså, relativ

systematisk feil vil få en tydeligere S-form i figur 4.4 (a) når antall observasjoner øker. Vi ser også at for synkende σ, trenger vi flere og flere observasjoner for at relativ systematisk feil skal holde seg konstant.

Høyt skadebegrensningsnivå (b=10)



(a) Relativ systematisk feil.

(b) Total feil.

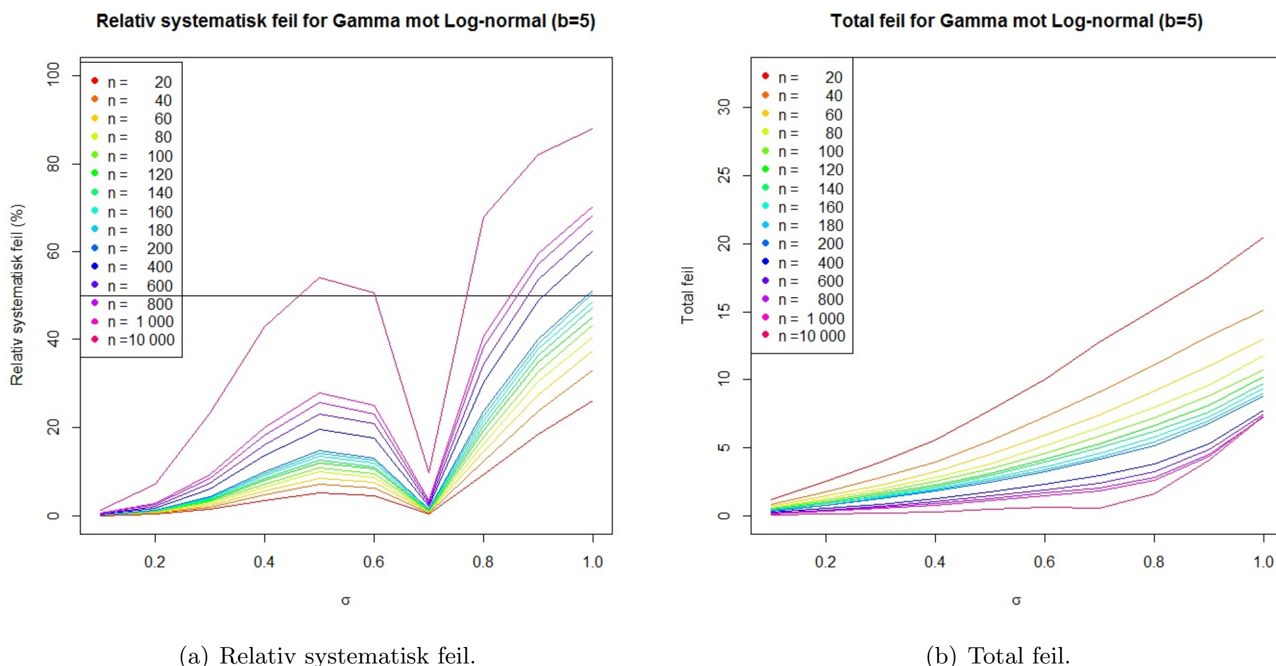
Figur 4.5: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skadebegrensning per polise er 10.

Fra figur 4.5 (b) ser vi at total feil er litt mindre enn for tilfellet uten skadebegrensning. Det skyldes naturlig nok at skadebegrensningen fører til mindre svingninger i skadeutbetalingene som trekkes, og dermed blir det også mindre svingninger i reservene som beregnes. Trenden til den totale feilen er den samme som for tilfellet uten skadebegrensning. Total feil øker for økende σ, mens total feil minker for økende antall observasjoner, bortsett fra tilfellet der vi har 10 000 observasjoner og σ øker fra 0.8 til 1. I dette tilfellet minker faktisk den totale feilen. Dette skyldes at tilfeldig feil er så liten at systematisk feil utgjør en større andel av total feil enn de andre tilfellene med færre observasjoner, i tillegg til at den systematiske feilen minker når σ øker fra 0.8 til 1.0, som vi ser i figur 4.3 (b).

Vi ser fra figur 4.5 (a) at relativ systematisk feil øker når σ øker i intervallet 0 til 0.8, men minker når σ øker i intervallet 0.8 til 1.0. Grunnen til dette er at systematisk feil minker når σ øker i intervallet 0.8 til 1.0, og dermed blir tilfeldig feil mer og mer dominerende, slik at relativ systematisk feil minker. Derfor får vi ikke en S-form for relativ systematisk feil, slik vi fikk i figur 4.4 (a). Ellers ser vi som tidligere at relativ systematisk feil øker for økende antall observasjoner, fordi tilfeldig feil minker. Vi ser også at det stort sett kreves flere observasjoner for at systematisk feil skal være større enn tilfeldig feil, når skadebegrensningsnåvet er 10 i

forhold til tilfellet uten skadebegrensning.

Middels skadebegrensningsnivå (b=5)



(a) Relativ systematisk feil.

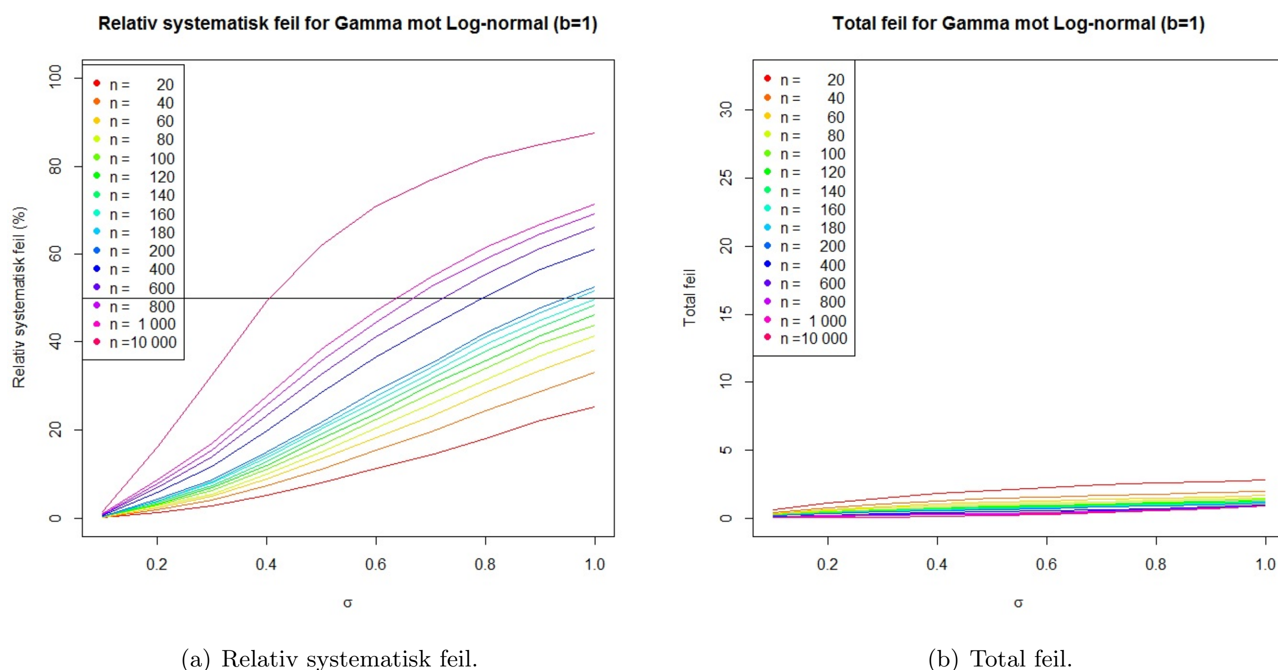
(b) Total feil.

Figur 4.6: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skadebegrensning per polise er 5.

Vi ser fra figur 4.6 (b) at total feil øker for økende σ og minker for økende antall observasjoner, slik vi har sett i de foregående tilfellene. Ved å senke skadebegrensningsnivået fra høyt (b=10) til middels (b=5), ser vi at den totale feilen minker.

Fra figur 4.6 (a) ser vi at relativ systematisk feil øker når antall observasjoner øker, slik vi har sett fra tidligere. Den relative systematiske feilen følger samme mønster som tilfellene med høy og ingen skadebegrensning når σ er lav. Dette skyldes at systematisk feil er liten når σ er liten. Videre ser vi at relativ systematisk feil går fra å minke når σ ligger mellom 0.5 og 0.7, til å øke når σ er større enn 0.7. Årsaken til dette er at ψ_0-ψ skifter fortegn når σ er ca. 0.7, som vi ser i figur 4.3 (a). Når skadebegrensningsnivået senkes fra høyt til middels, kreves det færre observasjoner for at systematisk feil skal være større enn tilfeldig feil for de største verdiene av σ. Ellers gjør lavere skadebegrensningsnivå stort sett at relativ systematisk feil minker. Det vil si at systematisk feil minker mer enn tilfeldig feil for lavere skadebegrensningsnivå.

Lavt skadebegrensningsnivå (b=1)



(a) Relativ systematisk feil.

(b) Total feil.

Figur 4.7: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skade-begrensning per polise er 1.

Fra figur 4.7 (b) ser vi først og fremst at den totale feilen har blitt mye mindre i forhold til de andre tilfellene med høyere skadebegrensning. Ved å sette en streng skadebegrensning sørger vi for mindre variasjoner i skadeutbetalingene vi trekker, og dermed blir variasjonen mindre i reservene, som igjen fører til mindre total feil. Ellers øker total feil for økende σ og minker for økende antall observasjoner, slik vi har sett fra tidligere.

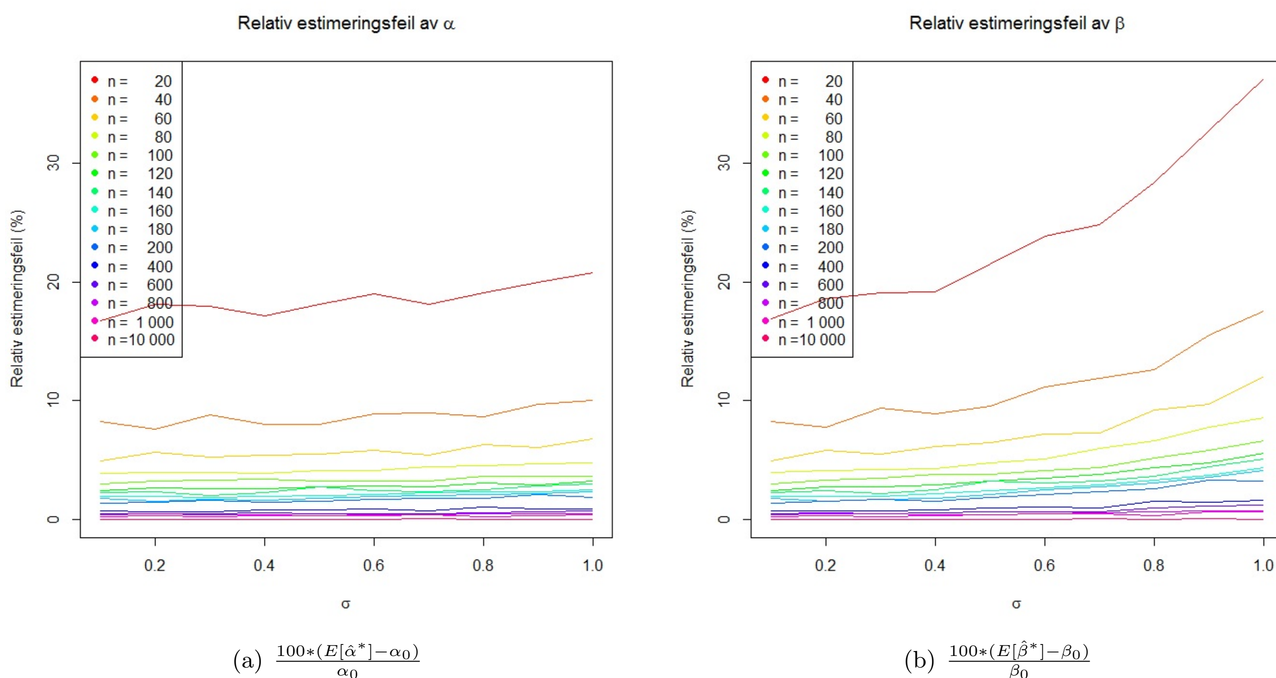
Vi ser fra figur 4.7 (a) at relativ systematisk feil øker for økende σ. Relativ systematisk feil øker også for økende antall observasjoner, slik hovedtrenden også har vært i de foregående tilfellene. Det kreves flere observasjoner for at den systematiske feilen skal overgå den tilfeldige feilen når σ minker. Dette skyldes at den systematiske feilen går mot 0 når σ minker, som vi ser i figur 4.3 (b). Når antall observasjoner øker, får vi en tydeligere S-form for relativ systematisk feil når vi plotter mot σ, slik tilfellet også var uten skadebegrensning. Dette skyldes at systematisk feil er veldig liten for lav σ og øker når σ øker, samtidig som den systematiske feilen dominerer tilfeldig feil når vi har mange observasjoner.

Effekten av strengere skadebegrensning på relativ systematisk feil er blandet. Sett i forhold til tilfellet uten skadebegrensning, kreves det her flere observasjoner for at systematisk feil skal utgjøre mer enn 50 % av den totale feilen. Dersom vi sammenligner relativ systematisk feil for tilfellet med lav skadebegrensning mot middels/høy skadebegrensningsnivå, ser vi at det blir vanskelig å trekke kun én konklusjon. Dette skyldes at trenden til den systematis-ke feilen er avhengig av om skadebegrensningsnivået er lavt, middels eller høyt [se figur 4.3

(b)]. Stort sett gjør lavere skadebegrensningsnivå at vi trenger flere observasjoner for at systematisk feil skal overgå tilfeldig feil, fordi systematisk feil minker relativt mest for lavere skadebegrensningsnivå.

4.2.3 Parameterfeil

Når vi skal beregne reserver, har vi et underliggende parametersett som påvirker skadeutbetalingene som trekkes. Dermed vil estimeringsfeil av parametere bidra til den tilfeldige feilen. Derfor er det vesentlig å se på om vi over- eller underestimerer parametere, og hvor mye parameterestimatene varierer når vi vil undersøke feil i beregningen av reserve. Her vil vi se på hvordan varierende σ og antall observasjoner n påvirker parameterestimatene, ved å se på relativ forventet estimeringsfeil av parameterene α og β, samt relativt standardavvik til parameterestimatene.



(a) $\frac{100*(E[\hat{\alpha}^*]-\alpha_0)}{\alpha_0}$ (b) $\frac{100*(E[\hat{\beta}^*]-\beta_0)}{\beta_0}$

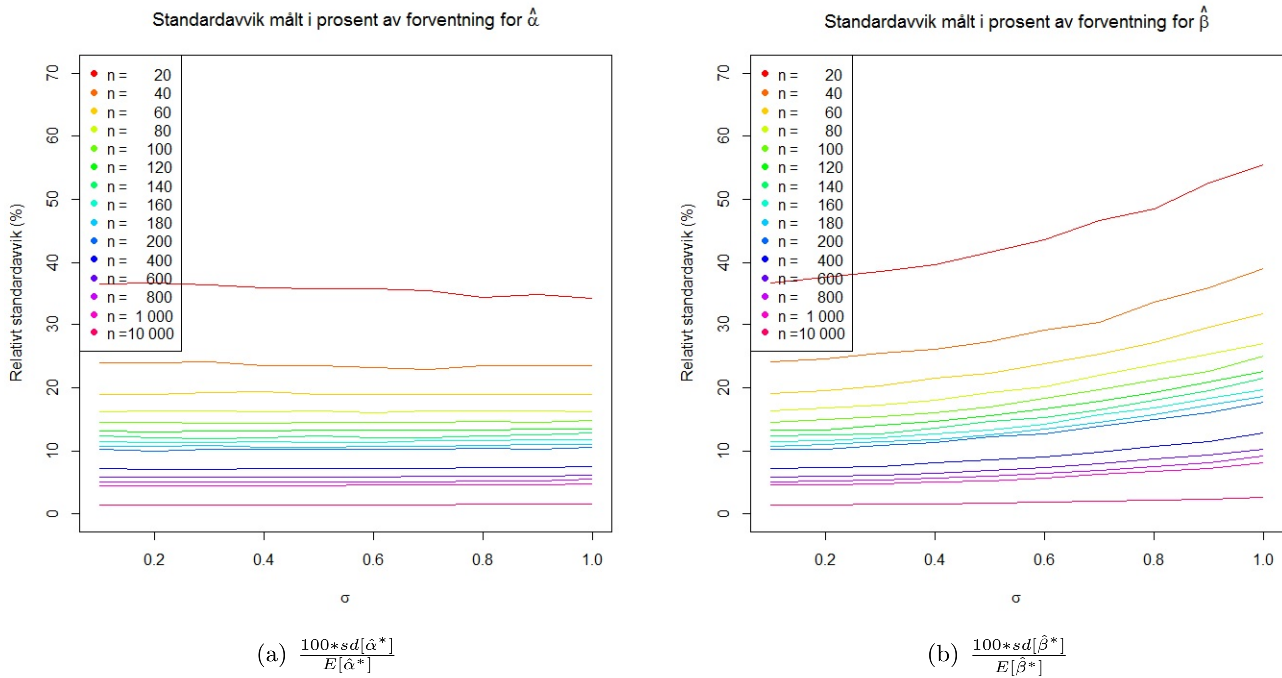
Figur 4.8: Relativ forventet estimeringsfeil av α og β, der $\hat{\alpha}^*$ og $\hat{\beta}^*$ er $\mathrm{m}_b = 10\ 000$ simulerte parameterestimater.

Fra figur 4.8 (a) ser vi at α overestimeres, men den relative estimeringsfeilen er nesten konstant for varierende σ og gitt n. Det er dog en liten tendens til at den relative overestimeringsfeilen er stigende når σ øker. Vi ser at estimeringsfeilen blir mindre når antall observasjoner øker. Dette bekrefter det vi har sett tidligere, at den tilfeldige feilen minker for økende antall observasjoner.

Vi ser fra figur 4.8 (b) at β også overestimeres, og det er en klar tendens til økende overestimering når σ øker. Estimeringsfeilen minker også for økende antall observasjoner. Videre ser

vi at β overestimeres mer enn α, spesielt for de største verdiene av σ. Dette skyldes blant annet at ML-estimatet $\hat{\beta}$ er en funksjon av $\hat{\alpha}$.

Forventningen til gamma-modellen er gitt ved $\frac{\alpha}{\beta}$. Det betyr at skadeutbetalingene vi trekker fra gamma-modellen vil øke i størrelse når α øker, og motsatt om β øker. Siden β stort sett overestimeres mer enn α, betyr det at vi underestimerer skadeutbetalingene og reservene under gamma-modellen. Dette bidrar til tilfeldig feil, og siden β overestimeres relativt mer enn α når σ øker, vil tilfeldig feil øke for økende σ.



(a) $\frac{100 * sd[\hat{\alpha}^*]}{E[\hat{\alpha}^*]}$

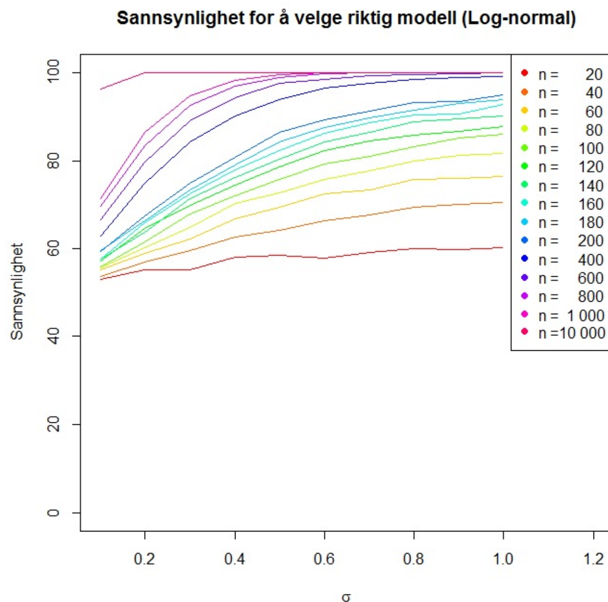
(b) $\frac{100 * sd[\hat{\beta}^*]}{E[\hat{\beta}^*]}$

Figur 4.9: Relativt standardavvik for $\hat{\alpha}$ og $\hat{\beta}$, der $\hat{\alpha}^*$ og $\hat{\beta}^*$ er $m_b = 10\ 000$ simulerte parameterestimater.

Figur 4.9 beskriver relativ usikkerhet i estimeringen av α og β. Den relative usikkerheten i estimeringen av α er tilnærmet konstant for varierende σ og gitt n. Dersom n øker, minker usikkerheten i estimeringen av α. Det samme gjelder for β, og fører til at tilfeldig feil minker for økende n. Videre ser vi at relativ usikkerhet i estimeringen av β øker når σ øker. Dette bidrar til at tilfeldig feil øker for økende σ. Forøvrig ser vi at den relative usikkerheten i estimeringen av β er større enn for α, som blant annet skyldes at ML-estimatet $\hat{\beta}$ er en funksjon av $\hat{\alpha}$.

4.2.4 Sannsynlighet for å velge riktig modell

Hittil har valgt modell vært gamma-modellen. Det er derimot rimelig å tenke seg at vi ikke ville valgt gamma-modellen når vi har mange observasjoner. Derfor vil vi her studere sannsynligheten for å velge korrekt modell, log-normal, mens alternativet er gamma-modellen. Vi ser på varierende antall observasjoner og varierende σ.

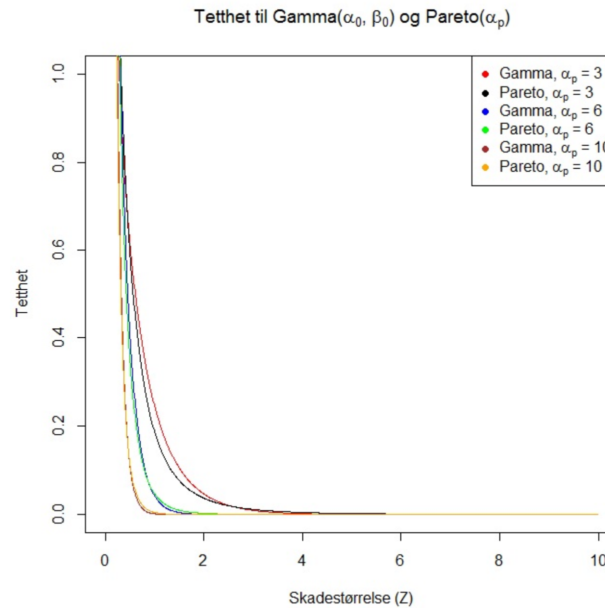


Figur 4.10: Sannsynlighet for å velge riktig modell når korrekt modell er Log-normal.

