

# MCMC - Tuning und Konvergenzdiagnostik

Volker Schmid

29. Mai 2017

## Beispiel MCMC bei Poisson-Lognormal

```
library(coda)
library(bayeskurs)
data("kaiserschnitt.raw")
data<-kaiserschnitt.raw

n <-dim(data)[1]
sumx <- sum(data$n)
mcmc.simple<-poisson.lognormal.mcmc(sumx,n)
```

# Konvergenz

MCMC-Algorithmen erzeugen eine Markovkette aus Ziehungen aus der Posteriori-Verteilung. Probleme:

- ▶ Der Algorithmus muss konvergieren, damit er aus der stationären Verteilung zieht
- ▶ Ziehungen sind automatisch abhängig. Damit ist der Algorithmus ineffizienter als unabhängiges Ziehen
- ▶ Die Effizienz hängt stark vom genauen Algorithmus ab, z.B. Metropolis-Hastings oder Gibbs-Sampler, Wahl der Vorschlagsdichte, etc..

# MCMC mit coda I

```
summary(mcmc.simple)
```

```
##  
## Iterations = 1:1000  
## Thinning interval = 1  
## Number of chains = 1  
## Sample size per chain = 1000  
##  
## 1. Empirical mean and standard deviation for each variable  
##    plus standard error of the mean:  
##  
##           Mean      SD Naive SE Time-series SE  
## mu          0.8579 0.5550  0.01755         0.19756  
## lambda 10.3986 0.6414  0.02028         0.03317  
##
```

## MCMC mit coda II

```
## 2. Quantiles for each variable:
```

```
##
```

```
##          2.5%    25%    50%    75%   97.5%
```

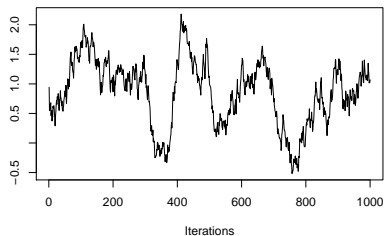
```
## mu      -0.264 0.4766 0.937  1.255  1.868
```

```
## lambda  9.232 9.9336 10.382 10.819 11.642
```

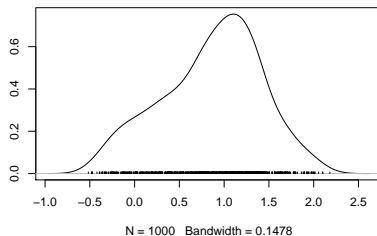
# MCMC mit coda

```
plot(mcmc.simple)
```

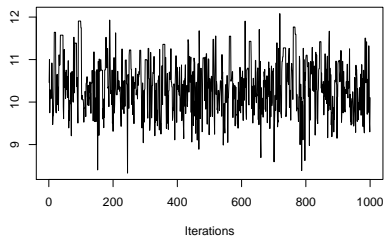
Trace of mu



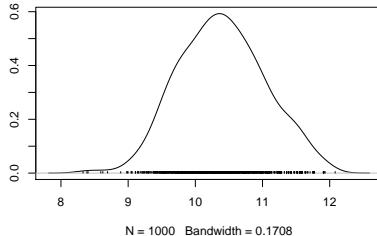
Density of mu



Trace of lambda



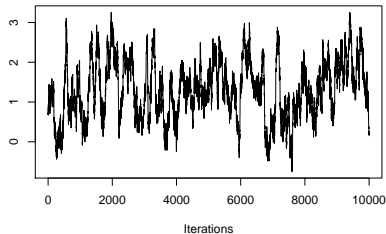
Density of lambda



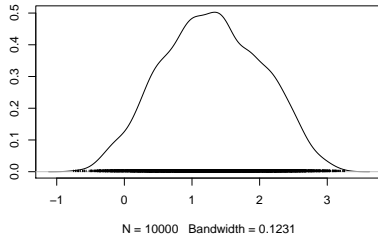
# Längere Ketten

```
mcmc.laenger<-poisson.lognormal.mcmc(sumx,n, I=10000)  
plot(mcmc.laenger)
```