Fra figur 4.10 ser vi at sannsynligheten for å velge korrekt modell (log-normal) øker for økende antall observasjoner. Denne sannsynligheten øker også for økende σ. For lav σ er det større sannsynlighet for å velge gamma-modellen enn for stor σ. Det skyldes at gamma-modellen gjør en bedre tilpasning når dataene er log-normal-fordelte med lav σ enn stor σ. Det er over 50 % sannsynlighet for å velge korrekt modell for alle tilfellene, men vi ser det kreves en del observasjoner for at denne sannsynligheten skal være over 90 %. Det er dermed ikke gitt at vi velger riktig modell dersom valget står mellom gamma- og log-normal-modellen, men det er størst sannsynlighet for å velge korrekt modell. For 10 000 observasjoner velger vi nesten alltid riktig modell.

4.3 Gamma-modellen mot Pareto-modellen

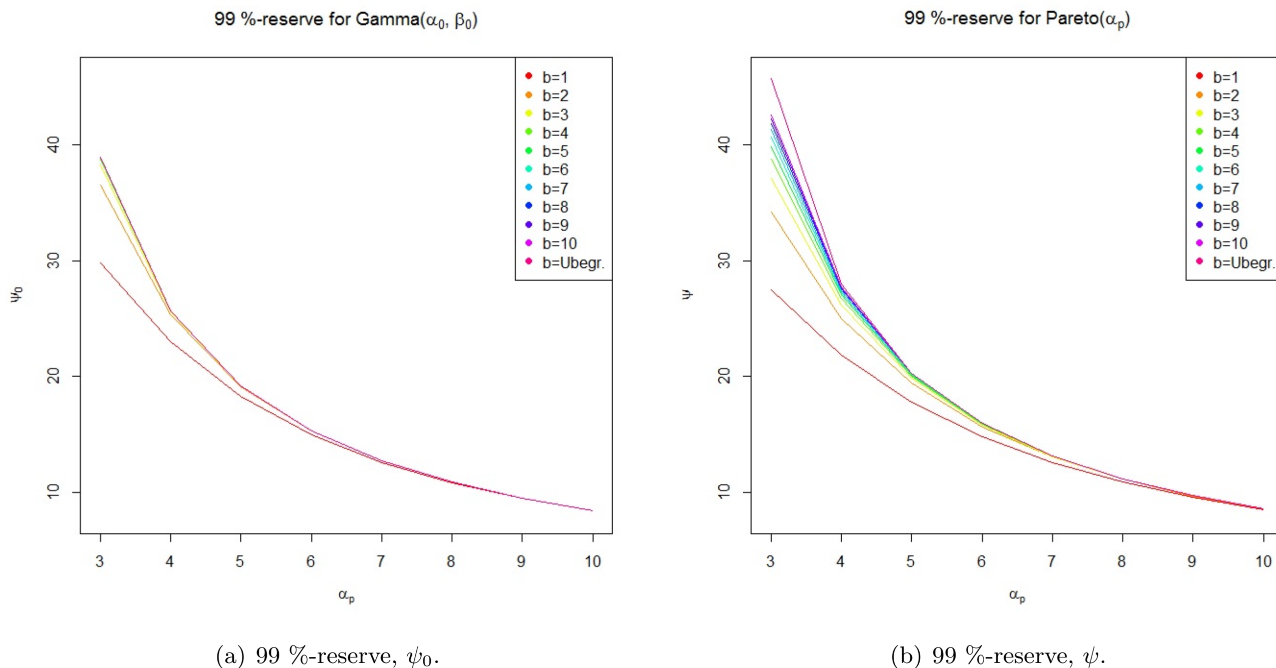
Vi ser først på fordelingene til Gamma(α_0, β_0) og Pareto(α_p) for å få et innblikk i størrelsen på skadeutbetalingene vi trekker i reserveberegningene.



Figur 4.11: Tettheten til Gamma(α_0, β_0) og Pareto(α_p) for $\alpha_p = 3, 6, 10$, der skadestørrelsene f.eks. er i millioner NOK.

Fra figur 4.11 ser vi at pareto-modellen har generelt mye tyngre hale enn gamma-modellen. Skadeutbetalingene under pareto-modellen kan bli veldig store, og vi ser gamma-modellen sliter med å tilpasse pareto-modellen når α_p er lav. Når α_p minker, og samtidig er større enn 2, vil standardavviket øke for pareto-modellen. Dersom α_p er 2 eller mindre er ikke standardavviket definert under pareto-modellen. Dermed blir det lettere for gamma-modellen å tilpasse pareto-fordelte data når α_p øker. Videre vil vi se på 99 %-reservene, ψ_0 og ψ, og effekten av varierende skadebegrensning b og α_p.

4.3.1 Systematisk feil

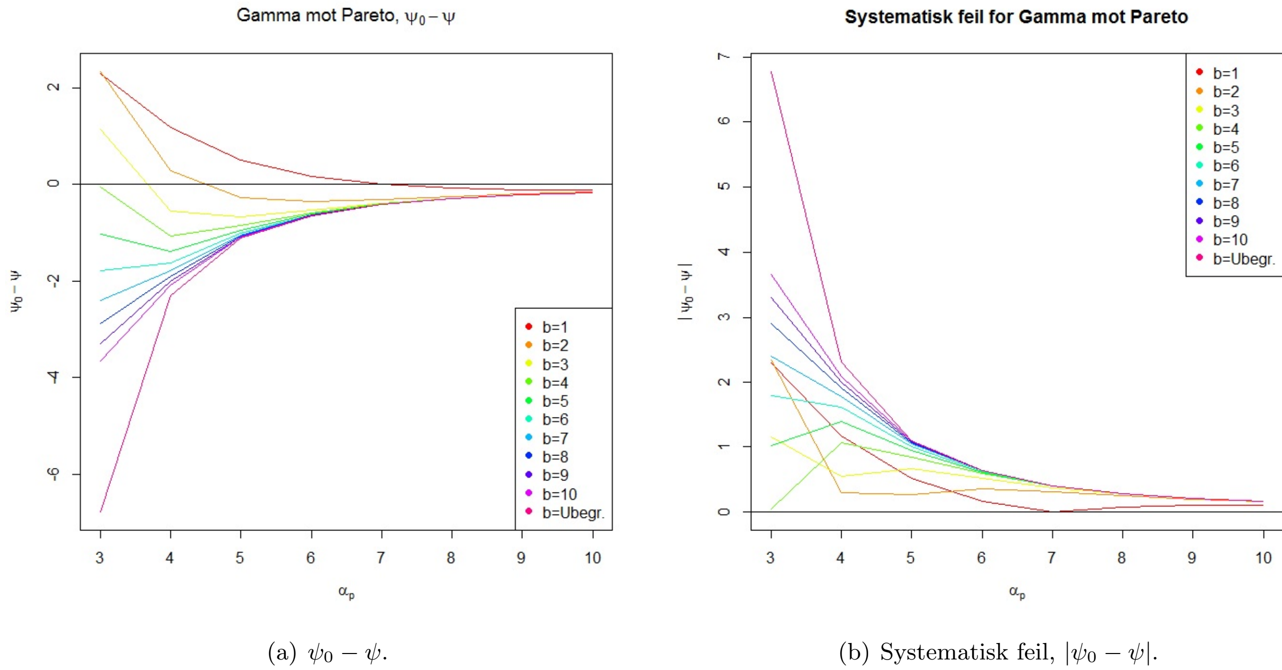


(a) 99 %-reserve, ψ_0.

(b) 99 %-reserve, ψ.

Figur 4.12: 99 %-reserve for ulike skadebegrensninger (b) under modell Gamma(α_0, β_0) og Pareto(α_p).

Vi ser fra figur 4.12 at 99 %-reserven er større under pareto-modellen enn under gamma-modellen når α_p er lav og skadebegrensningsnivået er høyt. Samtidig varierer ψ mer enn ψ_0, men reservene ser ut til å være mer like når α_p øker. Dette bekrefter det vi så i figur 4.11, at gamma-modellen treffer bedre når α_p er stor. Vi ser også at ψ er mindre enn ψ_0 for lavt skadebegrensningsnivå og lav α_p. Begge reservene er fallende for økende α_p og for minkende skadebegrensningsnivå. Vi vil se nærmere på avvik mellom ψ_0 og ψ ved å se på den systematiske feilen.



(a) $\psi_0 - \psi$.

(b) Systematisk feil, $|\psi_0 - \psi|$.

Figur 4.13: $\psi_0 - \psi$ (a) og systematisk feil (b) for Gamma mot Pareto for forskjellige skadebegrensninger per polise.

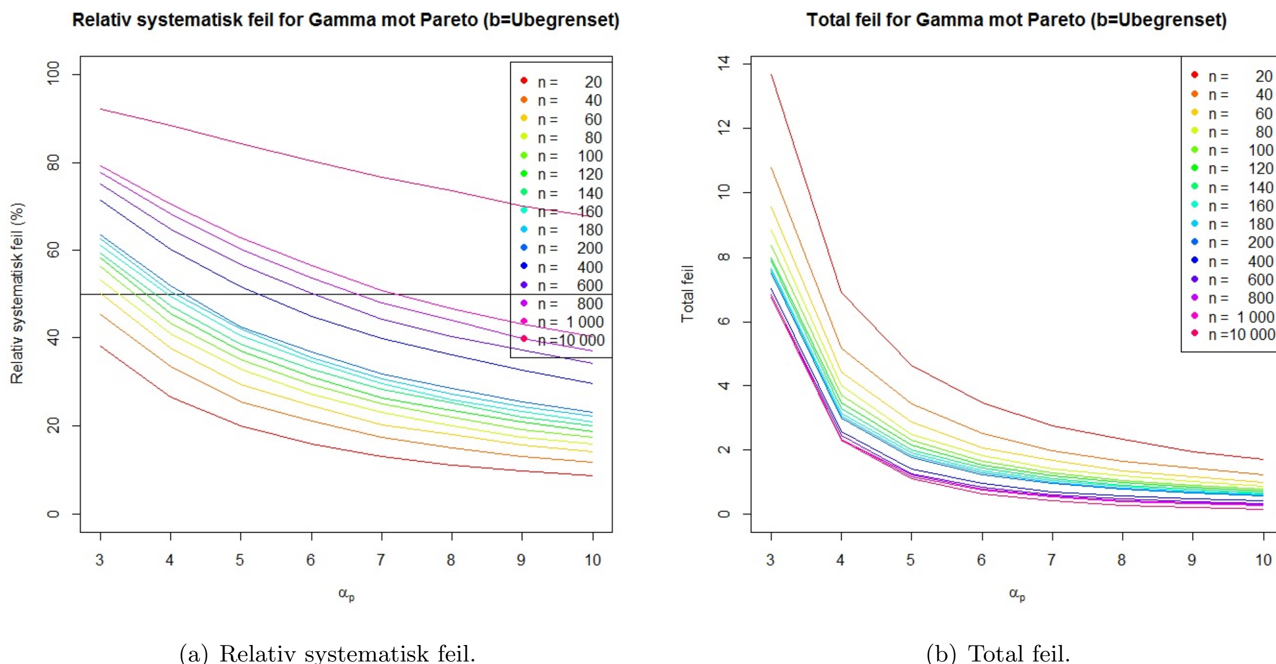
Fra figur 4.13 (a) ser vi at ψ_0 kun er større enn ψ for skadebegrensningsnivå 1 og 2 når α_p er henholdsvis mindre enn 5 og 8. Ellers er ψ større enn ψ_0 for alle de andre skadebegrensningsnivåene. Det er en klar tendens til underestimering av reserve under gamma-modellen, spesielt når α_p er lav.

Vi ser fra figur 4.13 (b) at den systematiske feilen stort sett øker for økende skadebegrensningsnivå og for minkende α_p. Noen av de laveste skadebegrensningsnivåene gir et blandet bilde av den systematiske feilen, der den systematiske feilen i noen tilfeller går fra å øke til å minke for økende α_p. Dette skyldes at pareto-modellen har en tyngre hale enn gamma-modellen. Når α_p minker, får begge modellene tyngre haler, og jo høyere må skadebegrensningsnivået være for at ψ skal være større enn ψ_0. Ellers ser det ut til at den systematiske feilen går mot 0 når α_p er stor. Det betyr at gamma-modellen klarer å gjøre gode reserveberegninger når underliggende data er pareto-fordelte med stor α_p.

4.3.2 Relativ systematisk feil og total feil

Vi vil se på relativ systematisk feil og total feil for varierende antall observasjoner n, skadebegrensning b og α_p. Først ser vi på et tilfelle uten skadebegrensning, før vi ser på tre tilfeller med lavere skadebegrensningsnivå ($b = 10,\ 5,\ 1$). Figurene for tilfellene med skadebegrensningsnivå 2, 3, 4, 6, 7, 8 og 9 er i appendix.

Ingen skadebegrensning



(a) Relativ systematisk feil.

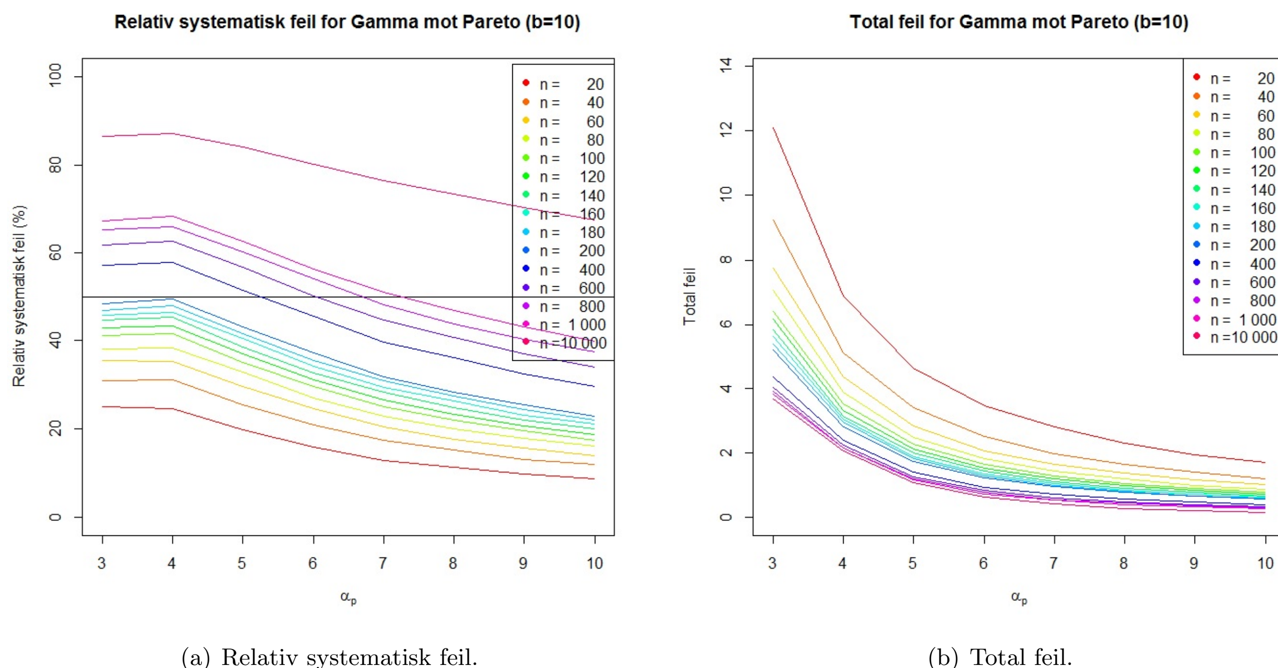
(b) Total feil.

Figur 4.14: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto uten skadebegrensning.

Figur 4.14 (b) viser at total feil minker for økende α_p og for økende antall observasjoner. Total feil avhenger av antall observasjoner gjennom tilfeldig feil, der vi ser at total feil varierer en del avhengig av om vi har få eller mange observasjoner. Når α_p øker i pareto-fordelingen, vil standardavviket og systematisk feil minke, som igjen fører til at den totale feilen faller fra venstre mot høyre i figur 4.14 (b). Den totale feilen minker relativt kraftig når α_p øker fra 3 til 4-5, mens fra 5 og oppover ser det ut til at den totale feilen flater ut mer og mer. Dette skyldes at den systematiske feilen i tilfellet uten skadebegrensning i figur 4.13 (b), flater ut og blir mindre for økende α_p.

Den relative systematiske feilen i figur 4.14 (a) er fallende fra venstre mot høyre i likhet med den totale feilen. Det er store forskjeller på relativ systematisk feil når antall observasjoner varierer. Vi ser at for få observasjoner og stor α_p dominerer den tilfeldige feilen den systematiske feilen. Altså, tilfeldig feil er langt større enn den systematiske feilen. Det er stikk motsatt når α_p er liten og vi har mange observasjoner, der den systematiske feilen dominerer. Grunnen til dette er at når α_p er liten får vi store standardavvik i pareto-modellen, og da er det vanskelig for gamma-modellen å tilnærme pareto-fordelte data. Derfor får vi større systematisk feil når α_p minker, og vi ser fra figur 4.14 (a) at den relative systematiske feilen også øker. Eller sagt på en annen måte, systematisk feil øker mer enn tilfeldig feil for minkende α_p. Dette betyr at det kreves færre og færre observasjoner for å holde relativ systematisk feil konstant når α_p minker. Videre vil vi se på effekten av å innføre skadebegrensning per polise.

Høyt skadebegrensningsnivå (b=10)



(a) Relativ systematisk feil.

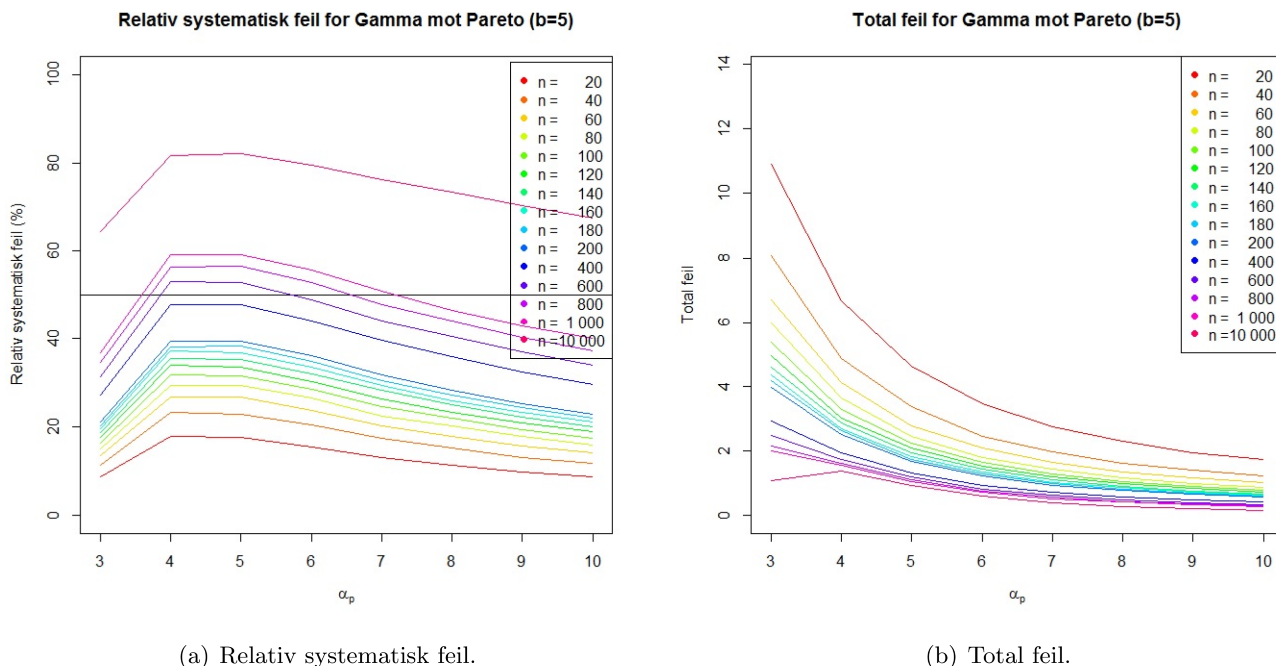
(b) Total feil.

Figur 4.15: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 10.

Vi ser fra figur 4.15 (b) at total feil minker ved å innføre skadebegrensning per polise. Skadebegrensningen gjør at det blir mindre avvik mellom skadeutbetalingene vi trekker, og dermed blir det mindre differanse mellom reservene som utgjør den totale feilen. Ellers er den totale feilen minkende når α_p øker og når antall observasjoner øker, slik vi har sett fra tidligere.

Fra figur 4.15 (a) ser vi at relativ systematisk feil stort sett minker når α_p øker, bortsett fra når α_p øker fra 3 til 4. Vi ser fra figur 4.13 (b) at den systematiske feilen minker for tilsvarende α_p. Det betyr at den systematiske feilen minker mindre enn den tilfeldige feilen når α_p øker fra 3 til 4. Ved å innføre skadebegrensning ser vi at det stort sett kreves flere observasjoner for at systematisk feil skal overgå tilfeldig feil, spesielt for lav α_p. Forøvrig ser vi at relativ systematisk feil øker når antall observasjoner øker, slik vi har sett tidligere.

Middels skadebegrensningsnivå (b=5)



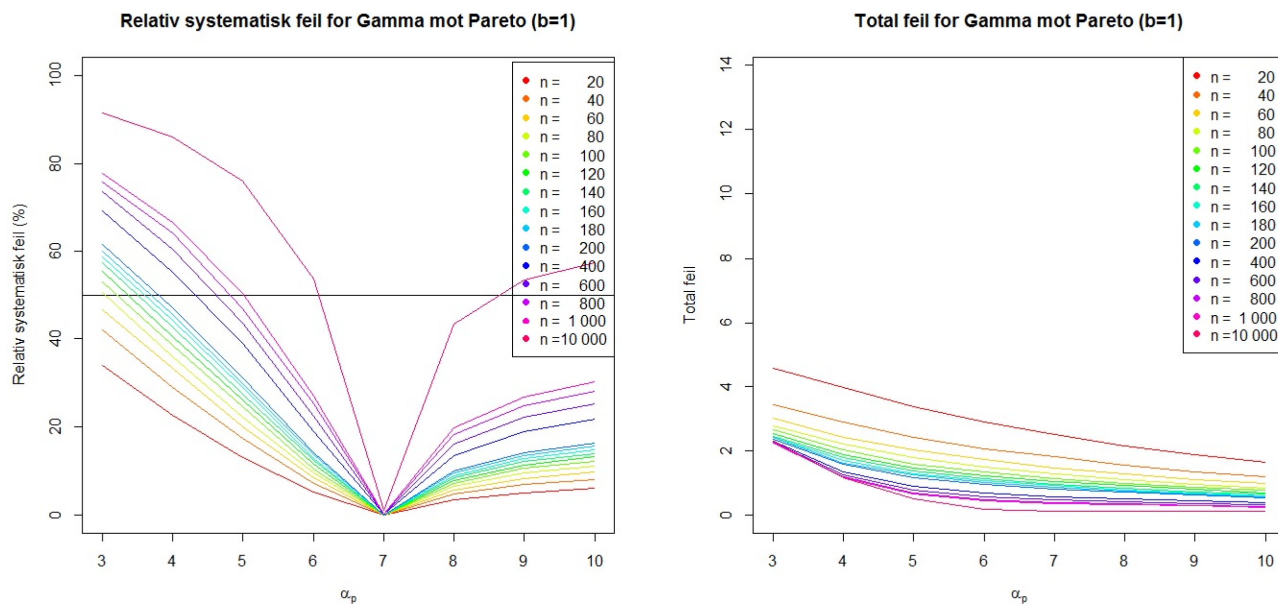
(a) Relativ systematisk feil. (b) Total feil.

Figur 4.16: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 5.

Fra figur 4.16 (b) ser vi at total feil minker ytterligere ved å senke skadebegrensningsnivået fra 10 til 5. Den totale feilen minker for økende α_p, bortsett fra når α_p øker fra 3 til 4 og antall observasjoner er lik 10 000. Årsaken til dette er at systematisk feil øker når α_p øker fra 3 til 4, samtidig som systematisk feil dominerer. Det gjør at total feil øker i dette tilfellet i motsetning til den generelle trenden, der total feil minker for økende α_p og for økende antall observasjoner.

Vi ser fra figur 4.16 (a) at den relative systematiske feilen minker for økende α_p, bortsett fra når α_p øker fra 3 til 4. Dette skyldes at den systematiske feilen øker for tilsvarende α_p. Ved å senke skadebegrensningsnivået fra 10 til 5 ser vi at den relative systematiske feilen ligger lavere i diagrammet. Det betyr at det kreves flere observasjoner for at systematisk feil skal overgå den tilfeldige feilen. Ellers ser vi at den systematiske feilen utgjør større andel av total feil når antall observasjoner øker, slik vi har sett tidligere.

Lavt skadebegrensningsnivå (b=1)



(a) Relativ systematisk feil.

(b) Total feil.

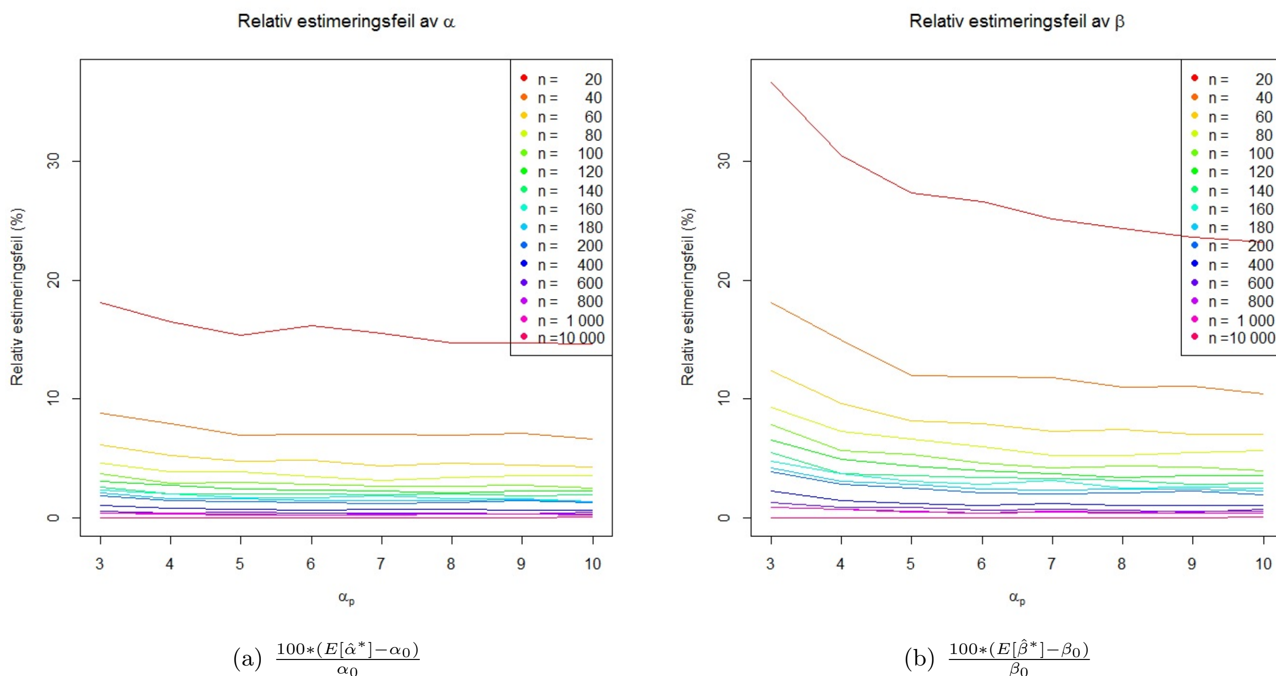
Figur 4.17: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 1.

Vi ser fra figur 4.17 (b) at total feil minker ved å senke skadebegrensningsnivået ytterligere fra 5 til 1. I tillegg minker total feil for økende α_p og for økende antall observasjoner, slik vi har sett i de andre tilfellene. Den totale feilen ser ut til å flate ut når α_p øker. Dette skyldes at vi gjør mindre systematisk feil for økende α_p. Når α_p øker, minker standardavviket til dataene, som igjen gjør det lettere for gamma-modellen å tilnærme disse dataene. Dermed vil den systematiske feilen bli forsvinnende liten for store α_p. Dette ser vi også i figur 4.13. Når den systematiske feilen er tilnærmet lik 0, blir den totale feilen tilnærmet lik tilfeldig feil. Den tilfeldige feilen blir kun forsvinnende liten for et stort antall observasjoner, slik at total feil går mot 0 når α_p er stor og vi har mange observasjoner.

Fra figur 4.17 (a) ser vi at den relative systematiske feilen faller for økende α_p i intervallet 3 til 7, for så å øke igjen i intervallet 7 til 10. Grunnen til dette ser vi i figur 4.13 (a). Når skadebegrensningsnivået er 1, ser vi at ψ_0 er større enn ψ for α_p mindre enn 7, og motsatt for α_p større enn 7. Ved α_p rundt 7 er ψ_0 ca. lik ψ. Ved å senke skadebegrensningsnivået fra 5 til 1, ligger den relative systematiske feilen lavere i diagrammet, bortsett fra når α_p er 3 og 4. Det skyldes at den systematiske feilen er større for skadebegrensningsnivå 1 enn 5, gitt at α_p er 3. Når α_p er 4, er de ca. like store. Hovedtrenden er dog at relativ systematisk feil minker når skadebegrensningsnivået minker. Ellers har vi som tidligere at relativ systematisk feil øker når antall observasjoner øker.

4.3.3 Parameterfeil

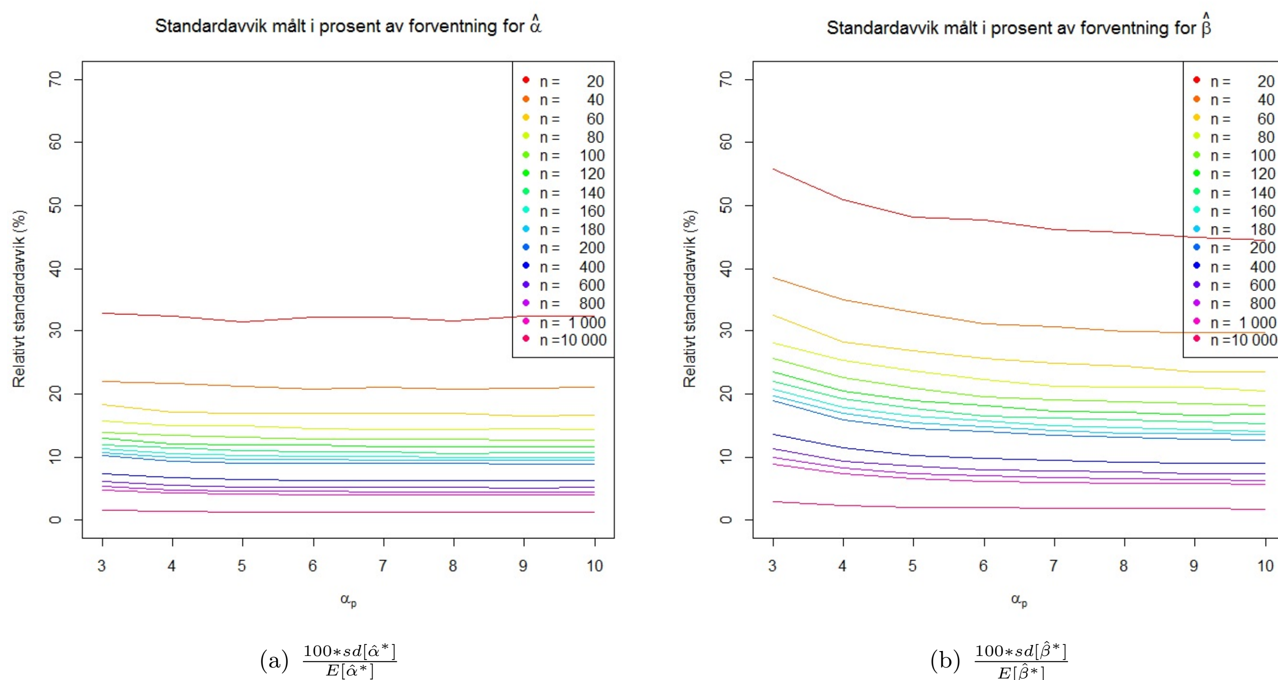
Vi ser på parameterfeilen for å undersøke svingningene bak den tilfeldige feilen. Det er interessant å se om vi over- eller underestimerer α og β under gamma-modellen, og hvor stor usikkerhet det er i parameterestimatene. Her vil vi se på hvilken effekt varierende α_p og antall observasjoner n har på parameterestimatene.



(a) $\frac{100*(E[\hat{\alpha}^*]-\alpha_0)}{\alpha_0}$ (b) $\frac{100*(E[\hat{\beta}^*]-\beta_0)}{\beta_0}$

Figur 4.18: Relativ forventet estimeringsfeil av α og β, der $\hat{\alpha}^*$ og $\hat{\beta}^*$ er $\text{m}_b = 10\ 000$ simulerte parameterestimater.

Vi ser fra figur 4.18 (a) at vi overestimerer både α og β, og at overestimeringen blir mindre når α_p øker. Vi overestimerer β relativt mer enn α, spesielt når α_p er liten. Det betyr at vi underestimerer skadeutbetalingene og reservene, siden forventningen til gamma-modellen er gitt ved $\frac{\alpha}{\beta}$. Dette bidrar til tilfeldig feil, og siden overestimeringen av β øker mer enn overestimeringen av α når α_p minker, får vi større tilfeldig feil for minkende α_p. Ellers ser vi at estimeringsfeilen minker for begge parameterene når antall observasjoner øker, slik at den tilfeldige feilen også minker når antall observasjoner øker.



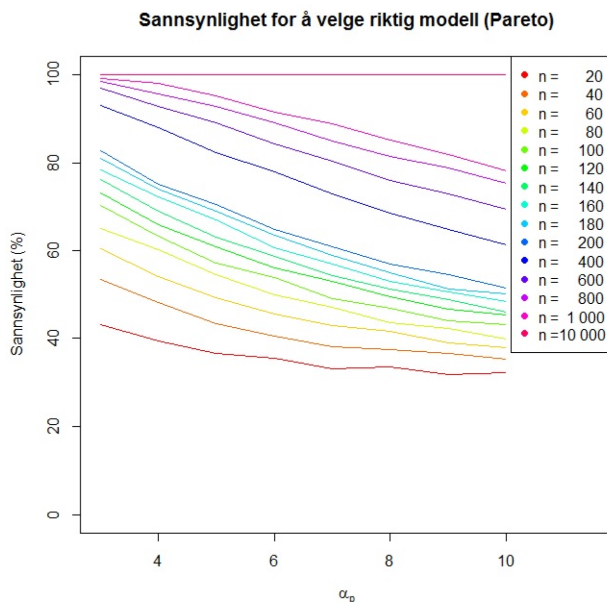
(a) $\frac{100*sd[\hat{\alpha}^*]}{E[\hat{\alpha}^*]}$ (b) $\frac{100*sd[\hat{\beta}^*]}{E[\hat{\beta}^*]}$

Figur 4.19: Relativt standardavvik for $\hat{\alpha}$ og $\hat{\beta}$, der $\hat{\alpha}^*$ og $\hat{\beta}^*$ er $m_b = 10\ 000$ simulerte parameterestimater.

Ved å se på relativ usikkerhet i estimeringen av α og β i figur 4.19, ser vi at den relative usikkerheten rundt estimeringen av α er tilnærmet konstant for varierende α_p og gitt antall observasjoner. Relativ usikkerhet i estimeringen av β øker når α_p minker, og er større for β enn for α. Dette bidrar til å øke tilfeldig feil når α_p minker. Usikkerheten i begge parameterestimatene er forøvrig minkende når antall observasjoner øker, slik at tilfeldig feil også minker når antall observasjoner øker.

4.3.4 Sannsynlighet for å velge riktig modell

Her vil vi studere sannsynligheten for å velge korrekt modell, pareto, mens alternativet er gamma-modellen. Vi ser på varierende antall observasjoner og varierende α_p.



Figur 4.20: Sannsynlighet for å velge riktig modell når korrekt modell er Pareto.

Fra figur 4.20 ser vi at sannsynligheten for å velge riktig modell minker for økende α_p. Når α_p øker, vil standardavviket i underliggende pareto-data minke, og gamma-modellen gjør en bedre og bedre tilpasning av dataene. Derfor øker sannsynligheten for å velge gamma-modellen når α_p minker. Vi ser at sannsynligheten for å velge korrekt modell øker når antall observasjoner øker, og for få observasjoner er det større sannsynlighet for å velge gamma-modellen enn pareto-modellen. Totalt sett kreves det mange observasjoner for at sannsynligheten for å velge korrekt modell skal overstige 90 %. Dersom vi har 10 000 observasjoner velger vi riktig modell med 100 % sannsynlighet.

4.4 Oppsummering

I dette kapittelet har vi sett på ulike typer feil innenfor feilreservering når valgt modell (gamma) for erstatningskrav er feil, der korrekt modell er log-normal-modellen eller pareto-modellen. Vi har studert effekten på systematisk, tilfeldig og total feil for varierende skadebegrensning per polise b, antall observasjoner n og parametere i underliggende korrekt modell (log-normal: σ, pareto: α_p).

Vi har sett at økende antall observasjoner fører til at tilfeldig feil minker, mens den systematiske feilen er uavhengig av antall observasjoner, slik at total feil minker når antall observasjoner øker. Systematisk feil utgjør en større og større andel av total feil når antall observasjoner øker, slik at systematisk feil dominerer tilfeldig feil når vi har et høyt antall observasjoner.

Standardavviket i underliggende data fra korrekt modell øker når parameter σ øker i log-normal-modellen og når α_p minker i pareto-modellen. Når standardavviket i underliggende

data minker, minker både tilfeldig og systematisk feil, der den systematiske feilen minker mest. Dette fører til at total feil minker når standardavviket i underliggende data minker.

Både systematisk og tilfeldig feil minker når skadebegrensningsnivået per polise minker, men vi får størst effekt på systematisk feil. Dette fører til at total feil minker når skadebegrensningen per polise minker. Det betyr at risikoen rundt reservering begrenses ved å innføre skadebegrensning, og det er en god måte å begrense risikoen på dersom modellen for erstatningskravene skulle være feil.

Det er verdt å nevne at de aller laveste skadebegrensningsnivåene gir et blandet bilde av reserveringsfeil når standardavviket i underliggende modell ikke er på de laveste nivåene. I disse tilfellene blir en stor andel av skadeutbetalingene påvirket av skadebegrensningen, og det er ikke hensikten med å innføre skadebegrensninger. Hensikten med skadebegrensninger er å beskytte seg mot de aller største skadene. Derfor ser vi bort i fra tilfellene der skadebegrensningen er satt lavt samtidig som standardavvik i underliggende data ikke er på de laveste nivåene.

I begge tilfellene der korrekt modell er log-normal- og pareto-modellen, overestimerer vi α og β under gamma-modellen. β overestimeres relativt mer enn α, slik at skadeutbetalinger og reserver underestimeres. Det er også større usikkerhet rundt estimeringen av β enn α, blant annet fordi ML-estimatet $\hat{\beta}$ er en funksjon av $\hat{\alpha}$. Vi overestimerer parameterene og underestimerer reserven, fordi gamma-modellen har lettere hale enn log-normal- og pareto-modellen.

Sannsynligheten for å velge riktig modell øker for økende antall observasjoner og for minkende standardavvik i underliggende data. Det kreves minst et par hundre observasjoner for å kunne velge riktig modell med over 90 % sannsynlighet. Derfor er det ikke helt urealistisk å velge feil modell om vi ikke har mye data. Vi velger riktig modell med nesten 100 % sannsynlighet for alle tilfeller når vi har 10 000 observasjoner.

Kapittel 5

Konklusjon

I denne oppgaven har vi introdusert en ny algoritme for reserveberegninger i skadeforsikring og vist hvordan vi kan beregne reserver opptil 240 ganger raskere i R ved å vektorisere, parallellisere og kombinere C++- og R-kode. Dette gjør at forsikringsselskaper kan øke hastighet og presisjon på reserveberegninger og bli mer effektive.

Vi har vist at parallellisering er et kraftig optimaliseringsverktøy, der parallellisering fungerer på de fleste simuleringsproblemer, så fremt et problem kan deles opp i uavhengige biter. Dette er typisk for problemer som krever MC-simuleringer, og forhåpentligvis kan denne oppgaven hjelpe andre til å gjøre utvidede analyser i R som ikke ville vært mulige å gjøre uten parallellisering. En aktuell problemstilling innenfor skadeforsikring er f. eks. optimalisering av dynamiske porteføljer.

Raskere reserveberegninger gjorde det mulig å studere feilreservering i flere tusen scenarioer. Ved å velge feil modell for erstatningskravene knytter det seg ekstra risiko til reserveberegninger. Det er spesielt viktig å velge riktig modell om vi har stor variasjon i underliggende data, fordi feilen knyttet til valg av feil modell dominerer ofte i slike tilfeller. Dersom vi velger en modell som har for lett hale i forhold til reelle krav underestimerer vi reserven, og motsatt dersom valgt modell har for tung hale. Det er dog bedre å overestimere enn å underestimere reserve, fordi et forsikringsselskap må ha nok reserver for å ikke gå konkurs.

Antall observasjoner er viktig for valg av modell og estimering av parametere. Risikoen for å velge feil modell er liten dersom vi har noen hundretalls observasjoner, og dette fører også til lav estimeringsfeil av parametere og reserve. Dersom vi ikke har mange observasjoner, dominerer ofte estimeringsfeilen, og det er stort behov for å begrense risiko. Ved å inngå kontrakter om skadebegrensninger med andre selskaper kan et forsikringsselskap begrense risikoen betraktelig, der skadebegrensningskontrakter er spesielt effektive dersom vi har valgt feil modell. Siden forsikringsfaget handler om å kontrollere risiko, er skadebegrensningskontrakter et viktig verktøy i skadeforsikring.

Kapittel 6

Videre arbeid

I kalkuleringen av reserver tar det mest tid å trekke tilfeldige skadeutbetalinger. Ved å trekke tilfeldige variable ved GPU-programmering kan vi oppnå betraktelig raskere beregning av reserver, der GPU står for *Graphics Processing Unit*. Det vil si å trekke variable fra skjermkortet, der vi utnytter at skjermkortet har flere hundre kjerner som kan kjøre parallellt. Dermed er det mye tid å spare på simuleringer av reserve om man får til dette, istedenfor at kun CPU (prosessor) skal generere tilfeldige variable. GPU-programmering er relativt ny teknologi, og derfor er det ikke alle skjermkort som kan brukes til å trekke tilfeldige variable. Det anbefales å bruke NVIDIA-skjermkort, der NVIDIA er en av de fremste utviklerne på GPU-programmering gjennom sin CUDA-teknologi.

Når det gjelder videre arbeid i feilreserveringsanalysen, er det interessant å teste flere forskjellige modeller mot hverandre. Det er også mulig å se på skadebegrensning på porteføljenivå istedenfor skadebegrensning per polise. Generelt kan vi variere ulike parametere og variabler for å studere hvordan reserver og feilreservering avhenger av disse.

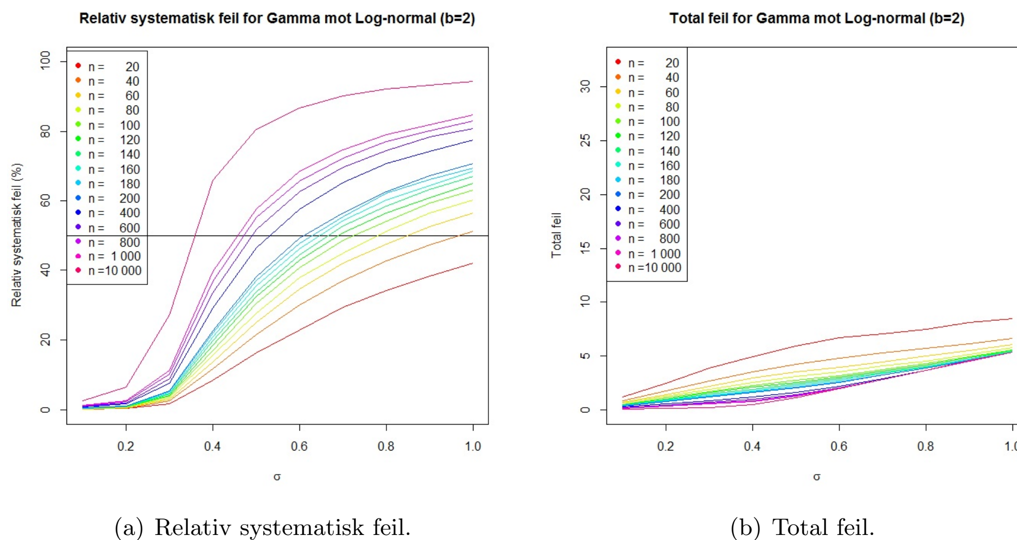
Tillegg A

Appendix

A.1 Figurer for relativ systematisk feil og total feil

Supplerende figurer for skadebegrensningsnivå $b = 2, 3, 4, 6, 7, 8$ og 9.