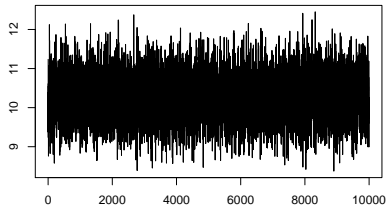
Trace of mu



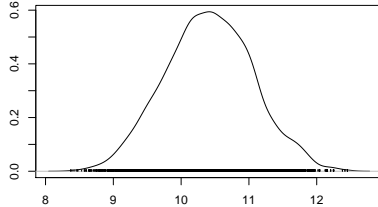
Density of mu



Trace of lambda



Density of lambda



# Tuning des Random Walk Proposals

Random Walk Proposal:

$$\theta^* = \theta^{(i-1)} + u$$

oft mit  $u \sim N(0, c)$ . Wie wählt man  $c$ ?

Akzeptanzrate = Anteil akzeptierter Vorschläge \* Niedrige

Akzeptanzrate: Kette bleibt oft im selben Zustand  $\rightarrow$  schlecht \* Zu

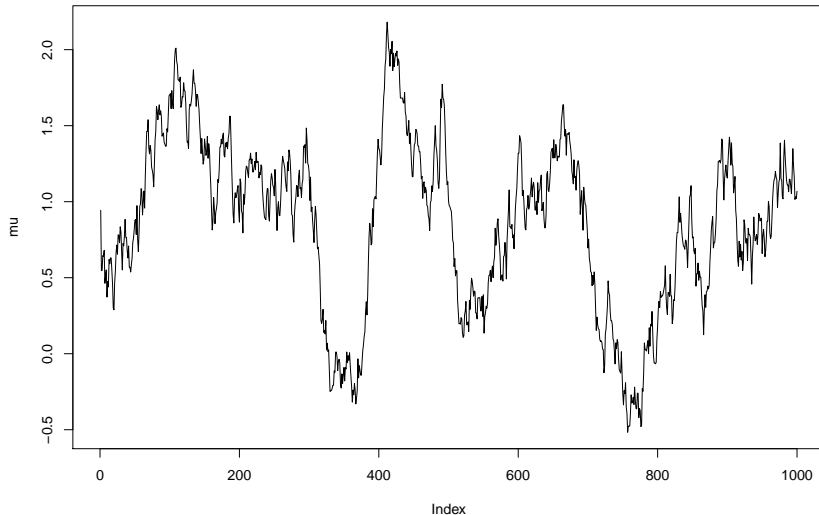
hohe Akzeptanzrate: Kette bewegt sich (eventuell) nur langsam  $\rightarrow$  schlecht

Tuning: Finde optimalen Wert für  $c$ . Für Random Walk Proposal gelten Akzeptanzraten zwischen ca. 30% und 60% als optimal (?)



## Beispiel zu hohe Akzeptanzrate

```
plot(as.vector(mcmc.simple[,1]),type="l",ylab="mu")
```



## Poisson-Lognormal mit Tuning

```
mcmc.tuning<-poisson.lognormal.mcmc(sumx,n,do.tuning=TRUE)
```

```
## Akzeptanzrate 0.94
```

```
## Akzeptanzrate 0.92
```

```
## Akzeptanzrate 0.84
```

```
## Akzeptanzrate 0.8
```

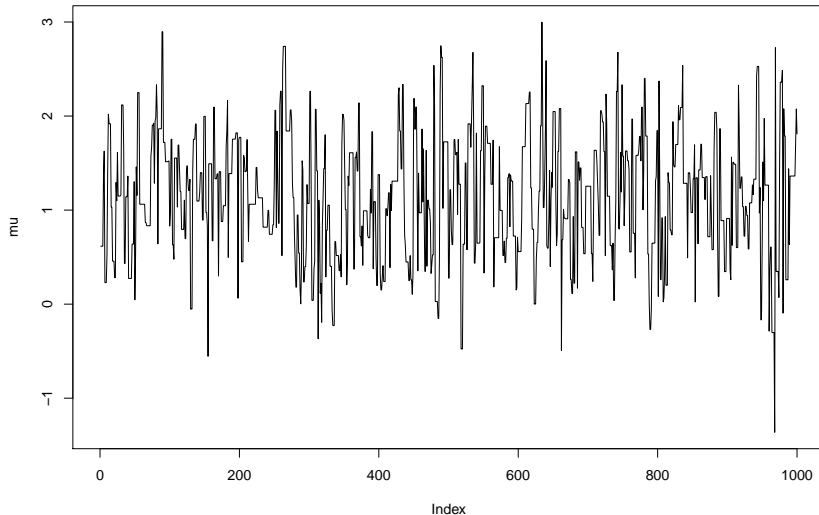
```
## Akzeptanzrate 0.56
```

```
## Akzeptanzrate 0.64
```

```
## Akzeptanzrate 0.46
```