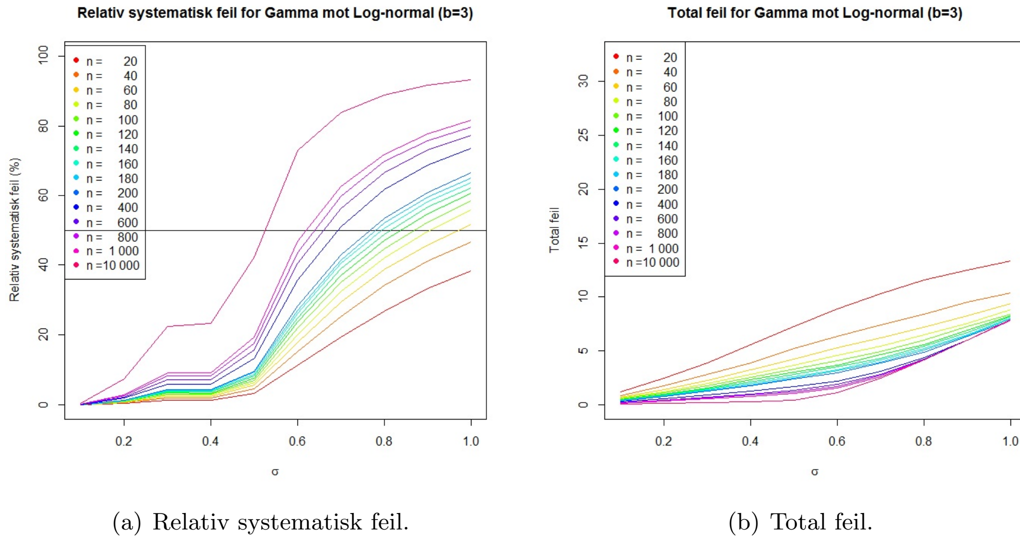
A.1.1 Gamma-modellen mot Log-normal-modellen



(a) Relativ systematisk feil.

(b) Total feil.

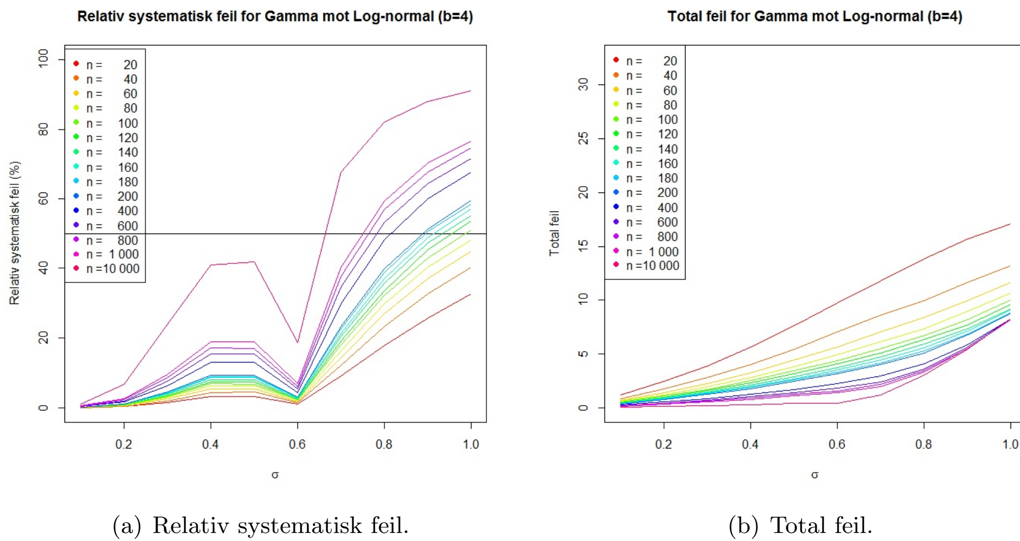
Figur A.1: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skadebegrensning per polise er 2.



(a) Relativ systematisk feil.

(b) Total feil.

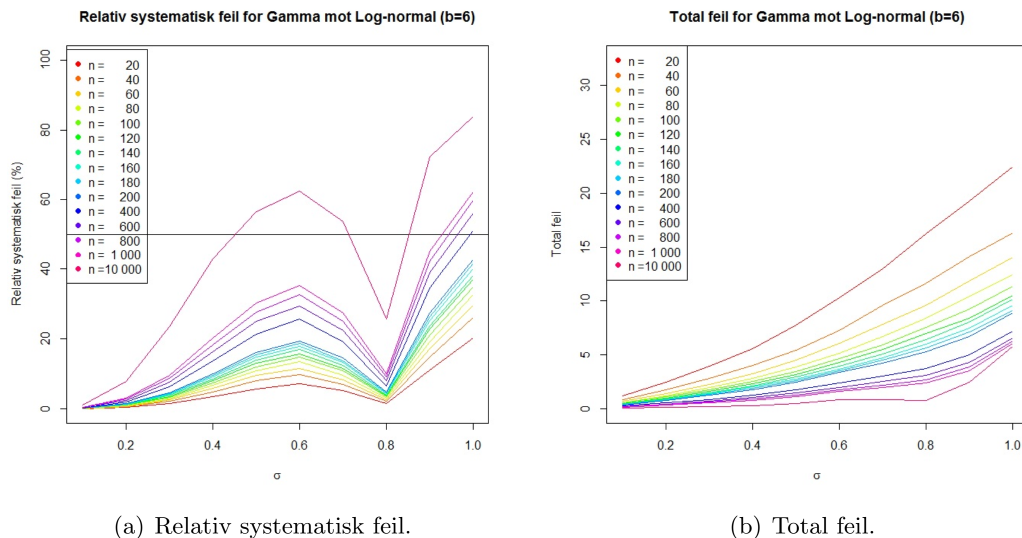
Figur A.2: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skadebegrensning per polise er 3.



(a) Relativ systematisk feil.

(b) Total feil.

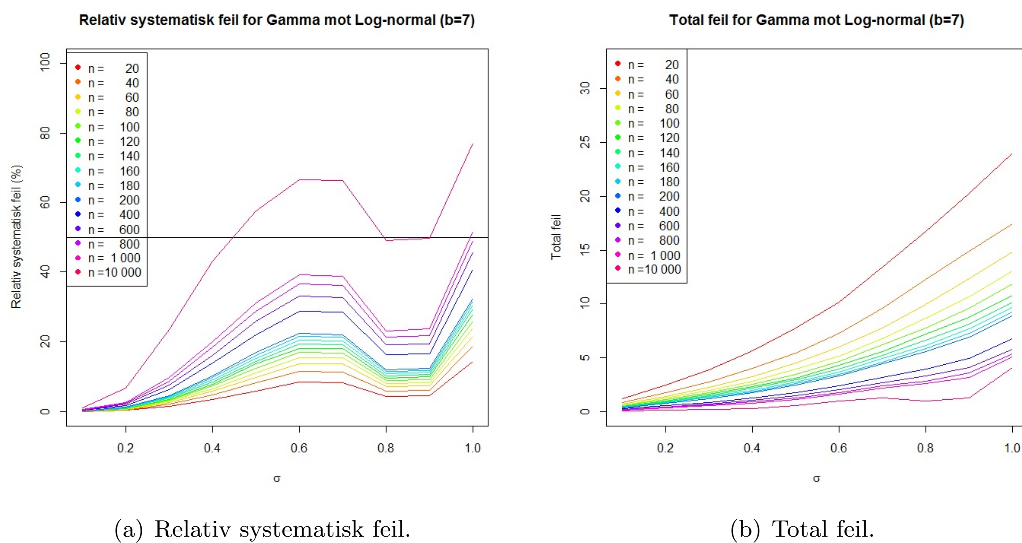
Figur A.3: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skadebegrensning per polise er 4.



(a) Relativ systematisk feil.

(b) Total feil.

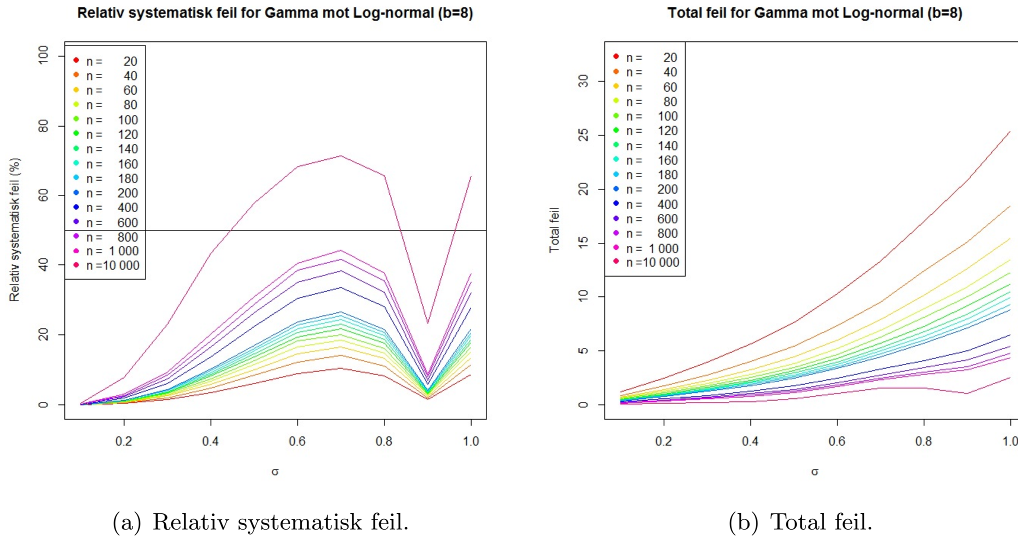
Figur A.4: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skadebegrensning per polise er 6.



(a) Relativ systematisk feil.

(b) Total feil.

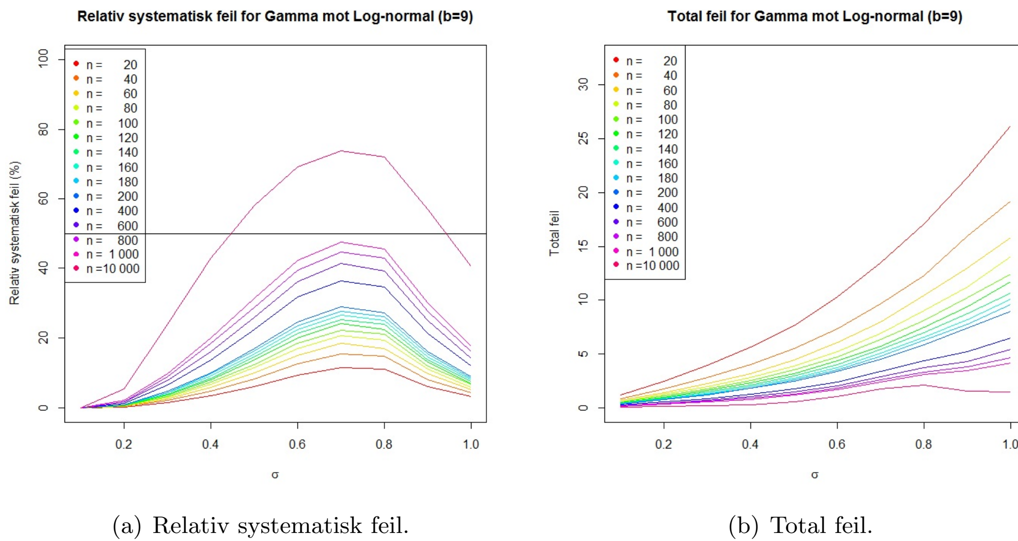
Figur A.5: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skadebegrensning per polise er 7.



(a) Relativ systematisk feil.

(b) Total feil.

Figur A.6: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skadebegrensning per polise er 8.

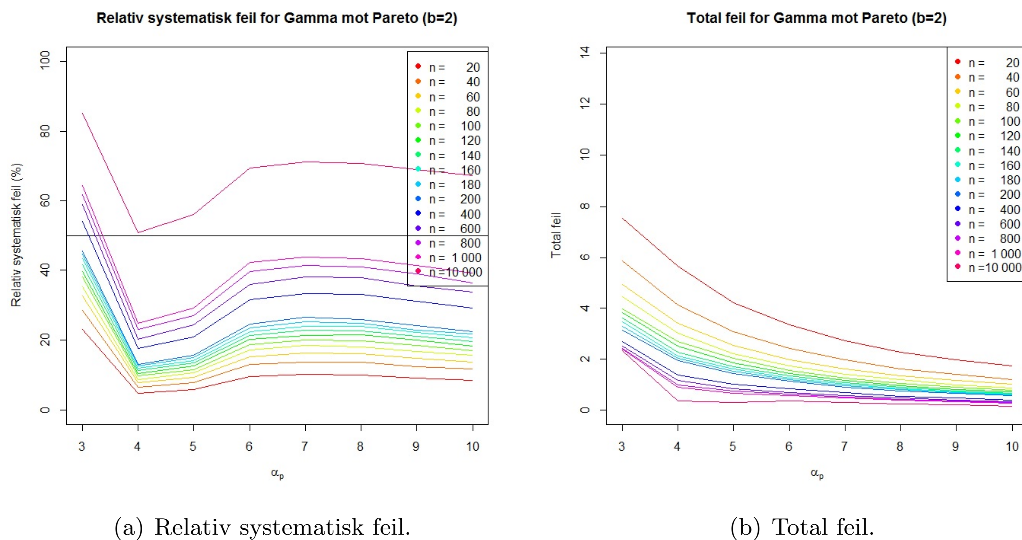


(a) Relativ systematisk feil.

(b) Total feil.

Figur A.7: Relativ systematisk feil (a) og total feil (b) for Gamma mot Log-normal når skadebegrensning per polise er 9.

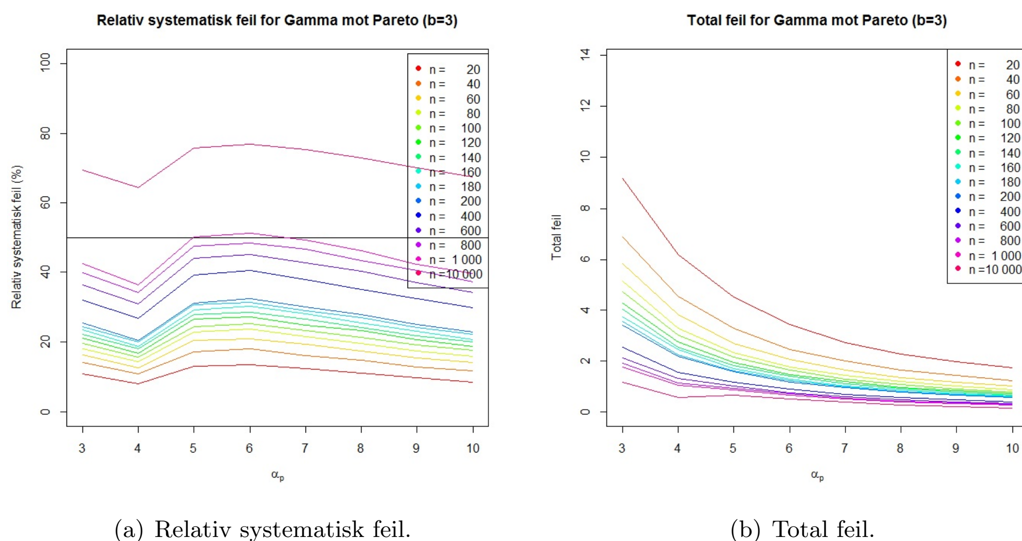
A.1.2 Gamma-modellen mot Pareto-modellen



(a) Relativ systematisk feil.

(b) Total feil.

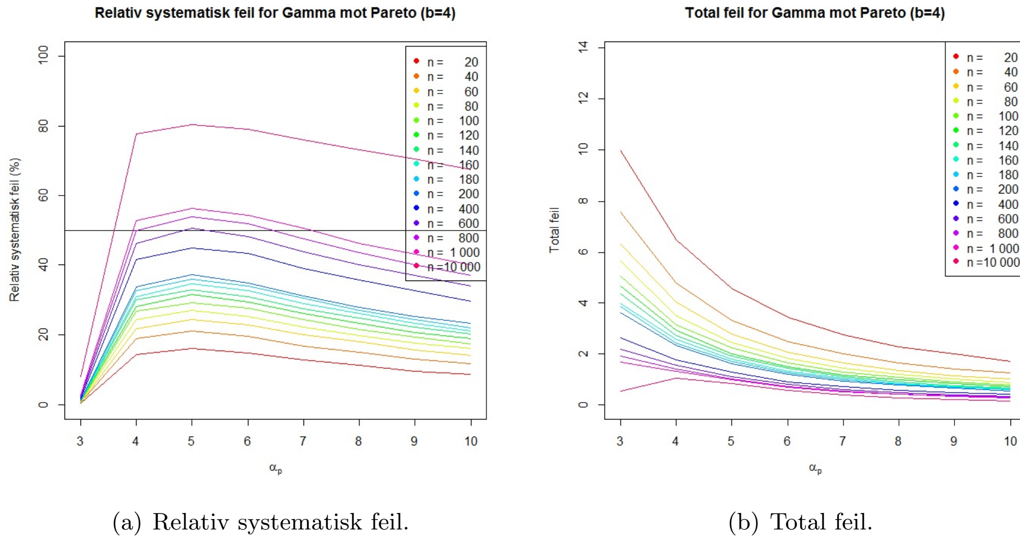
Figur A.8: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 2.



(a) Relativ systematisk feil.

(b) Total feil.

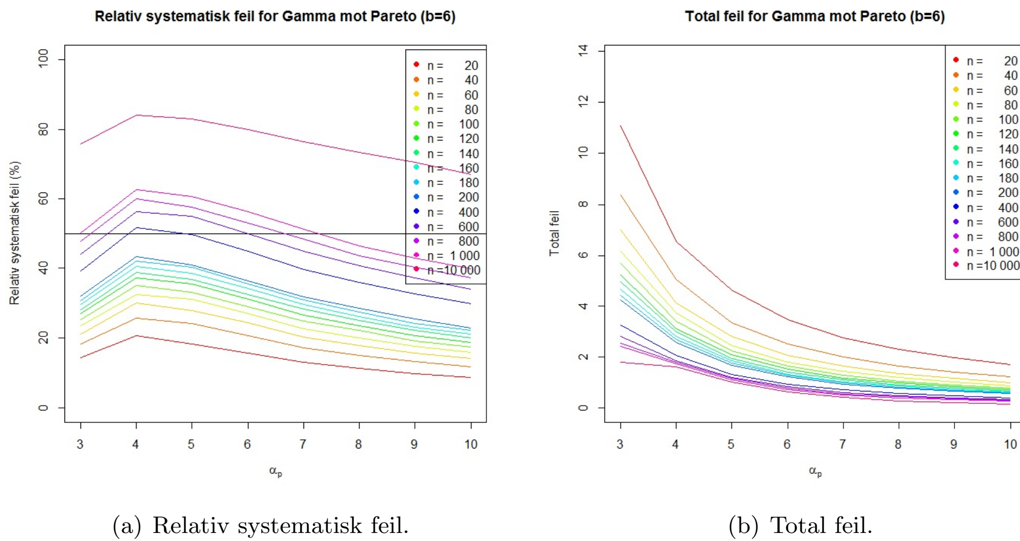
Figur A.9: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 3.



(a) Relativ systematisk feil.

(b) Total feil.

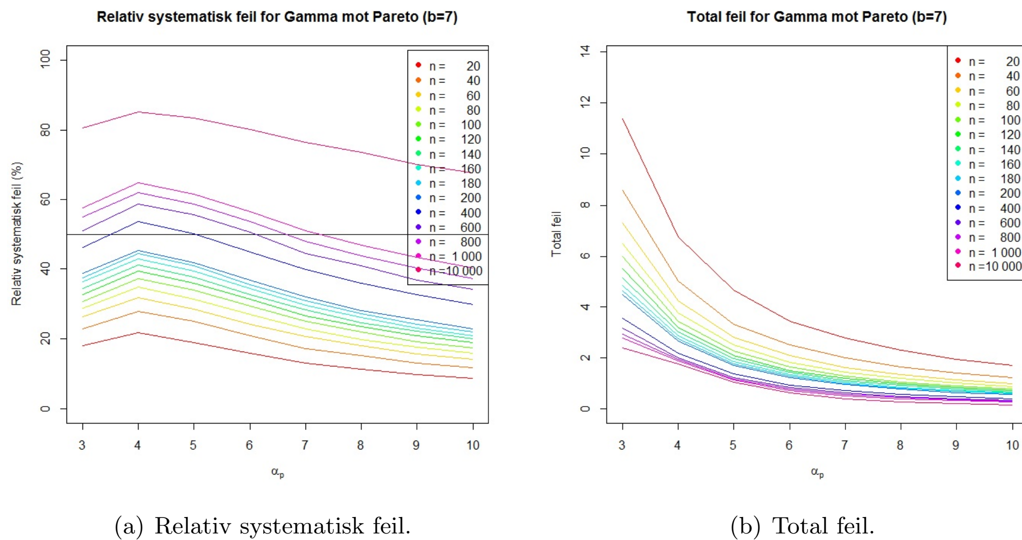
Figur A.10: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 4.



(a) Relativ systematisk feil.

(b) Total feil.

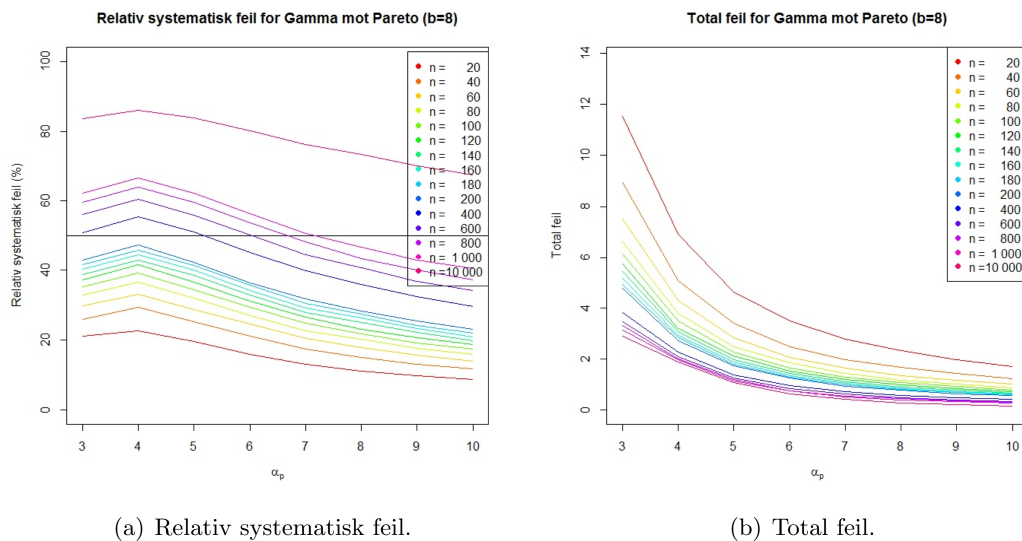
Figur A.11: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 6.



(a) Relativ systematisk feil.

(b) Total feil.

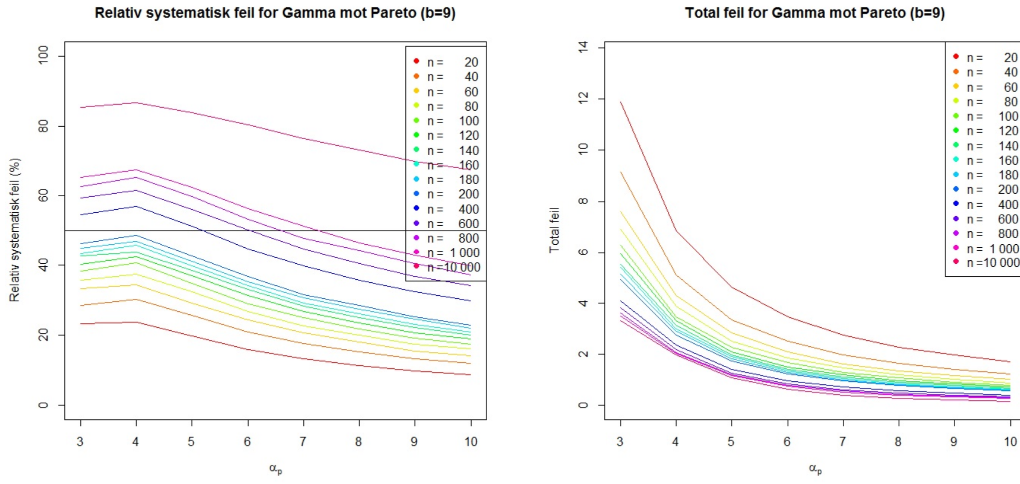
Figur A.12: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 7.



(a) Relativ systematisk feil.

(b) Total feil.

Figur A.13: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 8.



(a) Relativ systematisk feil.

(b) Total feil.

Figur A.14: Relativ systematisk feil (a) og total feil (b) for Gamma mot Pareto når skadebegrensning per polise er 9.

A.2 Tabeller

A.2.1 Optimalt parametersett under gamma-modellen

Korrekt modell: Log-normal (4.2)

	α_0	β_0
$\sigma = 0.1$	100.16992	99.67124
$\sigma = 0.2$	25.16606	24.66745
$\sigma = 0.3$	11.27325	10.77715
$\sigma = 0.4$	6.41245	5.91980
$\sigma = 0.5$	4.15966	3.67078
$\sigma = 0.6$	2.93361	2.45032
$\sigma = 0.7$	2.19252	1.71616
$\sigma = 0.8$	1.71062	1.24232
$\sigma = 0.9$	1.37763	0.91876
$\sigma = 1.0$	1.13782	0.69019

Tabell A.1: Optimale parametere (α_0, β_0) for Gamma-modellen når korrekt modell er Log-normal(μ=0, σ).

Korrekt modell: Pareto (4.3)

	α_0	β_0
$\alpha_p = 3$	0.74308	1.48614
$\alpha_p = 4$	0.80698	2.42153
$\alpha_p = 5$	0.84557	3.38244
$\alpha_p = 6$	0.87106	4.35647
$\alpha_p = 7$	0.88935	5.33537
$\alpha_p = 8$	0.90311	6.32301
$\alpha_p = 9$	0.91375	7.31031
$\alpha_p = 10$	0.92236	8.30247

Tabell A.2: Optimale parametere (α_0, β_0) for Gamma-modellen når korrekt modell er Pareto(α_p).

A.2.2 Optimal 99 %-reserve (ψ_0) under gamma-modellen

Korrekt modell: Log-normal (4.2)

	b=1	b=2	b=3	b=4	b=5	b=6
$\sigma = 0.1$	64.67518	67.59717	67.59935	67.59783	67.59820	67.59648
$\sigma = 0.2$	62.52792	68.91011	68.91087	68.91048	68.91272	68.91087
$\sigma = 0.3$	60.66974	71.05847	71.13639	71.13649	71.13850	71.13646
$\sigma = 0.4$	59.06785	73.59088	74.31918	74.33256	74.33545	74.33439
$\sigma = 0.5$	57.67820	75.99780	78.39908	78.60537	78.61872	78.62172
$\sigma = 0.6$	56.47814	78.06970	83.02423	83.93398	84.08436	84.10413
$\sigma = 0.7$	55.43673	79.81376	87.80085	90.13008	90.76686	90.93332
$\sigma = 0.8$	54.54116	81.29319	92.45512	96.84159	98.51865	99.14176
$\sigma = 0.9$	53.76549	82.57444	96.85536	103.76118	107.05956	108.62514
$\sigma = 1.0$	53.08677	83.67439	100.89580	110.54922	115.95916	118.98978

	b=7	b=8	b=9	b=10	b=∞
$\sigma = 0.1$	67.59748	67.59806	67.59769	67.59635	67.59743
$\sigma = 0.2$	68.91221	68.90954	68.91124	68.90960	68.91023
$\sigma = 0.3$	71.13863	71.13712	71.13692	71.13691	71.13755
$\sigma = 0.4$	74.33567	74.33432	74.33402	74.33841	74.33706
$\sigma = 0.5$	78.62264	78.62243	78.62134	78.62157	78.62217
$\sigma = 0.6$	84.10917	84.10889	84.10715	84.10820	84.10937
$\sigma = 0.7$	90.97716	90.98425	90.98768	90.99056	90.98773
$\sigma = 0.8$	99.37171	99.45684	99.48579	99.48969	99.50024
$\sigma = 0.9$	109.36607	109.70583	109.86898	109.94328	110.00893
$\sigma = 1.0$	120.68904	121.64405	122.17949	122.47717	122.86069

Tabell A.3: Optimal 99 %-reserve (ψ_0) under Gamma-modellen når korrekt modell er Log-normal(μ=0, σ) for ulike skadebegrensningsnivå per polise (*b*).

Korrekt modell: Pareto (4.3)

	b=1	b=2	b=3	b=4	b=5	b=6
$\alpha_p = 3$	29.84570	36.52492	38.26105	38.73617	38.86791	38.90624
$\alpha_p = 4$	22.99999	25.36547	25.65498	25.69073	25.69605	25.69583
$\alpha_p = 5$	18.28093	19.12763	19.17432	19.17720	19.17799	19.17706
$\alpha_p = 6$	14.97322	15.28220	15.28934	15.28946	15.28968	15.28957
$\alpha_p = 7$	12.60323	12.71799	12.71858	12.71927	12.71856	12.71891
$\alpha_p = 8$	10.83748	10.88077	10.88075	10.88122	10.88108	10.88089
$\alpha_p = 9$	9.49454	9.51073	9.51053	9.51064	9.51077	9.51080
$\alpha_p = 10$	8.43881	8.44497	8.44500	8.44511	8.44505	8.44485

	b=7	b=8	b=9	b=10	b=∞
$\alpha_p = 3$	38.91577	38.91786	38.91841	38.91960	38.91826
$\alpha_p = 4$	25.69567	25.69570	25.69467	25.69507	25.69553
$\alpha_p = 5$	19.17689	19.17775	19.17732	19.17736	19.17704
$\alpha_p = 6$	15.28897	15.28974	15.28942	15.29002	15.28977
$\alpha_p = 7$	12.71837	12.71873	12.71888	12.71878	12.71867
$\alpha_p = 8$	10.88087	10.88135	10.88117	10.88111	10.88040
$\alpha_p = 9$	9.51077	9.51053	9.51107	9.51083	9.51116
$\alpha_p = 10$	8.44529	8.44491	8.44520	8.44513	8.44513

Tabell A.4: Optimal 99 %-reserve (ψ_0) under Gamma-modellen når korrekt modell er Pareto(α_p) for ulike skadebegrensningsnivå per polise (*b*).

A.2.3 99 %-reserve (ψ) under korrekt modell

Korrekt modell: Log-normal (4.2)

	b=1	b=2	b=3	b=4	b=5	b=6
$\sigma = 0.1$	64.67399	67.59959	67.59970	67.59892	67.59956	67.59748
$\sigma = 0.2$	62.51184	68.91972	68.92178	68.92025	68.92337	68.92250
$\sigma = 0.3$	60.62437	70.98697	71.19334	71.19675	71.19733	71.19675
$\sigma = 0.4$	58.96678	73.12179	74.39921	74.52038	74.53477	74.53562
$\sigma = 0.5$	57.49734	74.84534	78.14601	78.86126	79.04259	79.09352
$\sigma = 0.6$	56.19373	76.10087	81.88867	83.82849	84.57446	84.89552
$\sigma = 0.7$	55.02938	76.98287	85.31048	88.93064	90.70154	91.64715
$\sigma = 0.8$	53.98386	77.58917	88.28617	93.80137	96.94456	98.87282
$\sigma = 0.9$	53.04198	77.99700	90.83554	98.27599	102.98244	106.14894
$\sigma = 1.0$	52.18846	78.26279	92.99904	102.30287	108.63783	113.18405

	b=7	b=8	b=9	b=10	b=∞
$\sigma = 0.1$	67.59859	67.59847	67.59766	67.59818	67.59819
$\sigma = 0.2$	68.92232	68.92101	68.91909	68.92130	68.92048
$\sigma = 0.3$	71.19922	71.19613	71.19851	71.19652	71.19641
$\sigma = 0.4$	74.53781	74.53818	74.53598	74.53849	74.54134
$\sigma = 0.5$	79.11443	79.11987	79.11860	79.12296	79.12404
$\sigma = 0.6$	85.04506	85.12098	85.16316	85.18480	85.21868
$\sigma = 0.7$	92.19039	92.51856	92.73117	92.86932	93.21661
$\sigma = 0.8$	100.12812	100.98033	101.58161	102.01757	103.68949
$\sigma = 0.9$	108.38155	110.01729	111.25435	112.21047	117.44670
$\sigma = 1.0$	116.58333	119.20911	121.28732	122.97518	135.65452

Tabell A.5: 99 %-reserve (ψ) under korrekt modell Log-normal(μ=0, σ) for ulike skadebegrensningsnivå per polise (b).

Korrekt modell: Pareto (4.3)

	b=1	b=2	b=3	b=4	b=5	b=6
$\alpha_p = 3$	27.55431	34.18807	37.11296	38.78415	39.88806	40.68978
$\alpha_p = 4$	21.83030	25.07337	26.19900	26.75454	27.08523	27.30794
$\alpha_p = 5$	17.76915	19.38920	19.83412	20.02259	20.12250	20.18337
$\alpha_p = 6$	14.81274	15.63515	15.81255	15.87493	15.90343	15.91698
$\alpha_p = 7$	12.60446	13.02660	13.09687	13.11635	13.12245	13.12533
$\alpha_p = 8$	10.91541	11.13398	11.16159	11.16645	11.16812	11.16827
$\alpha_p = 9$	9.59613	9.70942	9.71947	9.72104	9.72122	9.72146
$\alpha_p = 10$	8.54455	8.60351	8.60715	8.60796	8.60763	8.60746

	b=7	b=8	b=9	b=10	b=∞
$\alpha_p = 3$	41.30683	41.80791	42.22368	42.58316	45.69614
$\alpha_p = 4$	27.46975	27.59499	27.69394	27.77198	27.99237
$\alpha_p = 5$	20.22046	20.24546	20.25960	20.26896	20.27467
$\alpha_p = 6$	15.92215	15.92492	15.92523	15.92648	15.92603
$\alpha_p = 7$	13.12573	13.12557	13.12605	13.12556	13.12603
$\alpha_p = 8$	11.16854	11.16821	11.16877	11.16837	11.16812
$\alpha_p = 9$	9.72123	9.72135	9.72158	9.72138	9.72164
$\alpha_p = 10$	8.60770	8.60736	8.60741	8.60743	8.60774

Tabell A.6: 99 %-reserve (ψ) under korrekt modell Pareto(α_p) for ulike skadebegrensningsnivå per polise (b).

A.3 Programkode

A.3.1 Eksempel 1 i seksjon 3.3.1

Listing A.1: Eksempel 1

```
library(rbenchmark)

#Metode 1
x = 0
for (i in 1:1e4){ x[i] = i }

#Metode 2
x = 1
for (i in 2:1e4){ x = c(x,i) }

#Metode 3
x = rep(0,1e4)
for (i in 1:1e4){ x[i] = i }

#Metode 4
x = 1:1e4

#Kjøring av eksempel 1
benchmark("Metode 1" = {x = 0; for (i in 1:1e4){ x[i] = i }},
          "Metode 2" = {x = 1; for (i in 2:1e4){ x = c(x,i) }},
          "Metode 3" = {x = rep(0,1e4); for (i in 1:1e4){ x[i] = i }},
          "Metode 4" = {x = 1:1e4},
          columns=c("test",
            "replications",
            "elapsed",
            "relative"
            ),
          order="relative",
          replications=1000
          )
```

A.3.2 Eksempel 2 i seksjon 3.3.1

Listing A.2: Eksempel 2

```
library(inline)
library(Rcpp)
library(rbenchmark)
library(compiler)

#Standard R, metode 1-3
x1 = function(x){x^5}
x2 = function(x){x**5}
x3 = function(x){x*x*x*x*x}

#Compiler, metode 4-6
x1c = cmpfun(x1)
x2c = cmpfun(x2)
x3c = cmpfun(x3)

#Rcpp, metode 7-8
src4 <- "
double xx = as<double>(x) ;
return wrap(pow(xx,5.0)) ;
"
xRcpp4 <- cxxfunction(signature(x="numeric"), body = src4, plugin = "Rcpp")

src5 <- "
double xx = as<double>(x) ;
return wrap(xx*xx*xx*xx*xx) ;
"
xRcpp5 <- cxxfunction(signature(x="numeric"), body = src5, plugin = "Rcpp")

#Rcpp, metode 9-10
src2 <- "
int mb_ = as<int>(mb) ;
double xx = as<double>(x) ;
Rcpp::NumericVector tmp(mb_) ;
for(int i = 0; i < mb_; i++) {
  tmp(i) = pow(xx,5.0) ;
}
return tmp ;
"
xRcpp2 <- cxxfunction(signature(mb="integer",x="numeric"), body = src2, plugin = "Rcpp")

src3 <- "
int mb_ = as<int>(mb) ;
double xx = as<double>(x) ;
Rcpp::NumericVector tmp(mb_) ;
for(int i = 0; i < mb_; i++) {
  tmp(i) = xx*xx*xx*xx*xx ;
}
return tmp ;
"
xRcpp3 <- cxxfunction(signature(mb="integer",x="numeric"), body = src3, plugin = "Rcpp")


x = 2
m = 1e4 #Antall simuleringer

```

```
56 #Kjøring av eksempel 2
57 benchmark("x^5 R" = {for(i in 1:m) { x1(x) }},
58           "x**5 R" = {for(i in 1:m) { x2(x) }},
59         "x*x*x*x*x R" = {for(i in 1:m) { x3(x) }},
60           "x^5 compiler" = {for(i in 1:m) { x1c(x) }},
61         "x**5 compiler" = {for(i in 1:m) { x2c(x) }},
62         "x*x*x*x*x compiler" = {for(i in 1:m) { x3c(x) }},
63         "x^5 Rcpp" = {for(i in 1:m) { xRcpp4(x) }},
64         "x*x*x*x*x Rcpp" = {for(i in 1:m) { xRcpp5(x) }},
65         "x^5 for Rcpp" = { xRcpp2(m,x) },
66         "x*x*x*x*x for Rcpp" = { xRcpp3(m,x) },
67           columns=c("test",
68             "replications",
69             "elapsed",
70             "relative"
71             ),
72           order="relative",
73           replications=1000
74           )
```

A.3.3 Eksempel 3 i seksjon 3.3.1

Listing A.3: Eksempel 3

```
1  library(inline)
2  library(Rcpp)
3  library(rbenchmark)
4  library(compiler)
5
6  #Standard R, metode 2
7  mean1 = function(zz){sum(zz)/length(zz)}
8
9  #Compiler, metode 3
10 mean1c = cmpfun(mean1)
11
12 #Rcpp, metode 4-5
13 src1 <- "
14 double m ;
15 Rcpp::NumericVector zz(z) ;
16 m = mean(zz) ;
17 return wrap(m) ;
18 "
19 meanRcpp1 <- cxxfunction(signature(z="vector"), body = src1, plugin = "Rcpp")
20
21 src2 <- "
22 Rcpp::NumericVector zz(z) ;
23 return wrap(sum(zz)/zz.size()) ;
24 "
25 meanRcpp2 <- cxxfunction(signature(z="vector"), body = src2, plugin = "Rcpp")
26
27 #Rcpp, metode 6-7
28 src3 <- "
29 int mb_ = as<int>(mb) ;
30 Rcpp::NumericVector zz(z) ;
31 Rcpp::NumericVector tmp(mb_) ;
32 for(int i = 0; i < mb_; i++) {
33   tmp(i) = mean(zz) ;
34 }
35 return wrap(tmp) ;
```

```
36 "
37 meanRcpp3 <- cxxfunction(signature(mb="integer",z="vector"), body = src3, plugin = "Rcpp")
38
39 src4 <- "
40 int mb_ = as<int>(mb) ;
41 int n ;
42 Rcpp::NumericVector zz(z) ;
43 n = zz.size() ;
44 Rcpp::NumericVector tmp(mb_) ;
45 for(int i = 0; i < mb_; i++) {
46   tmp(i) = sum(zz)/n ;
47 }
48 return wrap(tmp) ;
49 "
50 meanRcpp4 <- cxxfunction(signature(mb="integer",z="vector"), body = src4, plugin = "Rcpp")
51
52 z = runif(100)  #Trekker 100 standard uniformt-fordelte variable
53 m = 1e4      #Antall simuleringer
54
55 #Kjøring av eksempel 3
56 benchmark("mean(z) R" = {for(i in 1:m){ mean(z) }},
57           "sum(zz)/length(zz) R" = {for(i in 1:m){ mean1(z) }},
58         "sum(zz)/length(zz) compiler" = {for(i in 1:m){ mean1c(z) }},
59         "mean(z) Rcpp" = {for(i in 1:m){ meanRcpp1(z) }},
60         "sum(zz)/zz.size() Rcpp" = {for(i in 1:m){ meanRcpp2(z) }},
61           "mean(z) for Rcpp" = { meanRcpp3(mb=m,z=z) },
62         "sum(zz)/zz.size() for Rcpp" = { meanRcpp4(mb=m,z=z) },
63           columns=c("test",
64             "replications",
65             "elapsed",
66             "relative"
67             ),
68           order="relative",
69           replications=1000
70           )
```

A.3.4 Test 1 i seksjon 3.4.2

Listing A.4: Test 1

```
1  library(rbenchmark)
2  library(compiler)
3  library(inline)
4  library(Rcpp)
5
6  #Algoritme 1
7  A1 = function(m=1000,lambda=50,A=1,B=1){
8  X = rep(0,m)
9  for (i in 1:m){
10     N = rpois(1,lambda)
11     Z = rgamma(N,A,B)
12     X[i] = sum(Z)
13 }
14 sort(X)[m*0.99]
15 }
16
17 #Algoritme 2
```

```
18 A2 = function(m=1000,lambda=50,A=1,B=1){
19 X = rep(0,m)
20 N = rpois(m,lambda)
21 Z = rgamma(sum(N),A,B)
22 Nk = cumsum(N)
23 Zk = cumsum(Z)
24 x1 = Zk[Nk]
25 x2 = c(0,x1[-m])
26 X = x1-x2
27 sort(X)[m*0.99]
28 }
29
30 #Compiler-versjoner av Algoritme 1 og 2
31 A1c = cmpfun(A1)
32 A2c = cmpfun(A2)
33
34 #Rcpp-versjoner av Algoritme 1 og 2
35 src1 <- "
36     int m_ = as<int>(m) ;
37     int lambda_ = as<int>(lambda) ;
38     double A_ = as<double>(A) ;
39     double B_ = as<double>(B) ;
40     double res99 ;
41     NumericVector X(m_) ;
42     IntegerVector N(1) ;
43     for(int i = 0; i < m_; i++) {
44         N = rpois(1,lambda_) ;
45         NumericVector Z(N[0]) ;
46         Z = rgamma(N[0],A_,1.0/B_) ;
47         X[i] = sum(Z) ;
48         }
49     std::sort(X.begin(), X.end()) ;
50     res99 = X[m_*0.99-1] ;
51     return wrap(res99) ;
52 "
53 A1_Rcpp <- cxxfunction(signature(m="int",lambda="int",A="numeric",B="numeric"),
54         body = src1, plugin = "Rcpp")
55
56 src2 <- "
57     int m_ = as<int>(m) ;
58     int lambda_ = as<int>(lambda) ;
59     double A_ = as<double>(A) ;
60     double B_ = as<double>(B) ;
61     double res99 ;
62     NumericVector N = rpois(m_,lambda_) ;
63     int sumN = sum(N) ;
64     NumericVector Z = rgamma(sumN,A_,1.0/B_) ;
65     NumericVector X(m_);
66     int offset = 0;
67     for(int i = 0; i < m_; ++i) {
68         int endInd = N[i] + offset;
69         for(int j = offset; j < endInd; ++j) {
70                 X[i] += Z[j];
71         }
72         offset += N[i];
73     }
74     std::sort(X.begin(), X.end()) ;
75     res99 = X[m_*0.99-1] ;
76     return wrap(res99) ;
77 "
78 A2_optimal_Rcpp <- cxxfunction(signature(m="int",lambda="int",A="numeric",B="numeric↩
    "),
79         body = src2, plugin = "Rcpp")
```

```
80
81 m = 1e5      #Antall simuleringer
82 lambda = 50      #Poisson−parameter
83
84 #Kjøring av test 1
85 benchmark("Algoritme 1" = { metode1 <− A1(m,lambda) },
86           "Algoritme 2" = { metode2 <− A2(m,lambda) },
87         "Algoritme 2 Rcpp" = { metode3 <− A2_optimal_Rcpp(m,lambda,A=1,B=1) },
88         "Algoritme 1 compiler" = { metode4 <− A1c(m,lambda) },
89         "Algoritme 2 compiler" = { metode5 <− A2c(m,lambda) },
90         "Algoritme 1 Rcpp" = { metode6 <− A1_Rcpp(m,lambda,A=1,B=1) },
91
92           columns=c("test",
93             "replications",
94             "elapsed",
95             "relative"
96             ),
97           order="relative",
98           replications=100
99           )
```