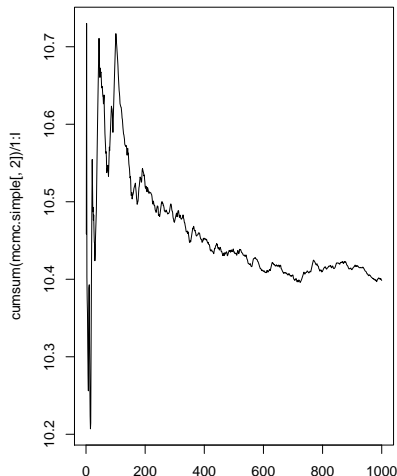
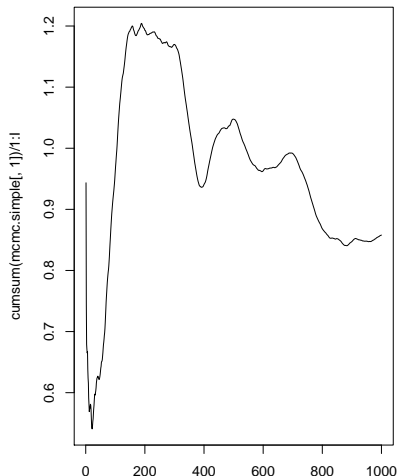
# Poisson-Lognormal mit Tuning

```
plot(as.vector(mcmc.tuning[,1]),type="l",ylab="mu")
```



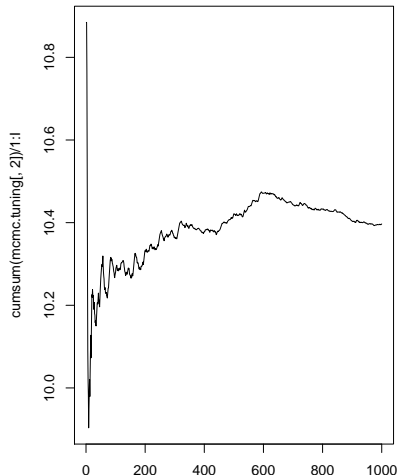
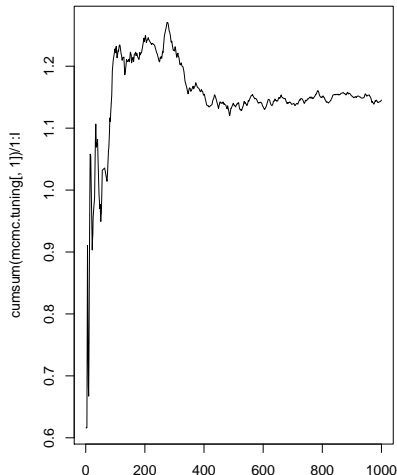
# Running mean plots

```
par(mfrow=c(1,2))  
I=1000  
plot(cumsum(mcmc.simple[,1])/1:I,type="l")  
plot(cumsum(mcmc.simple[,2])/1:I,type="l")
```