A.3.5 Test 2 i seksjon 3.4.2

Listing A.5: Test 2 - del 1

```
1  #Test 2, del 1 (metode 1−4)
2
3  #Metode 5 kommer i del 2.
4  #Metode 5 må kjøres i eget program siden metode 5
5  #parallelliseres over flere maskiner
6
7  library(rbenchmark)
8  library(foreach)
9  library(doParallel)
10 library(Rcpp)
11
12 #Algoritme 1
13 A1 = function(m=1000,lambda=50,A=1,B=1){
14 X = rep(0,m)
15 for (i in 1:m){
16     N = rpois(1,lambda)
17     Z = rgamma(N,A,B)
18     X[i] = sum(Z)
19 }
20 sort(X)[m*0.99]
21 }
22
23 #Algoritme 2
24 A2 = function(m=1000,lambda=50,A=1,B=1){
25 X = rep(0,m)
26 N = rpois(m,lambda)
27 Z = rgamma(sum(N),A,B)
28 Nk = cumsum(N)
29 Zk = cumsum(Z)
30 x1 = Zk[Nk]
31 x2 = c(0,x1[−m])
32 X = x1−x2
33 sort(X)[m*0.99]
34 }
```

```
35
36 #Henter Rcpp−versjonen av Algoritme 2 (A2_optimal_Rcpp)
37 #fra en egenprodusert R−pakke som heter reserveRcpp
38 library(reserveRcpp)
39
40 #Setter opp klyngen for parallellisering på 1 maskin
41 cl <− makeCluster(32)
42 registerDoParallel(cl)
43
44 #Laster inn pakken reserveRcpp til alle kjerner i klyngen,
45 #slik at vi kan parallellisere A2_optimal_Rcpp−funksjonen
46 clusterEvalQ(cl,library(reserveRcpp))
47
48 m = 1e5      #Antall simuleringer
49 mb = 160         #Antall reserver som beregnes per repetisjon
50 lambda = 50      #Poisson−parameter
51
52 #Kjøring av test 2 for metode 1−4
53 benchmark("Algoritme 1" = { for (i in 1:mb){A1(m,lambda)} },
54           "Algoritme 2" = { for (i in 1:mb){A2(m,lambda)} },
55         "Algoritme 2 Rcpp" = { for (i in 1:mb){A2_optimal_Rcpp(m,lambda)} },
56         "Algoritme 2 Rcpp PAR−32 (1 PC)" = {
57         metode4 <− unlist(foreach(j=1:mb ) %dopar% A2_optimal_Rcpp(m,lambda)) },
58           columns=c("test",
59             "replications",
60             "elapsed",
61             "relative"
62             ),
63           order="relative",
64           replications=10
65           )
```

Listing A.6: Test 2 - del 2

```
1 #Test 2, del 2 (metode 5)
2
3 library(rbenchmark)
4 library(snow)
5 library(Rmpi)
6 library(reserveRcpp)    #egenprodusert R−pakke
7
8 #Antall prosessorkjerner tilgjengelig
9 pk = 160
10
11 #Setter opp klyngen
12 cl <− makeMPIcluster(pk)
13
14 #Laster inn reserveRcpp−pakken i klyngen som inneholder
15 #Rcpp−versjonen av Algoritme 2 (A2_optimal_Rcpp)
16 clusterEvalQ(cl,library(reserveRcpp))
17
18 m = 1e5      #Antall simuleringer
19 mb = 160         #Antall reserver som beregnes per repetisjon
20 lambda = 50      #Poisson−parameter
21 k = 1:mb         #Vektor fra 1 til 160
22
23 #Kjøring av test 2 for metode 5
24 benchmark("Algoritme 2 Rcpp PAR−160 (5 PCer)" = {
25         metode5 <− parSapply(cl, k, A2_optimal_Rcpp, m=1e5, lambda=50, A=1, B=1) },
26           columns=c("test",
27             "replications",
```

```
28                "elapsed",
29                "relative"
30                ),
31             order="relative",
32             replications=10
33             )
34
35 #Avslutter klyngen
36 stopCluster(cl)
37 mpi.exit()
```

A.3.6 Egenprodusert R-pakke: reserveRcpp

Eksempel på R-pakke. R-pakker inneholder 3 mapper *man, src* og *R*. Vi trenger 3 programfiler for å lage en C++-funksjoner i R som vi laster inn ved pakker. Disse er følgende (for Algoritme 2 Rcpp-versjonen):

Listing A.7: Filnavn: *A2_optimal_Cpp.cpp* (legges i *src*-mappen)

```
1  #include "A2_optimal_Cpp.h"
2
3  SEXP A2_optimal_Cpp(SEXP m, SEXP lambda, SEXP A, SEXP B, SEXP dummy){
4      using namespace Rcpp ;
5
6      int m_ = as<int>(m) ;
7      int lambda_ = as<int>(lambda) ;
8      double A_ = as<double>(A) ;
9      double B_ = as<double>(B) ;
10     double res99 ;
11
12     NumericVector N = rpois(m_,lambda_) ;
13     int sumN = sum(N) ;
14     NumericVector Z = rgamma(sumN,A_,1.0/B_) ;
15     NumericVector X(m_);
16     int offset = 0;
17     for(int i = 0; i < m_; ++i) {
18         int endInd = N[i] + offset;
19         for(int j = offset; j < endInd; ++j) {
20                 X[i] += Z[j];
21         }
22         offset += N[i];
23     }
24
25     std::sort(X.begin(), X.end()) ;
26     res99 = X[m_*0.99-1] ;
27     return wrap(res99) ;
28 }
```

Listing A.8: Filnavn: *A2_optimal_Cpp.h* (legges i *src*-mappen)

```
1  #ifndef _reserveRcpp_A2_OPTIMAL_CPP_H
2  #define _reserveRcpp_A2_OPTIMAL_CPP_H
3
4  #include <Rcpp.h>
5
6
```

```
7  RcppExport SEXP A2_optimal_Cpp(SEXP m, SEXP lambda, SEXP A, SEXP B, SEXP dummy) ;
8
9  #endif
```

Listing A.9: Filnavn: *A2_optimal_Rcpp.R* (legges i *R*-mappen)

```
1
2  A2_optimal_Rcpp <- function(m=1e5, lambda=50, A=1, B=1, dummy=1){
3      .Call( "A2_optimal_Cpp", m, lambda, A, B, dummy, PACKAGE = "reserveRcpp" )
4  }
```

A.3.7 Test som viser at Algoritme 1 og 2 gjør samme beregninger

Listing A.10: Test som viser at Algoritme 1 og 2 gjør samme beregninger (inkl. Rcpp-versjon av Algoritme 2)

```
1  library(Rcpp)
2  library(inline)
3
4  #Vi må modifisere Algoritme 1, der vi trekker antall skader utenfor for-løkken og
5  #indekserer disse inne i løkken. Dette gir samme beregner som Algoritme 1, men
6  #trekningen av hvilke tilfeldige tall blir lettere å holde styr på.
7  #Dermed trekker vi samme tilfeldige tall som Algoritme 2 ved kommandoen (set.seed),
8  #slik at vi kan sjekke om de gir samme resultat. Dersom vi bruker set.seed og
9  #tester Algoritme 1 og 2 mot hverandre gir det ingen mening, fordi Algoritme 1
10 #trekker en og en tilfeldig poisson-fordelt inne i løkken, mens Algoritme 2 trekker
11 #alle på en gang. Når vi trekker en og en poisson-fordelt variabel blir det
12 #usynkronisert i trekningen av samme tilfeldige variable mellom Algoritme 1 og 2.
13
14 #Algoritme 1
15 A1 = function(m=1000,lambda=50,A=1,B=1){
16 X = rep(0,m)
17 for (i in 1:m){
18     N = rpois(1,lambda)
19     Z = rgamma(N,A,B)
20     X[i] = sum(Z)
21 }
22 sort(X)[m*0.99]
23 }
24
25 #Algoritme 1 (modifisert, gjør samme beregninger som Algoritme 1)
26 A1m = function (m,lambda,A,B){
27         X = rep(0, m)
28     N = rpois(m,lambda)
29         for (i in 1:m) {
30         Z = rgamma(N[i], A, B)
31             X[i] = sum(Z)
32         }
33         sort(X)[m * 0.99]
34 }
35
36 #Algoritme 2
37 A2 = function (m,lambda,A,B){
38     X = rep(0, m)
39     N = rpois(m, lambda)
40     Z = rgamma(sum(N), A, B)
```

```
41      Nk = cumsum(N)
42      Zk = cumsum(Z)
43      x1 = Zk[Nk]
44      x2 = c(0, x1[-m])
45      X = x1 - x2
46      sort(X)[m * 0.99]
47  }
48
49  #Algoritme 2 Rcpp
50  src2 <- "
51      int m_ = as<int>(m) ;
52      int lambda_ = as<int>(lambda) ;
53      double A_ = as<double>(A) ;
54      double B_ = as<double>(B) ;
55      double res99 ;
56      NumericVector N = rpois(m_,lambda_) ;
57      int sumN = sum(N) ;
58      NumericVector Z = rgamma(sumN,A_,1.0/B_) ;
59      NumericVector X(m_);
60      int offset = 0;
61      for(int i = 0; i < m_; ++i) {
62          int endInd = N[i] + offset;
63          for(int j = offset; j < endInd; ++j) {
64                  X[i] += Z[j];
65          }
66          offset += N[i];
67      }
68      std::sort(X.begin(), X.end()) ;
69      res99 = X[m_*0.99-1] ;
70      return wrap(res99) ;
71  "
72  A2_optimal_Rcpp <- cxxfunction(signature(m="int",lambda="int",A="numeric",B="numeric↩
        "),
73          body = src2, plugin = "Rcpp")
74
75  #Input
76  m=1e5
77  lambda=50
78  A=1
79  B=1
80
81  #Kjører de tre funksjonene A1m, A2, A2_optimal_Rcpp, der set.seed(1)
82  #gjør at vi trekker samme tilfeldige variable.
83  set.seed(1)
84  r1 = A1m(m,lambda,A,B)
85  set.seed(1)
86  r2 = A2(m,lambda,A,B)
87  set.seed(1)
88  r3 = A2_optimal_Rcpp(m,lambda,A,B)
89
90  #Utskrift (alle er like)
91  rbind(r1,r2,r3)
92  #       [,1]
93  #r1 75.51363
94  #r2 75.51363
95  #r3 75.51363
```

A.3.8 Finne vekter og abskisser for (2.30)

Listing A.11: Finne vekter (w) og abskisser (x) - hovedprogram (Fortran)

```
c-----------------------------------------------------------------------
c       Filnavn: genlegendre1.f
c       Putt inn nedre/øvre grense på integral + antall vekter og abskisser
c       i genlegendre.par
c------------------------------------------------------------------------

     implicit none
       real*8 x(100),w(100),a,b
       integer n,i

       open(unit=10,file='genlegendre.par')
       open(unit=20,file='genlegendre.res')

       read(10,*)
       read(10,*)a,b,n
       call gauleg(a,b,x,w,n)
       do i=1,n
          write(20,100)x(i),w(i)
       enddo

 100   format(1x,2f16.8)
       end

c----------------------------------------

       subroutine gauleg(x1,x2,x,w,n)
       implicit none
       real*8 x1,x2,x(100),w(100)
       real*8 eps,xm,xl,z,p1,p2,p3,pp,z1,h
       integer n,m,i,j

c-----------------------------------------------------------------------
c       Copied from p. 125-26 in Numerical recipes. Generates abscissas
c       (x) and weights (w) for integration over (x1,x2) by
c       the Gauss-Legendre algorithm. Tested against tables given
c       in Abramowich and Stegun and found okay. Also tested in use
c       for various integrals up to n=100 and again found okay.
c------------------------------------------------------------------------


       eps=3.d-14
       m=(n+1)/2
       xm=0.5*(x2+x1)
       xl=0.5*(x2-x1)
       do i=1,m
         h=3.141592654*(i-0.25)/(n+0.5)
         z=dcos(h)
 1       continue
         p1=1.0
         p2=0.0
         do j=1,n
           p3=p2
           p2=p1
           p1=((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j
         enddo
         pp=n*(z*p1-p2)/(z*z-1.0)
         z1=z
         z=z1-p1/pp
         if(dabs(z-z1).gt.eps) goto 1
          x(i)=xm-xl*z
          x(n+1-i)=xm+xl*z
          w(i)=2.0*xl/((1.0-z*z)*pp*pp)
```

```
63          w(n+1-i)=w(i)
64        enddo
65        return
66        end
```

Listing A.12: Finne vekter (w) og abskisser (x) - input i hovedprogram

```
1  c------------------------------------------------------------------------
2  c       Filnavn: genlegendre.par
3  c       Input for genlegendre1.f
4  c------------------------------------------------------------------------
5
6     a      b     n
7    -6      6    100
8
9  c------------------------------------------------------------------------
10 c       a er nedre grense på integralet
11 c       b er øvre grense på integralet
12 c       n er antall vekter og abskisser
13 c       vekter og abskisser kommer i filen genlegendre.res
14 c------------------------------------------------------------------------
```

A.3.9 Finne optimalt parametersett under gamma-modellen

Listing A.13: Finne α_0 og β_0 når korrekt modell er Log-normal

```
1  #Ligning (2.30). Finner minste avstand mellom gamma-modellen og log-normal-modellen ↩
      mhp.
2  #alpha og beta med variabelskifte y = exp(x) for å gjøre intervallet for ↩
      definisjonsområdet kortere.
3  gam=function(s,x,sigma){
4  h=exp(s)
5  sum(-w*dnorm(x,0,sigma)*log((h[2]**h[1]/gamma(h[1]))*exp(x)**(h[1]-1)*exp(-h[2]*exp(↩
      x))))
6  }
7
8  #Finner minste intervall på integralgrensene for ulike sigma for å dekke hele
9  #definisjonsområdet for de ulike fordelingene. Sjekker om F(x) = 1.
10 c(pnorm(6,0,1.0), pnorm(5,0,0.9), pnorm(5,0,0.8), pnorm(4,0,0.7), pnorm(4,0,0.6),
      pnorm(3,0,0.5), pnorm(3,0,0.4), pnorm(2,0,0.3), pnorm(2,0,0.2), pnorm(1,0,0.1))
12
13 #Definerer vektorene alpha0 og beta0
14 alpha0 = beta0 = rep(0,10)
15
16 #Må kjøre fortran-programmet på forhånd for å finne vekter og abskisser.
17
18 #sigma = 0.1
19 #M1 består av vekter (w) og abskisser (x), dim(M1): 100x2.
20 #Nedre grense for integralet vi tilnærmer ved (2.30) er første tall i genleg_....txt ↩
      (her: -1).
21 #Øvre grense er andre tall i genleg_....txt (her: 1).
22 #Antall vekter og abskisser er tredje tall i genleg_....txt (her: 100).
23 M1=matrix(scan("genleg_-1_1_100.txt"), byrow=T, ncol=2)
24 x=M1[,1]      #abskisser
25 w=M1[,2]      #vekter
26 opt=optim(c(1,0.01),gam,sigma=0.1,x=x)  #minimerer (2.30) mhp. alpha og beta
27 alpha0[1]=exp(opt$par[1])            #finner alpha0 når sigma=0.1
```

```
28 beta0[1]=exp(opt$par[2])             #finner beta0 når sigma=0.1
29
30 #sigma = 0.2
31 M2=matrix(scan("genleg_-2_2_100.txt"), byrow=T, ncol=2)
32 x=M2[,1]
33 w=M2[,2]
34 opt=optim(c(1,0.01),gam,sigma=0.2,x=x)
35 alpha0[2]=exp(opt$par[1])
36 beta0[2]=exp(opt$par[2])
37
38 #sigma = 0.3, 0.4, 0.5, 0.6
39 M4=matrix(scan("genleg_-4_4_100.txt"), byrow=T, ncol=2)
40 x=M4[,1]
41 w=M4[,2]
42 sigma4 = c(0.3,0.4,0.5,0.6)
43 for (k in 3:6){
44     opt=optim(c(1,0.01),gam,sigma=sigma4[k-2],x=x)
45     alpha0[k]=exp(opt$par[1])
46     beta0[k]=exp(opt$par[2])
47 }
48
49 #sigma = 0.7, 0.8, 0.9, 1.0
50 M6=matrix(scan("genleg_-6_6_100.txt"), byrow=T, ncol=2)
51 x=M6[,1]
52 w=M6[,2]
53 sigma6 = c(0.7,0.8,0.9,1.0)
54 for (k in 7:10){
55     opt=optim(c(1,0.01),gam,sigma=sigma6[k-6],x=x)
56     alpha0[k]=exp(opt$par[1])
57     beta0[k]=exp(opt$par[2])
58 }
59
60 #Skriver ut alpha0 og beta0
61 cbind(alpha0,beta0)
```

Listing A.14: Finne α_0 og β_0 når korrekt modell er Pareto

```
1 #Tetthetsfunksjon for pareto-modellen
2 dpareto = function(x,ap){
3 ap/(1+x)^(ap+1)
4 }
5
6 #Kumulativfunksjon for pareto-modellen
7 ppareto = function(x,ap){
8 1-1/(1+x)^ap
9 }
10
11 #Ligning (2.30). Finner minste avstand mellom gamma- og
12 #pareto-modellen mhp. alpha og beta under gamma-modellen.
13 gam=function(s,x,ap){
14 h=exp(s)
15 sum(-w*(ap*exp(x)/((1+exp(x))^(ap+1)))*log((h[2]**h[1]/gamma(h[1]))*exp(x)**(h[1]-1)↩
       *exp(-h[2]*exp(x))))
16 }
17
18 #Finner minste intervall på integralgrensene for ulike ap (parameter fra Pareto) for↩
       å
19 #dekke hele definisjonsområdet for de ulike fordelingene. Sjekker om F(x) = 1.
20
21 #Sjekker nedre grense
22 1-ppareto(exp(-20),3:10)
```

```
#Sjekker øvre grense
ppareto(exp(5.8),3:10)

#Definerer vektorene alpha0 og beta0
alpha0 = beta0 = rep(0,8)

#Må kjøre fortran-programmet på forhånd for å finne vekter og abskisser.

#ap=3
#M1 består av vekter (w) og abskisser (x), dim(M1): 100x2.
#Nedre grense for integralet vi tilnærmer ved (2.30) er første tall i genleg_....txt ↩
    (her: -20).
#Øvre grense er andre tall i genleg_....txt (her: 5.8).
#Antall vekter og abskisser er tredje tall i genleg_....txt (her: 100).
M1=matrix(scan("genleg_-20_5-8_100.txt"), byrow=T, ncol=2)
x=M1[,1]     #abskisser
w=M1[,2]     #vekter
opt=optim(c(1,0.01),gam,ap=3,x=x)    #minimerer (2.30) mhp. alpha og beta
alpha0[1]=exp(opt$par[1])        #finner alpha0 når ap=3
beta0[1]=exp(opt$par[2])         #finner beta0 når ap=3

#ap = 4,5,6
M2=matrix(scan("genleg_-20_5_100.txt"), byrow=T, ncol=2)
x=M2[,1]
w=M2[,2]
ap2 = c(2,3,4)
for (k in 4:6){
    opt=optim(c(1,0.01),gam,ap=k,x=x)
    alpha0[k-2]=exp(opt$par[1])
    beta0[k-2]=exp(opt$par[2])
}

#ap = 7,8,9,10
M3=matrix(scan("genleg_-20_4_100.txt"), byrow=T, ncol=2)
x=M3[,1]
w=M3[,2]
ap3 = c(7,8,9,10)
for (k in 7:10){
    opt=optim(c(1,0.01),gam,ap=k,x=x)
    alpha0[k-2]=exp(opt$par[1])
    beta0[k-2]=exp(opt$par[2])
}

#Skriver ut alpha0 og beta0
cbind(alpha0,beta0)
```

A.3.10 Beregne systematisk feil

Listing A.15: Beregne systematisk feil for gamma- mot log-normal-modellen

```
library(foreach)
library(doParallel)

#Parameter sigma under korrekt modell (log-normal)
sigma1 = seq(0.1,1,0.1)

#Optimal alpha0 og beta0 under gamma-modellen
alpha0 = c(100.169920, 25.166065, 11.273246, 6.412448, 4.159660,
        2.933612, 2.192515, 1.710618, 1.377630, 1.137820)
```

```
10 beta0 = c(99.6712388, 24.6674451, 10.7771491, 5.9197978, 3.6707835,
11       2.4503209, 1.7161565, 1.2423191, 0.9187613, 0.6901873)
12
13 #Funksjon (Algoritme 2) som beregner psi0 og psi
14 A2par = function(m=1e5,lambda=50,b){
15 eps = 0.01          #Bestemmer nivået på reserven (her: 99 %-reserve)
16 psi0 = numeric(0)       #Definerer optimal reserve under gamma-modellen
17 psi = numeric(0)        #Definerer korrekt reserve
18 for (k in 1:length(sigma1)){
19     N = rpois(m,lambda)
20     Zgam0 = pmin(rgamma(sum(N),alpha0[k],beta0[k]),b)
21     Zlnorm = pmin(rlnorm(sum(N),0,sigma1[k]),b)
22     Nk = cumsum(N)
23     Zkgam0 = cumsum(Zgam0)
24     Zklnorm = cumsum(Zlnorm)
25     x1G = Zkgam0[Nk]
26     x2G = c(0,x1G[-length(x1G)])
27     x1LN = Zklnorm[Nk]
28     x2LN = c(0,x1LN[-length(x1LN)])
29     psi0[k] = sort(x1G-x2G)[m*(1-eps)]
30     psi[k] = sort(x1LN-x2LN)[m*(1-eps)]
31 }
32 c(psi0,psi)
33 }
34
35 #Setter opp klynge på 1 maskin med 32 prosessorkjerner
36 cl <- makeCluster(32)
37 registerDoParallel(cl)
38
39 #Beregner psi0 og psi mb=10000 ganger, og tar gjennomsnitt av psi0- og
40 #psi-vektor for å gjøre MC-feilen neglisjerbar for psi0 og psi
41 mb = 10000
42 xx <- foreach(j=1:mb ) %dopar% A2par(b=1)   #Parallelliserer funksjon A2par. BYTT UT↩
       for passende skadebegrensning (b). Her: b=1.
43 res = matrix(unlist(xx),byrow=T,mb,20)  #Henter ut resultater
44 psi0 = apply(res[,1:10],2,mean)     #Henter ut psi0
45 psi = apply(res[,11:20],2,mean)     #Henter ut psi
46 sysfeil = abs(psi0-psi)             #Beregner systematisk feil
```

Listing A.16: Beregne systematisk feil for gamma- mot pareto-modellen

```
1 library(foreach)
2 library(doParallel)
3
4 #Funksjon for å trekke m antall pareto-fordelte med parameter ap
5 rPareto = function(m,ap){
6 exp(rexp(m,ap))-1
7 }
8
9 #Parameter ap under korrekt modell (pareto)
10 ap = 3:10
11
12 #Optimal alpha0 og beta0 under gamma-modellen
13 alpha0 = c(0.7430800, 0.8069785, 0.8455711, 0.8710604, 0.8893478,
14          0.9031141, 0.9137544, 0.9223556)
15 beta0 = c(1.486143, 2.421529, 3.382436, 4.356468, 5.335372,
16          6.323013, 7.310311, 8.302472)
17
18 #Funksjon (Algoritme 2) som beregner psi0 og psi
19 A2par = function(m=1e5,lambda=50,b){
20 eps = 0.01          #Bestemmer nivået på reserven (her: 99 %-reserve)
```

```
21 psi0 = numeric(0)        #Definerer optimal reserve under gamma-modellen
22 psi = numeric(0)         #Definerer korrekt reserve
23 for (k in 1:length(ap)){
24     N = rpois(m,lambda)
25     Zgam0 = pmin(rgamma(sum(N),alpha0[k],beta0[k]),b)
26     Zpareto = pmin(rPareto(sum(N),ap[k]),b)
27     Nk = cumsum(N)
28     Zkgam0 = cumsum(Zgam0)
29     Zkpareto = cumsum(Zpareto)
30     x1G = Zkgam0[Nk]
31     x2G = c(0,x1G[-length(x1G)])
32     x1P = Zkpareto[Nk]
33     x2P = c(0,x1P[-length(x1P)])
34     psi0[k] = sort(x1G-x2G)[m*(1-eps)]
35     psi[k] = sort(x1P-x2P)[m*(1-eps)]
36 }
37 c(psi0,psi)
38 }
39
40 #Setter opp klynge på 1 maskin med 32 prosessorkjerner
41 cl <- makeCluster(32)
42 registerDoParallel(cl)
43
44 #Beregner psi0 og psi mb=10000 ganger, og tar gjennomsnitt av psi0- og
45 #psi-vektor for å gjøre MC-feilen neglisjerbar for psi0 og psi
46 mb = 10000
47 xx <- foreach(j=1:mb ) %dopar% A2par(b=1)   #Parallelliserer funksjon A2par. BYTT UT↩
       for passende skadebegrensning (b). Her: b=1
48 res = matrix(unlist(xx),byrow=T,mb,16)  #Henter ut resultater
49 psi0 = apply(res[,1:8],2,mean)      #Henter ut psi0
50 psi = apply(res[,9:16],2,mean)      #Henter ut psi
51 sysfeil = abs(psi0-psi)             #Beregner systematisk feil
```

A.3.11 Beregne relativ systematisk feil/total feil/parameterfeil

Listing A.17: Beregne relativ systematisk feil/total feil/parameterfeil for gamma- mot log-normal-modellen

```
1  #Kjøretid: 10-12 timer
2  #HER: Skadebegrensning per polise b=1
3  library(compiler)
4  library(foreach)
5  library(doParallel)
6
7  #Minus-likelihooden til gamma
8  llmin = function(alpha,x){
9  n = length(x) ;
10 -alpha*(log(alpha)-1) + lgamma(alpha) + alpha*(log(sum(x)/n)-sum(log(x))/n) }
11 llminc = cmpfun(llmin)
12
13 #Tekst
14 sig_text = c("sigma=0.1","sigma=0.2","sigma=0.3","sigma=0.4","sigma=0.5",
15         "sigma=0.6","sigma=0.7","sigma=0.8","sigma=0.9","sigma=1.0")
16 n_text = c("n=20","n=40","n=60","n=80","n=100","n=120","n=140","n=160","n=180",
17         "n=200","n=400","n=600","n=800","n=1000","n=10000")
18
19 n1 = c(seq(20,200,20),400,600,800,1000,10000)   #Antall observasjoner
20 sigma1 = seq(0.1,1,0.1)             #Parameter sigma under log-normal-modellen
```

```
21 nsim = length(n1)*length(sigma1)        #Antall ulike scenarioer
22 nn = c(mapply(rep,n1,length(sigma1)))   #Vektor med antall observasjoner for hvert ↩
       scenario
23 sig = rep(sigma1,length(n1))            #Vektor med parameter sigma for hvert ↩
       scenario
24
25 #BYTT UT psi0 og psi for andre skadebegrensningsnivåer (her: b=1)
26 psi0 = c(64.67518, 62.52792, 60.66974, 59.06785, 57.67820, 56.47814,
27          55.43673, 54.54116, 53.76549, 53.08677)
28 psi = c(64.67399, 62.51184, 60.62437, 58.96678, 57.49734, 56.19373,
29          55.02938, 53.98386, 53.04198, 52.18846)
30 sysfeil = abs(psi0-psi)
31
32 #Funksjon som returnerer estimerte 99 %-reserver (psi_hat), alpha_hat og beta_hat
33 A2fast = function(m,lambda,n,sigma,b){
34 z = rlnorm(n,0,sigma)           #Trekker underliggende data fra korrekt modell (log-↩
       normal)
35 A = optimize(llminc,c(0.001,1000),x=z)$minimum  #Estimerer alpha_hat under gamma-↩
       modellen ved ML
36 B = n*A/sum(z)                        #Estimerer beta_hat under gamma-modellen ved ML
37
38 #Algoritme 2
39 N = rpois(m,lambda)
40 Z = pmin(rgamma(sum(N),A,B),b)
41 Nk = cumsum(N)
42 Zk = cumsum(Z)
43 x1 = Zk[Nk]
44 x2 = c(0,Zk[Nk[-length(Nk)]])
45 res = sort(x1-x2)[m*0.99]
46 c(res,A,B)  #Returnerer estimert reserve (psi_hat), alpha_hat og beta_hat
47 }
48
49 #Setter opp klynge på 1 maskin med 32 prosessorkjerner
50 cl <- makeCluster(32)
51 registerDoParallel(cl)
52
53 #Evaluerer funksjonen A2fast for alle scenarioer (antall: nsim)
54 A2fastALL = function(){
55 result = matrix(0,nsim,3)
56 for (k in 1:nsim){
57     result[k,] = A2fast(m=1e5,lambda=50,n=nn[k],sigma=sig[k],b=1)   #BYTT UT for ↩
           passende skadebegrensningsnivå (her: b=1)
58 }
59 result
60 }
61
62 mb=1e4          #Antall beregnede estimerte reserver per scenario
63 x <- foreach(j=1:mb ) %dopar% A2fastALL()       #Parallelliserer funksjonen ↩
       A2fastALL
64 output = unlist(x)  #Henter resultater fra parallelliseringen
65
66 #Henter ut resultater: estimert reserve (psi_hat), alpha_hat/beta_hat under gamma-↩
       modellen
67 outR=outA=outB=matrix(0,nsim,mb)
68 for (i in 1:mb){
69     outR[,i] = output[1:nsim+(nsim*3)*(i-1)]
70     outA[,i] = output[(nsim+1):(nsim*2)+(nsim*3)*(i-1)]
71     outB[,i] = output[(nsim*2+1):(nsim*3)+(nsim*3)*(i-1)]
72 }
73
74 psihat = outR
75 psihat = matrix(unlist(x),byrow=F,nsim) #Legger estimerte reserver i en matrise (↩
       dimensjon: nsim x mb)
```

```
76 total = matrix(apply(abs(psihat-psi),1,mean),length(sigma1),length(n1),dimnames = ↩
      list(sig_text,n_text))        #Total feil
77 tilfeldig = matrix(apply(abs(psihat-psi0),1,mean),length(sigma1),length(n1),dimnames↩
      = list(sig_text,n_text))   #Tilfeldig feil
78 RSF = sysfeil/(sysfeil+tilfeldig)   #Relativ systematisk feil
79
80
81 #Parameterfeil
82
83 #Optimalt parametersett under gamma-modellen
84 alpha0 = c(100.169920, 25.166065, 11.273246, 6.412448, 4.159660,
85         2.933612, 2.192515, 1.710618, 1.377630, 1.137820)
86 beta0 = c(99.6712388, 24.6674451, 10.7771491, 5.9197978, 3.6707835,
87         2.4503209, 1.7161565, 1.2423191, 0.9187613, 0.6901873)
88
89 mA = matrix(apply(outA,1,mean),byrow=F,length(sigma1),dimnames = list(sig_text,n_↩
      text)) #Gjennomsnitt av mb=10000 estimerte alpha_hat
90 sA = matrix(apply(outA,1,sd),byrow=F,length(sigma1),dimnames = list(sig_text,n_text)↩
      )   #Standardavvik av mb=10000 estimerte alpha_hat
91 mB = matrix(apply(outB,1,mean),byrow=F,length(sigma1),dimnames = list(sig_text,n_↩
      text)) #Gjennomsnitt av mb=10000 estimerte beta_hat
92 sB = matrix(apply(outB,1,sd),byrow=F,length(sigma1),dimnames = list(sig_text,n_text)↩
      )   #Standardavvik av mb=10000 estimerte beta_hat
93 REFA = (mA-alpha0)/alpha0       #Relativ estimatingsfeil av alpha
94 RSDA = sA/mA                #Relativt standardavvik for alpha_hat
95 REFB = (mB-beta0)/beta0         #Relativ estimatingsfeil av beta
96 RSDB = sB/mB                #Relativt standardavvik for beta_hat
```

Listing A.18: Beregne relativ systematisk feil/total feil/parameterfeil for gamma- mot pareto-modellen

```
1 #Kjøretid: ca. 10 timer
2 #HER: Skadebegrensning per polise b=1
3 library(compiler)
4 library(foreach)
5 library(doParallel)
6
7 #Minus-likelihooden til gamma
8 llmin = function(alpha,x){
9 n = length(x) ;
10 -alpha*(log(alpha)-1) + lgamma(alpha) + alpha*(log(sum(x)/n)-sum(log(x))/n) }
11 llminc = cmpfun(llmin)
12
13 #Funksjon for å trekke n antall pareto-fordelte variable med parameter ap
14 rPareto = function(n,ap){
15 exp(rexp(n,ap))-1
16 }
17
18 #Tekst
19 ap_text = c("ap=3","ap=4","ap=5","ap=6","ap=7","ap=8","ap=9","ap=10")
20 n_text = c("n=20","n=40","n=60","n=80","n=100","n=120","n=140","n=160","n=180",
21         "n=200","n=400","n=600","n=800","n=1000","n=10000")
22
23 n1 = c(seq(20,200,20),400,600,800,1000,10000)   #Antall observasjoner
24 ap1 = 3:10                          #Parameter ap under pareto-modellen
25 nsim = length(n1)*length(ap1)           #Antall ulike scenarioer
26 nn = c(mapply(rep,n1,length(ap1)))      #Vektor med antall observasjoner for hvert ↩
      scenario
27 ap = rep(ap1,length(n1))             #Vektor med parameter ap for hvert scenario
28
29 #BYTT UT psi0 og psi for andre skadebegrensningsnivåer (her: b=1)
```

```
psi0 = c(29.845697, 22.999993, 18.280933, 14.973217, 12.603228, 10.837479,
         9.494535, 8.438808)
psi = c(27.554307, 21.830298, 17.769152, 14.812741, 12.604459, 10.915408,
     9.596126, 8.544551)
sysfeil = abs(psi0-psi)

#Funksjon som returnerer estimerte 99 %-reserver (psi_hat), alpha_hat og beta_hat
A2fast = function(m,lambda,n,ap,b){
z = rPareto(n,ap)        #Trekker underliggende data fra korrekt modell (pareto)
A = optimize(llminc,c(0.001,1000),x=z)$minimum  #Estimerer alpha_hat under gamma-modellen ved ML
B = n*A/sum(z)       #Estimerer beta_hat under gamma-modellen ved ML

#Algoritme 2
N = rpois(m,lambda)
Z = pmin(rgamma(sum(N),A,B),b)
Nk = cumsum(N)
Zk = cumsum(Z)
x1 = Zk[Nk]
x2 = c(0,Zk[Nk[-length(Nk)]])
res = sort(x1-x2)[m*0.99]
c(res,A,B)  #Returnerer estimert reserve (psi_hat), alpha_hat og beta_hat
}

#Setter opp klynge på 1 maskin med 32 prosessorkjerner
cl <- makeCluster(32)
registerDoParallel(cl)

#Evaluerer funksjonen A2fast for alle scenarioer (antall: nsim)
A2fastALL = function(){
result = matrix(0,nsim,3)
for (k in 1:nsim){
    result[k,] = A2fast(m=1e5,lambda=50,n=nn[k],ap=ap[k],b=1)   #BYTT UT for passende skadebegrensningsnivå (her: b=1)
}
result
}

mb=1e4          #Antall beregnede estimerte reserver per scenario
x <- foreach(j=1:mb ) %dopar% A2fastALL()       #Parallelliserer funksjonen A2fastALL
output = unlist(x)  #Henter resultater fra parallelliseringen

#Henter ut resultater: estimert reserve (psi_hat), alpha_hat/beta_hat under gamma-modellen
outR=outA=outB=matrix(0,nsim,mb)
for (i in 1:mb){
    outR[,i] = output[1:nsim+(nsim*3)*(i-1)]
    outA[,i] = output[(nsim+1):(nsim*2)+(nsim*3)*(i-1)]
    outB[,i] = output[(nsim*2+1):(nsim*3)+(nsim*3)*(i-1)]
}

psihat = outR
psihat = matrix(unlist(x),byrow=F,nsim) #Legger estimerte reserver i en matrise (dimensjon: nsim x mb)
total = matrix(apply(abs(psihat-psi),1,mean),length(ap1),length(n1),dimnames = list(ap_text,n_text))        #Total feil
tilfeldig = matrix(apply(abs(psihat-psi0),1,mean),length(ap1),length(n1),dimnames = list(ap_text,n_text))   #Tilfeldig feil
RSF = sysfeil/(sysfeil+tilfeldig)   #Relativ systematisk feil


#Parameterfeil
```

```
86
87 #Optimalt parametersett under gamma−modellen
88 alpha0 = c(0.7430800, 0.8069785, 0.8455711, 0.8710604, 0.8893478,
89          0.9031141, 0.9137544, 0.9223556)
90 beta0 = c(1.486143, 2.421529, 3.382436, 4.356468, 5.335372,
91          6.323013, 7.310311, 8.302472)
92
93 mA = matrix(apply(outA,1,mean),byrow=F,length(ap1),dimnames = list(ap_text,n_text)) ↩
       #Gjennomsnitt av mb=10000 estimerte alpha_hat
94 sA = matrix(apply(outA,1,sd),byrow=F,length(ap1),dimnames = list(ap_text,n_text))   ↩
       #Standardavvik av mb=10000 estimerte alpha_hat
95 mB = matrix(apply(outB,1,mean),byrow=F,length(ap1),dimnames = list(ap_text,n_text)) ↩
       #Gjennomsnitt av mb=10000 estimerte beta_hat
96 sB = matrix(apply(outB,1,sd),byrow=F,length(ap1),dimnames = list(ap_text,n_text))   ↩
       #Standardavvik av mb=10000 estimerte beta_hat
97 REFA = (mA−alpha0)/alpha0      #Relativ estimatingsfeil av alpha
98 RSDA = sA/mA                  #Relativt standardavvik for alpha_hat
99 REFB = (mB−beta0)/beta0        #Relativ estimatingsfeil av beta
100 RSDB = sB/mB                  #Relativt standardavvik for beta_hat
```

A.3.12 Sannsynlighet for å velge riktig modell

Listing A.19: Sannsynlighet for å velge korrekt modell når korrekt modell er log-normal-modellen

```
1 library(compiler)
2 library(foreach)
3 library(doParallel)
4
5 #Minus−likelihooden til gamma
6 llmin = function(alpha,x){
7 n = length(x) ;
8 −alpha∗(log(alpha)−1) + lgamma(alpha) + alpha∗(log(sum(x)/n)−sum(log(x))/n) }
9 llminc = cmpfun(llmin)
10
11 #Kriterie for hvilken modell vi velger. Lavest Q−score = best modell.
12 Q = function(q,z){sum(abs(q−sort(z)))}
13 Qc = cmpfun(Q)
14
15 #Tekst
16 sig_text = c("sigma=0.1","sigma=0.2","sigma=0.3","sigma=0.4","sigma=0.5",
17          "sigma=0.6","sigma=0.7","sigma=0.8","sigma=0.9","sigma=1.0")
18 n_text = c("n=20","n=40","n=60","n=80","n=100","n=120","n=140","n=160","n=180",
19          "n=200","n=400","n=600","n=800","n=1000","n=10000")
20
21 n1 = c(seq(20,200,20),400,600,800,1000,10000)   #Antall observasjoner
22 sigma1 = seq(0.1,1,0.1)                  #Parameter sigma under log−normal−modellen
23 nsim = length(n1)∗length(sigma1)         #Antall ulike scenarioer
24 nn = c(mapply(rep,n1,length(sigma1)))    #Vektor med antall observasjoner for hvert ↩
       scenario
25 sig = rep(sigma1,length(n1))             #Vektor med parameter sigma for hvert ↩
       scenario
26
27 #Funksjon som returnerer sannsynlighet for at vi velger riktig modell (log−normal)
28 quick = function(mb,n,sigma){
29 i = 1:n
30 p = (i−0.5)/n
31 check = rep(0,mb)
32 M = rep(0,2)
```

```
33 for (k in 1:mb){
34 z = rlnorm(n,0,sigma)   #Trekker underliggende data fra korrekt modell (log-normal)
35 aG = optimize(llminc,c(0.001,1000),x=z)$minimum     #Estimerer alpha_hat under gamma↩
       -modellen ved ML
36 bG = n*aG/sum(z)            #Estimerer beta_hat under gamma-modellen ved ML
37 qG = qgamma(p,aG,bG)        #Finner persentiler for tilpasset gamma-modell
38 logz = log(z)
39 qLN = qlnorm(p,sum(logz)/n,sd(logz))    #Finner persentiler for tilpasset log-normal↩
       -modell med ML-estimat for mu og sigma
40 M = c(Qc(qG,z),Qc(qLN,z))            #Beregner Q-score
41 check[k] = which.min(M)              #Velger modell med lavest Q-score
42 }
43 sum(check==2)/mb    #Sannsynlighet for å velge riktig modell (log-normal)
44 }
45
46 #Parallelliserer funksjonen quick og beregner hvor stor sannsynlighet det er for å ↩
       velge riktig
47 #modell for alle scenarioer (antall: nsim)
48 x <- foreach (j=1:nsim) %dopar% quick(mb=1e4,n=nn[j],sigma=sig[j])
49 resultLN = matrix(unlist(x),length(sigma1),length(n1),dimnames=list(c(sig_text),c(n_↩
       text)))
```

Listing A.20: Sannsynlighet for å velge korrekt modell når korrekt modell er pareto-modellen

```
1 library(compiler)
2 library(foreach)
3 library(doParallel)
4
5 #Minus-likelihooden til gamma
6 llmin = function(alpha,x){
7 n = length(x) ;
8 -alpha*(log(alpha)-1) + lgamma(alpha) + alpha*(log(sum(x)/n)-sum(log(x))/n) }
9 llminc = cmpfun(llmin)
10
11 rPareto = function(n,ap=3){exp(rexp(n,ap))-1}   #Funksjon for å trekke n pareto-↩
       fordelte variable med parameter ap
12 qPareto = function(q,ap){ X = (1-q)^(1/ap) ; (1-X)/X }  #Kumulativfunksjonen for ↩
       pareto med parameter ap
13
14 #Kriterie for hvilken modell vi velger. Lavest Q-score = best modell.
15 Q = function(q,z){ sum(abs(q-sort(z))) }
16 Qc = cmpfun(Q)
17
18 #Tekst
19 ap_text = c("ap=3","ap=4","ap=5","ap=6","ap=7","ap=8","ap=9","ap=10")
20 n_text = c("n=20","n=40","n=60","n=80","n=100","n=120","n=140","n=160","n=180",
21         "n=200","n=400","n=600","n=800","n=1000","n=10000")
22
23 n1 = c(seq(20,200,20),400,600,800,1000,10000)   #Antall observasjoner
24 ap1 = 3:10                          #Parameter ap under pareto-modellen
25 nsim = length(n1)*length(ap1)             #Antall ulike scenarioer
26 nn = c(mapply(rep,n1,length(ap1)))        #Vektor med antall observasjoner for hvert ↩
       scenario
27 ap = rep(ap1,length(n1))              #Vektor med parameter ap for hvert scenario
28
29 #Funksjon som returnerer sannsynlighet for at vi velger riktig modell (pareto)
30 quick = function(mb=1e3,n=20,ap=3){
31 i = 1:n
32 p = (i-0.5)/n
33 check = rep(0,mb)
34 M = rep(0,2)
```

```
35 for (k in 1:mb){
36 z = rPareto(n,ap)       #Trekker underliggende data fra korrekt modell (pareto)
37 aG = optimize(llminc,c(0.001,1000),x=z)$minimum     #Estimerer alpha_hat under gamma↩
      -modellen ved ML
38 bG = n*aG/sum(z)        #Estimerer beta_hat under gamma-modellen ved ML
39 aP = n/sum(log(1+z))    #Finner ML-estimat for ap under pareto-modellen
40 qG = qgamma(p,aG,bG)    #Finner persentiler for tilpasset gamma-modell
41 qP = qPareto(p,aP)  #Finner persentiler for tilpasset pareto-modell
42 M = c(Qc(qG,z),Qc(qP,z))#Beregner Q-score
43 check[k] = which.min(M) #Velger modell med lavest Q-score
44 }
45 sum(check==2)/mb    #Sannsynlighet for å velge riktig modell (pareto)
46 }
47 system.time(quick(1e4,20,3))    #test for funksjonen quick
48
49 #Parallelliserer funksjonen quick og beregner hvor stor sannsynlighet det er for å ↩
      velge riktig
50 #modell for alle scenarioer (antall: nsim)
51 x <- foreach (j=1:nsim) %dopar% quick(mb=1e4,n=nn[j],ap=ap[j])
52 resultP = matrix(unlist(x),length(ap1),length(n1),dimnames=list(c(ap_text),c(n_text)↩
      ))
```