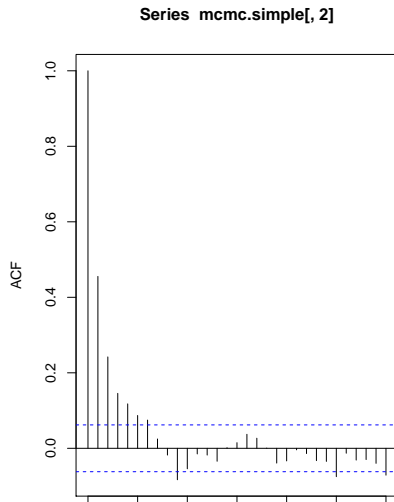
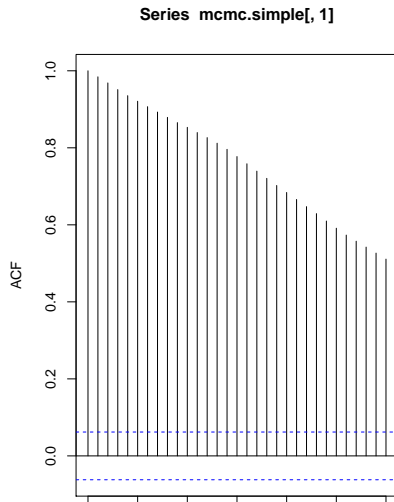
# Running mean plots

```
par(mfrow=c(1,2))  
I=1000  
plot(cumsum(mcmc.tuning[,1])/1:I,type="l",)  
plot(cumsum(mcmc.tuning[,2])/1:I,type="l")
```



# Auto correlation function

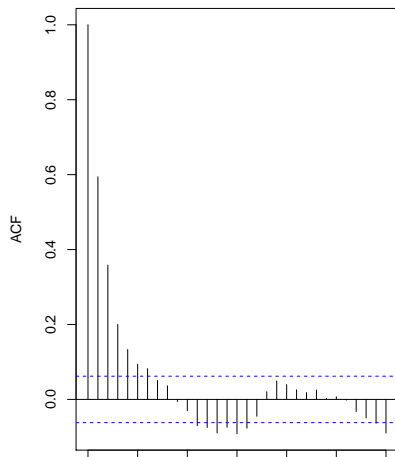
```
par(mfrow=c(1,2))  
acf(mcmc.simple[,1])  
acf(mcmc.simple[,2])
```



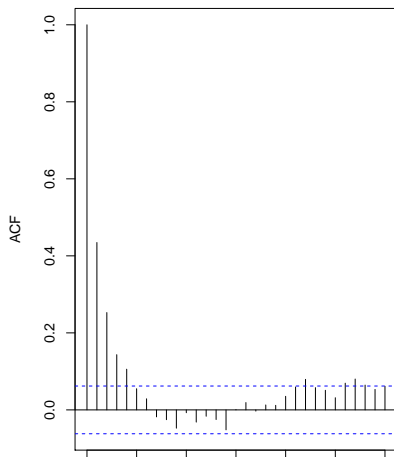
# Auto correlation function

```
par(mfrow=c(1,2))  
acf(mcmc.tuning[,1])  
acf(mcmc.tuning[,2])
```

Series mcmc.tuning[, 1]



Series mcmc.tuning[, 2]



# Konvergenzdiagnostik



# Gelman-Rubin-Diagnostik

- ▶ Idee: vergleiche die Varianz von mehreren parallel gelaufenen Ketten
- ▶ Bei Konvergenz sollte die Varianz in den Ketten der Varianz zwischen den Ketten entsprechen
- ▶ Within Chain Variance (unterschätzt die wahre Varianz, wenn Ketten noch nicht konvergiert)

$$W = \frac{1}{m} \sum_{j=1}^m s_j^2, s_j^2 = \frac{1}{n-1} \sum_{i=1}^n (\theta_{ij} - \bar{\theta}_j)^2$$

- ▶ Between Chain Variance

$$B = \frac{n}{m-1} \sum_{j=1}^m (\bar{\theta}_j - \bar{\bar{\theta}})^2; \bar{\bar{\theta}} = \frac{1}{m} \sum_{j=1}^m \bar{\theta}_j$$

# Gelman-Rubin-Diagnostik

- ▶ Geschätzte Varianz

$$\hat{Var}(\theta) = (1 - \frac{1}{n}W + \frac{1}{n}B)$$

- ▶ Potential scale reduction factor

$$\hat{R} = \sqrt{\frac{\bar{Var}(\theta)}{W}}$$

- ▶ Ist  $\hat{R}$  zu groß ( $> 1.1?$ ), sollten die Ketten länger laufen.