A.3.13 Figurer

Listing A.21: Figur 4.1

```
1  #Optimalt parametersett under gamma-modellen
2  alpha0 = c(100.169920, 25.166065, 11.273246, 6.412448, 4.159660, 2.933612,
3            2.192515, 1.710618, 1.377630, 1.137820)
4  beta0 = c(99.6712388, 24.6674451, 10.7771491, 5.9197978, 3.6707835, 2.4503209,
5            1.7161565, 1.2423191, 0.9187613, 0.6901873)
6
7  x = seq(0,10,0.01)
8  sigma = seq(0.1,1,0.1)  #Parameter sigma under korrekt modell log-normal
9  ##############################################################
10 plot(x,dgamma(x,alpha0[1],beta0[1]),"l",ylim=c(0,1),col="red",
11     main=expression("Tetthet til Gamma(" * alpha[0] * "," * beta[0] * ") og Log-↩
         normal(0," * sigma * ")"),
12     ylab="Tetthet", xlab="Skadestørrelse (Z)")
13 lines(x,dlnorm(x,0,sigma[1]),col="black")
14 lines(x,dgamma(x,alpha0[5],beta0[5]),"l",col="blue")
15 lines(x,dlnorm(x,0,sigma[5]),col="green")
16 lines(x,dgamma(x,alpha0[10],beta0[10]),"l",col="brown")
17 lines(x,dlnorm(x,0,sigma[10]),col="orange")
18 legend("topright", pch= 16, col=c("red","black","blue","green","brown","orange"),
19      legend=c(expression("Gamma, " * sigma * " = 0.1"),
20         expression("Log-normal, " * sigma * " = 0.1"),
21         expression("Gamma, " * sigma * " = 0.5"),
22         expression("Log-normal, " * sigma * " = 0.5"),
23         expression("Gamma, " * sigma * " = 1.0"),
24         expression("Log-normal, " * sigma * " = 1.0")))
```

Listing A.22: Figurer for relativ systematisk feil og total feil for gamma- mot log-normal-modellen

```
1  #Leser inn resultater for total feil og relativ systematisk feil i
```

```
#en 3D-matrise med dimensjon 10x15x11 (sigma x n x b)
tot_b = array(0,dim=c(10,15,11))
tot_b[,,1] = matrix(scan("total_b1.txt"),byrow=T,10)
tot_b[,,2] = matrix(scan("total_b2.txt"),byrow=T,10)
tot_b[,,3] = matrix(scan("total_b3.txt"),byrow=T,10)
tot_b[,,4] = matrix(scan("total_b4.txt"),byrow=T,10)
tot_b[,,5] = matrix(scan("total_b5.txt"),byrow=T,10)
tot_b[,,6] = matrix(scan("total_b6.txt"),byrow=T,10)
tot_b[,,7] = matrix(scan("total_b7.txt"),byrow=T,10)
tot_b[,,8] = matrix(scan("total_b8.txt"),byrow=T,10)
tot_b[,,9] = matrix(scan("total_b9.txt"),byrow=T,10)
tot_b[,,10] = matrix(scan("total_b10.txt"),byrow=T,10)
tot_b[,,11] = matrix(scan("total_bU.txt"),byrow=T,10)
RSF_b = array(0,dim=c(10,15,11))
RSF_b[,,1] = matrix(scan("RSF_b1.txt"),byrow=T,10)
RSF_b[,,2] = matrix(scan("RSF_b2.txt"),byrow=T,10)
RSF_b[,,3] = matrix(scan("RSF_b3.txt"),byrow=T,10)
RSF_b[,,4] = matrix(scan("RSF_b4.txt"),byrow=T,10)
RSF_b[,,5] = matrix(scan("RSF_b5.txt"),byrow=T,10)
RSF_b[,,6] = matrix(scan("RSF_b6.txt"),byrow=T,10)
RSF_b[,,7] = matrix(scan("RSF_b7.txt"),byrow=T,10)
RSF_b[,,8] = matrix(scan("RSF_b8.txt"),byrow=T,10)
RSF_b[,,9] = matrix(scan("RSF_b9.txt"),byrow=T,10)
RSF_b[,,10] = matrix(scan("RSF_b10.txt"),byrow=T,10)
RSF_b[,,11] = matrix(scan("RSF_bU.txt"),byrow=T,10)

n = c(seq(20,200,20),400,600,800,1000,10000)    #Antall observasjoner
sigma = seq(0.1,1,0.1)  #Parameter sigma i korrekt modell (log-normal)
colors= rainbow(15) #Farger i figur
leg_text=c("n =       20","n =       40","n =       60","n =       80","n =     100"↩
    ,"n =     120",
           "n =     140","n =     160","n =     180","n =     200","n =     400",
           "n =     600","n =     800","n =  1 000","n =10 000")

#Relativ systematisk feil
#b=1,2,3,4,5,6,7,8,9,10,ubegrenset(k=11)
k = 1      #BYTT UT k=b for andre skadebegrensningsnivåer (her: b=1)
plot(sigma,100*RSF_b[,1,k],"l",main="Relativ systematisk feil for Gamma mot Log-↩
    normal (b=1)",
    col=colors[1],ylim=c(0,100),ylab="Relativ systematisk feil (%)",xlab=expression(↩
        sigma))
abline(50,0)
for (i in 2:15){ lines(sigma,100*RSF_b[,i,k],col=colors[i]) }
legend(x=0.064,y=103, pch= 16, col=colors, legend=leg_text)
###############################################################
#Total feil
plot(sigma,tot_b[,1,k],"l",main="Total feil for Gamma mot Log-normal (b=1)",
    ylim=c(0,max(tot_b)),ylab="Total feil",xlab=expression(sigma),col=colors[1])
for (i in 2:15){ lines(sigma,tot_b[,i,k],col=colors[i]) }
legend("topleft", pch= 16, col=colors, legend=leg_text)
```

Listing A.23: Figurer for parameterfeil for gamma- mot log-normal-modellen

```
#Bruk mA, sA, mB og sB fra programmet som beregner
#parameterfeil for gamma- mot log-normal-modellen

#Optimalt parametersett under gamma-modellen
alpha0 = c(100.169920, 25.166065, 11.273246, 6.412448, 4.159660,
        2.933612, 2.192515, 1.710618, 1.377630, 1.137820)
beta0 = c(99.6712388, 24.6674451, 10.7771491, 5.9197978, 3.6707835,
      2.4503209, 1.7161565, 1.2423191, 0.9187613, 0.6901873)
```

```
9
10 sigma = seq(0.1,1,0.1)  #Parameter sigma under korrekt modell (log-normal)
11 colors = rainbow(15)    #Farger i figur
12 leg_text=c("n =      20","n =      40","n =      60","n =      80","n =     100","n =     120",
13            "n =     140","n =     160","n =     180","n =     200","n =     400",
14            "n =     600","n =     800","n =   1 000","n =10 000")
15 ############################################################
16 #Relativ estimeringsfeil av alpha
17 ESTa = 100*(mA-alpha0)/alpha0
18 plot(sigma,ESTa[,1],col=colors[1],type="l",ylim=c(0,37),
19     main=expression("Relativ estimeringsfeil av " * alpha),
20     xlab=expression(sigma),ylab="Relativ estimeringsfeil (%)")
21 for (i in 2:15){
22     lines(sigma,ESTa[,i],col=colors[i])
23 }
24 legend("topleft", pch= 16, col=colors, legend=leg_text)
25
26 #Relativt standardavvik for alpha_hat
27 sAm = 100*sA/mA
28 plot(sigma,sAm[,1],col=colors[1],type="l",ylim=c(0,70),
29     main=expression("Standardavvik målt i prosent av forventning for " * hat(alpha)),
30     xlab=expression(sigma),ylab="Relativt standardavvik (%)")
31 for (i in 2:15){
32     lines(sigma,sAm[,i],col=colors[i])
33 }
34 legend("topleft", pch= 16, col=colors, legend=leg_text)
35
36 #Relativ estimeringsfeil av beta
37 ESTb = 100*(mB-beta0)/beta0
38 plot(sigma,ESTb[,1],col=colors[1],type="l",ylim=c(0,37),
39     main=expression("Relativ estimeringsfeil av " * beta),
40     xlab=expression(sigma),ylab="Relativ estimeringsfeil (%)")
41 for (i in 2:15){
42     lines(sigma,ESTb[,i],col=colors[i])
43 }
44 legend("topleft", pch= 16, col=colors, legend=leg_text)
45
46 #Relativt standardavvik for beta_hat
47 sBm = 100*sB/mB
48 plot(sigma,sBm[,1],col=colors[1],type="l",ylim=c(0,70),
49     main=expression("Standardavvik målt i prosent av forventning for " * hat(beta)),
50     xlab=expression(sigma),ylab="Relativt standardavvik (%)")
51 for (i in 2:15){
52     lines(sigma,sBm[,i],col=colors[i])
53 }
54 legend("topleft", pch= 16, col=colors, legend=leg_text)
```

Listing A.24: Figur 4.11

```
1  dpareto = function(x,ap=3){ap/(1+x)^(ap+1)} #Tetthetsfunksjon for pareto-modellen
2
3  #Optimalt parametersett under gamma-modellen
4  alpha0 = c(0.7430800, 0.8069785, 0.8455711, 0.8710604, 0.8893478,
5             0.9031141, 0.9137544, 0.9223556)
6  beta0 = c(1.486143, 2.421529, 3.382436, 4.356468, 5.335372,
7             6.323013, 7.310311, 8.302472)
8
9  x = seq(0,10,0.01)
10 ap = 3:10        #Parameter ap under korrekt modell pareto
```

```
11 ############################################################
12 plot(x,dgamma(x,alpha0[1],beta0[1]),"l",ylim=c(0,1),col="red",
13     main=expression("Tetthet til Gamma(" * alpha[0] * ", " * beta[0] * ") og Pareto(↩
       " * alpha[p] * ")"),
14     ylab="Tetthet", xlab="Skadestørrelse (Z)")
15 lines(x,dpareto(x,ap[1]),col="black")
16 lines(x,dgamma(x,alpha0[4],beta0[4]),"l",col="blue")
17 lines(x,dpareto(x,ap[4]),col="green")
18 lines(x,dgamma(x,alpha0[8],beta0[8]),"l",col="brown")
19 lines(x,dpareto(x,ap[8]),col="orange")
20 legend("topright", pch= 16, col=c("red","black","blue","green","brown","orange"),
21      legend=c(expression("Gamma, " * alpha[p] * " = 3"),
22         expression("Pareto, " * alpha[p] * " = 3"),
23         expression("Gamma, " * alpha[p] * " = 6"),
24         expression("Pareto, " * alpha[p] * " = 6"),
25         expression("Gamma, " * alpha[p] * " = 10"),
26         expression("Pareto, " * alpha[p] * " = 10")))
```

Listing A.25: Figurer for relativ systematisk feil og total feil for gamma- mot pareto-modellen

```
1  #Leser inn resultater for total feil og relativ systematisk feil i
2  #en 3D-matrise med dimensjon 8x15x11 (ap x n x b)
3  tot_b = array(0,dim=c(8,15,11))
4  tot_b[,,1] = matrix(scan("total_b1.txt"),byrow=T,8)
5  tot_b[,,2] = matrix(scan("total_b2.txt"),byrow=T,8)
6  tot_b[,,3] = matrix(scan("total_b3.txt"),byrow=T,8)
7  tot_b[,,4] = matrix(scan("total_b4.txt"),byrow=T,8)
8  tot_b[,,5] = matrix(scan("total_b5.txt"),byrow=T,8)
9  tot_b[,,6] = matrix(scan("total_b6.txt"),byrow=T,8)
10 tot_b[,,7] = matrix(scan("total_b7.txt"),byrow=T,8)
11 tot_b[,,8] = matrix(scan("total_b8.txt"),byrow=T,8)
12 tot_b[,,9] = matrix(scan("total_b9.txt"),byrow=T,8)
13 tot_b[,,10] = matrix(scan("total_b10.txt"),byrow=T,8)
14 tot_b[,,11] = matrix(scan("total_bU.txt"),byrow=T,8)
15 RSF_b = array(0,dim=c(8,15,11))
16 RSF_b[,,1] = matrix(scan("RSF_b1.txt"),byrow=T,8)
17 RSF_b[,,2] = matrix(scan("RSF_b2.txt"),byrow=T,8)
18 RSF_b[,,3] = matrix(scan("RSF_b3.txt"),byrow=T,8)
19 RSF_b[,,4] = matrix(scan("RSF_b4.txt"),byrow=T,8)
20 RSF_b[,,5] = matrix(scan("RSF_b5.txt"),byrow=T,8)
21 RSF_b[,,6] = matrix(scan("RSF_b6.txt"),byrow=T,8)
22 RSF_b[,,7] = matrix(scan("RSF_b7.txt"),byrow=T,8)
23 RSF_b[,,8] = matrix(scan("RSF_b8.txt"),byrow=T,8)
24 RSF_b[,,9] = matrix(scan("RSF_b9.txt"),byrow=T,8)
25 RSF_b[,,10] = matrix(scan("RSF_b10.txt"),byrow=T,8)
26 RSF_b[,,11] = matrix(scan("RSF_bU.txt"),byrow=T,8)
27
28 n = c(seq(20,200,20),400,600,800,1000,10000)    #Antall observasjoner
29 ap = 3:10           #Parameter ap i korrekt modell (pareto)
30 colors= rainbow(15) #Farger i figur
31 leg_text=c("n =       20","n =       40","n =       60","n =       80","n =      100"↩
       ,"n =      120",
32            "n =      140","n =      160","n =      180","n =      200","n =      400",
33            "n =      600","n =      800","n =  1 000","n =10 000")
34
35 #Relativ systematisk feil
36 #b=1,2,3,4,5,6,7,8,9,10,ubegrenset(k=11)
37 k = 1       #BYTT UT k=b for andre skadebegrensningsnivåer (her: b=1)
38 plot(ap,100*RSF_b[,1,k],"l",main="Relativ systematisk feil for Gamma mot Pareto (b↩
       =1)",
```

```
39      col=colors[1],ylim=c(0,100),ylab="Relativ systematisk feil (%)",xlab=expression(↩
            alpha[p]))
40  abline(50,0)
41  for (i in 2:15){ lines(ap,100*RSF_b[,i,k],col=colors[i]) }
42  legend(x=8.83,y=102.7, pch= 16, col=colors, legend=leg_text)
43  ##################################################
44  #Total feil
45  plot(ap,tot_b[,1,k],"l",main="Total feil for Gamma mot Pareto (b=1)",
46      ylim=c(0,max(tot_b)),ylab="Total feil",xlab=expression(alpha[p]),col=colors[1])
47  for (i in 2:15){ lines(ap,tot_b[,i,k],col=colors[i]) }
48  legend("topright", pch= 16, col=colors, legend=leg_text)
```

Listing A.26: Figurer for parameterfeil for gamma- mot pareto-modellen

```
1   #Bruk mA, sA, mB og sB fra programmet som beregner
2   #parameterfeil for gamma- mot log-normal-modellen
3
4   #Optimalt parametersett under gamma-modellen
5   alpha0 = c(0.7430800, 0.8069785, 0.8455711, 0.8710604, 0.8893478,
6              0.9031141, 0.9137544, 0.9223556)
7   beta0 = c(1.486143, 2.421529, 3.382436, 4.356468, 5.335372,
8              6.323013, 7.310311, 8.302472)
9
10  ap = 3:10          #Parameter ap under korrekt modell (pareto)
11  colors = rainbow(15)    #Farger i figur
12  leg_text=c("n =      20","n =      40","n =      60","n =      80","n =     100"↩
        ,"n =     120",
13             "n =     140","n =     160","n =     180","n =     200","n =     400",
14             "n =     600","n =     800","n =  1 000","n =10 000")
15  ############################################################
16  #Relativ estimeringsfeil av alpha
17  ESTa = 100*(mA-alpha0)/alpha0
18  plot(ap,ESTa[,1],col=colors[1],type="l",ylim=c(0,37),
19      main=expression("Relativ estimeringsfeil av " * alpha),
20      xlab=expression(alpha[p]),ylab="Relativ estimeringsfeil (%)")
21  for (i in 2:15){
22      lines(ap,ESTa[,i],col=colors[i])
23  }
24  legend("topright", pch= 16, col=colors, legend=leg_text)
25
26  #Relativt standardavvik for alpha_hat
27  sAm = 100*sA/mA
28  plot(ap,sAm[,1],col=colors[1],type="l",ylim=c(0,70),
29      main=expression("Standardavvik målt i prosent av forventning for " * hat(alpha))↩
        ,
30      xlab=expression(alpha[p]),ylab="Relativt standardavvik (%)")
31  for (i in 2:15){
32      lines(ap,sAm[,i],col=colors[i])
33  }
34  legend("topright", pch= 16, col=colors, legend=leg_text)
35
36  #Relativ estimeringsfeil av beta
37  ESTb = 100*(mB-beta0)/beta0
38  plot(ap,ESTb[,1],col=colors[1],type="l",ylim=c(0,37),
39      main=expression("Relativ estimeringsfeil av " * beta),
40      xlab=expression(alpha[p]),ylab="Relativ estimeringsfeil (%)")
41  for (i in 2:15){
42      lines(ap,ESTb[,i],col=colors[i])
43  }
44  legend("topright", pch= 16, col=colors, legend=leg_text)
45
```

```
46 #Relativt standardavvik for beta_hat
47 sBm = 100*sB/mB
48 plot(ap,sBm[,1],col=colors[1],type="l",ylim=c(0,70),
49     main=expression("Standardavvik målt i prosent av forventning for " * hat(beta)),
50     xlab=expression(alpha[p]),ylab="Relativt standardavvik (%)")
51 for (i in 2:15){
52     lines(ap,sBm[,i],col=colors[i])
53 }
54 legend("topright", pch= 16, col=colors, legend=leg_text)
```

Listing A.27: Figur for sannsynlighet for å velge riktig modell for både log-normal- og pareto-modellen

```
1  #Tekst
2  sig_text = c("sigma=0.1","sigma=0.2","sigma=0.3","sigma=0.4","sigma=0.5","sigma=0.6"↩
     ,
3              "sigma=0.7","sigma=0.8","sigma=0.9","sigma=1.0")
4  ap_text = c("ap=3","ap=4","ap=5","ap=6","ap=7","ap=8","ap=9","ap=10")
5  n_text = c("n=20","n=40","n=60","n=80","n=100","n=120","n=140","n=160","n=180",
6          "n=200","n=400","n=600","n=800","n=1000","n=10000")
7  leg_text=c("n =        20","n =        40","n =        60","n =        80","n =      100"↩
     ,"n =      120",
8              "n =      140","n =      160","n =      180","n =      200","n =      400",
9              "n =      600","n =      800","n =  1 000","n =10 000")
10
11 n1 = c(seq(20,200,20),400,600,800,1000,10000)   #Antall observasjoner
12 sigma = seq(0.1,1,0.1)  #Parameter sigma under log-normal-modellen
13 ap = 3:10           #Parameter ap under pareto-modellen
14 colors= rainbow(length(n1)) #Farger i figur
15
16 #Henter resultat resultLN fra program som beregner sannsynlighet for å velge korrekt↩
      modell (log-normal)
17 LN = scan("resultLN.txt")
18 LNtab = 100*matrix(LN,byrow=T,length(sigma),length(n1),dimnames=list(c(sig_text),c(n↩
     _text)))
19
20 #Henter resultat resultP fra program som beregner sannsynlighet for å velge korrekt ↩
     modell (pareto)
21 P = scan("resultP.txt")
22 Ptab = 100*matrix(P,byrow=T,length(ap),length(n1),dimnames=list(c(ap_text),c(n_text)↩
     ))
23
24 #Log-normal
25 plot(sigma,LNtab[,1],"l",main="Sannsynlighet for å velge riktig modell (Log-normal)"↩
     ,
26         ylim=c(0,100),xlab=expression(sigma),ylab="Sannsynlighet",
27         col=colors[1],xlim=c(0.1,1.2))
28 for (i in 2:15){
29     lines(sigma,LNtab[,i],col=colors[i])
30 }
31 legend("topright", pch=16, col=colors, legend=leg_text)
32
33 #Pareto
34 plot(ap,Ptab[,1],"l",main="Sannsynlighet for å velge riktig modell (Pareto)",
35         ylim=c(0,100),xlim=c(3,11.5),xlab=expression(alpha[p]),
36         ylab="Sannsynlighet (%)", col=colors[1])
37 for (i in 2:15){
38     lines(ap,Ptab[,i],col=colors[i])
39 }
40 legend("topright", pch=16, col=colors, legend=leg_text)
```

Bibliografi

[1] Revolution Analytics. *doParallel: Foreach parallel adaptor for the parallel package*, 2012. URL `http://CRAN.R-project.org/package=doParallel`. R package version 1.0.1.

[2] Revolution Analytics. *foreach: Foreach looping construct for R*, 2012. URL `http://CRAN.R-project.org/package=foreach`. R package version 1.4.0.

[3] Erik Bølviken. Evaluating risk: A primer. Lecture notes STK4540, to be published by Cambridge University Press, 2011.

[4] Erik Bølviken. Modelling claim size. Lecture notes STK4540, to be published by Cambridge University Press, 2011.

[5] Jay L. Devore and Kenneth N. Berk. *Modern Mathematical Statistics with Applications*. Thomson Brooks/Cole, Duxbury, 2007.

[6] Luc Devroye and Lászlό Györfi. *Nonparametric Density Estimation: The L B1 S View*. Wiley, 1985.

[7] Dirk Eddelbuettel and Romain François. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011. URL `http://www.jstatsoft.org/v40/i08/`.

[8] Gene H Golub and John H Welsch. Calculation of gauss quadrature rules. *Mathematics of Computation*, 23(106):221–230, 1969.

[9] John R Hershey and Peder A Olsen. Approximating the kullback leibler divergence between gaussian mixture models. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 4, pages IV–317. IEEE, 2007.

[10] Peter J. Huber. The behavior of maximum likelihood estimates under nonstandard conditions. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 221–33, 1967.

[11] Wacek Kusnierczyk. *rbenchmark: Benchmarking routine for R*, 2012. URL `http://CRAN.R-project.org/package=rbenchmark`. R package version 1.0.0.

[12] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL `http://www.R-project.org/`. ISBN 3-900051-07-0.

[13] Oleg Sklyar, Duncan Murdoch, Mike Smith, Dirk Eddelbuettel, and Romain Francois. *inline: Inline C, C++, Fortran function calls from R*, 2013. URL `http://CRAN.R-project.org/package=inline`. R package version 0.3.11.

[14] Erwin Straub. *Non-Life Insurance Mathematics*. Springer-Verlag, Zürich, 1988.

[15] Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova. *snow: Simple Network of Workstations*, 2012. URL `http://CRAN.R-project.org/package=snow`. R package version 0.3-10.

[16] Hao Yu. Rmpi: Parallel statistical computing in r. *R News*, 2(2):10–14, 2002. URL `http://cran.r-project.org/doc/Rnews/Rnews_2002-2.pdf`.