# Gelman-Rubin-Diagnostik

```
require(coda)
mc.list<-parallel::mclapply(rep(sumx, 5),
                           poisson.lognormal.mcmc,n=n)
print(gelman.diag(mc.list))
```

```
## Potential scale reduction factors:
```

```
##
```

```
##          Point est. Upper C.I.
```

```
## mu          1.04          1.11
```

```
## lambda      1.01          1.01
```

```
##
```

```
## Multivariate psrf
```

```
##
```

```
## 1.04
```

# Gelman-Rubin-Diagnostik

```
require(coda)
mc.list<-parallel::mclapply(rep(sumx, 5),
                           poisson.lognormal.mcmc,n=n,I=10000)
print(gelman.diag(mc.list))
```

```
## Potential scale reduction factors:
```

```
##
```

```
##          Point est. Upper C.I.
```

```
## mu          1.09          1.22
```

```
## lambda      1.00          1.00
```

```
##
```

```
## Multivariate psrf
```

```
##
```

```
## 1.1
```

# Gelman-Rubin-Diagnostik

- ▶ Gelman-Rubins  $R$  muss für jeden Parameter geschätzt werden
- ▶ Ziehungen aller Ketten werden dann zusammengeworfen
- ▶ Alternativ auch Aufteilung einer Kette möglich

# Geweke-Diagnostik

- ▶ Idee: Teste, ob zwei Teile einer Kette aus der selben Verteilung stammen (Teste Differenz der Mittelwerte)
- ▶ In der Regel erste 10% und letzte 50% der Kette

```
print(geweke.diag(mcmc.simple))
```

```
##  
## Fraction in 1st window = 0.1  
## Fraction in 2nd window = 0.5  
##  
##      mu lambda  
## 0.9507 2.1244
```

# Geweke-Diagnostik

```
print(geweke.diag(mcmc.tuning))
```

```
##  
## Fraction in 1st window = 0.1  
## Fraction in 2nd window = 0.5  
##  
##      mu  lambda  
## 0.4567 -0.7459
```

# Raftery und Lewis Diagnostik

- ▶ Wir interessieren uns für ein Quantil  $q$ .
- ▶ Wieviele Iterationen  $N$  und welchen burn-in  $M$  brauchen wir, um mit Wahrscheinlichkeit  $s$  innerhalb der Toleranz  $r$  das Quantil  $q$  zu schätzen?
- ▶ Nach einer Pilotkette lassen wir die längere Kette entsprechend laufen.



# Raftery und Lewis Diagnostik

```
print(raftery.diag(mcmc.simple,  
                  q = 0.5, r = 0.05, s = 0.9))
```

```
##  
## Quantile (q) = 0.5  
## Accuracy (r) = +/- 0.05  
## Probability (s) = 0.9  
##  
##          Burn-in   Total Lower bound   Dependence  
##          (M)       (N)   (Nmin)         factor (I)  
##  mu          95      8355   271          30.80  
##  lambda 6       563    271          2.08
```

# Heidelberg und Welch Diagnostik

- ▶ Teste, ob die Kette aus einer stationären Verteilung kommt
- ▶ Zuerst: Cramer-von Mises-Test mit Niveau  $\alpha$  auf ganzer Kette
- ▶ Falls Nullhypothese abgelehnt, verwirf erste 10%, 20%, ..., bis zu 50%
- ▶ Falls Nullhypothese angenommen: Half-width-test
- ▶ Berechne  $(1 - \alpha)$ -Kreditibilitätsintervall (KI)
- ▶ Teststatistik: Hälfte der Breite des KI durch Mittelwert

# Heidelberg und Welch Diagnostik

```
print(heidel.diag(mcmc.simple))
```

```
##
##      Stationarity start      p-value
##      test          iteration
## mu      passed      1         0.332
## lambda passed      1         0.131
##
##      Halfwidth Mean    Halfwidth
##      test
## mu      failed      0.858 0.387
## lambda passed      10.399 0.065
```

# Heidelberg und Welch Diagnostik

```
print(heidel.diag(mcmc.tuning))
```

```
##
##      Stationarity start      p-value
##      test          iteration
## mu      passed      1         0.4992
## lambda passed      1         0.0917
##
##      Halfwidth Mean  Halfwidth
##      test
## mu      passed      1.15 0.0807
## lambda passed      10.40 0.0714
```