# TMA4300 Computer Intensive Statistical Methods Exercise 2, Spring 2023, Group 8

August Arnstad, Johan Sæbø

## Contents



# Task 1: MCMC theory, simulation and analysis of the Tokyo rainfall dataset

In this problem, we will look at a portion of the Tokyo rainfall dataset which contains daily rainfall data from 1951-1989. We will consider the response to be whether the amount of rainfall exceeded 1mm over the given time period:

$$y_t \mid \tau_t \sim \text{Bin}(n_t, \pi(\tau_t)), \quad \pi(\tau_t) = \frac{\exp\{\tau_t\}}{1 + \exp\{\tau_t\}}$$

for $n_t$ being 10 for $t = 60$ (February 29th) and 39 for all other days in the year, and $\pi(\tau_t)$ being the probability of rainfall exceeding 1mm for days t = 1, . . . , T and T = 366. Note that $\tau_t$ is the logit probability of exceedence and can be obtained from $\pi(\tau_t)$ ($\pi(\cdot)$ is known as the 'expit' or 'inverse logit' function) via the logit function: $\tau_t = log(\pi(\tau_t)/(1 - \pi(\tau_t)))$. We assume conditional independence among the $y_t|\tau_t$ for all $t = 1, ..., 366$.

## Task 1a): Exploration of data

One start by downloading and examining the given data of the Tokyo rainfall

```
load("rain.rda")
```

```
head(rain)
```

```
##   day n.years n.rain
## 1   1      39      8
```

```
## 2   2      39      7
## 3   3      39      8
## 4   4      39     11
## 5   5      39      8
## 6   6      39      6
```

```
summary(rain)
```

```
##       day            n.years          n.rain
##  Min.   :  1.00   Min.   :10.00   Min.   : 0.00
##  1st Qu.: 92.25   1st Qu.:39.00   1st Qu.: 8.00
##  Median :183.50   Median :39.00   Median :11.00
##  Mean   :183.50   Mean   :38.92   Mean   :10.98
##  3rd Qu.:274.75   3rd Qu.:39.00   3rd Qu.:14.00
##  Max.   :366.00   Max.   :39.00   Max.   :23.00
```

Then one can plot the data.

```
plot = ggplot(data = rain) + geom_line(aes(x = day, y = n.rain)) + geom_smooth(aes(x = day,
    y = n.rain), color = "red") + ggtitle("Rain") + ylab("Number of days with over 1 mm rain") +
    xlab("Day of the year") + labs(title = "Rain from the Tokyo rainfall dataset")
plot
```

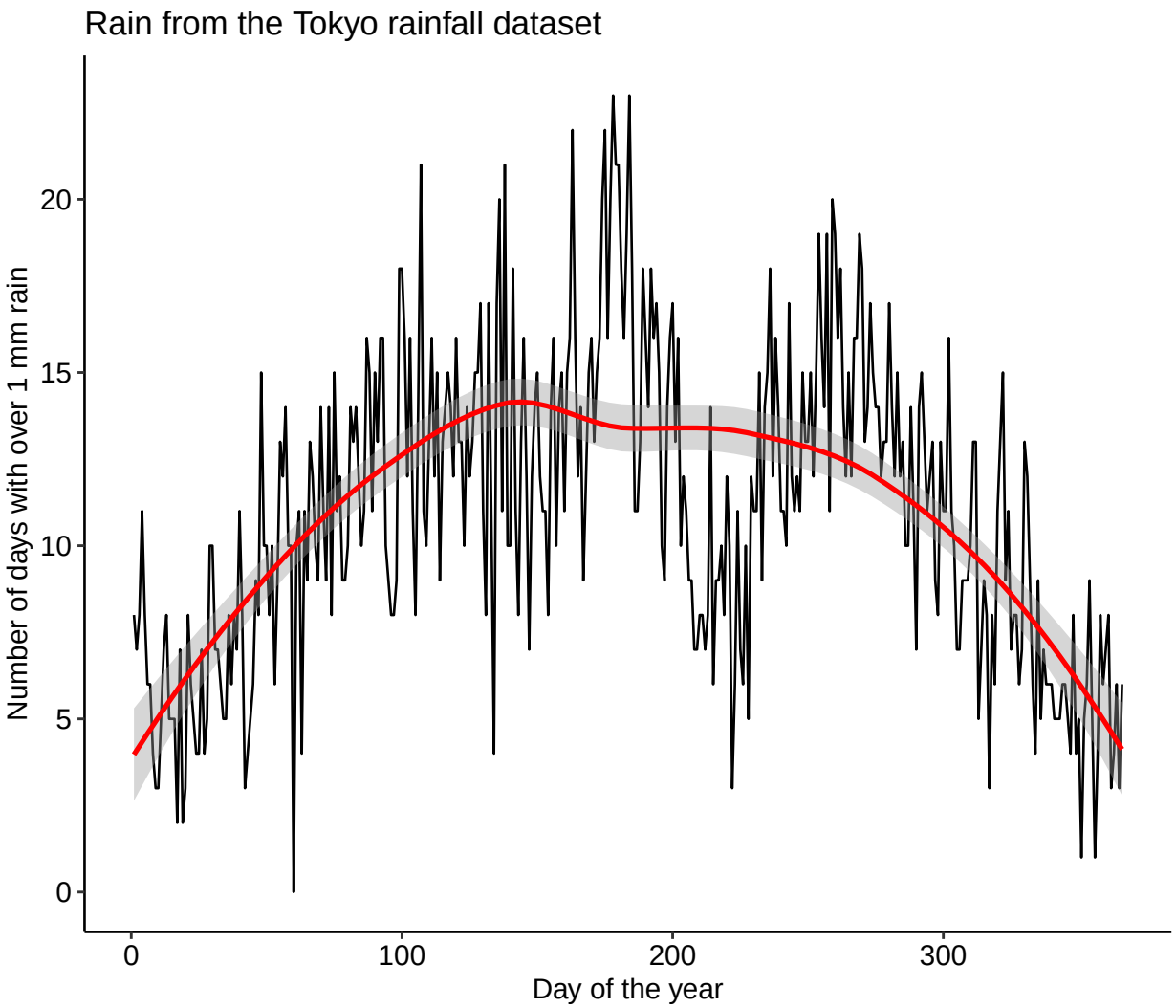


Figure 1: The total number of days from 1951 to 1989 with more than 1mm of rain in Tokyo and with the smoothed mean in red.

Looking at the plot, one can see that there is less rain at the start and end of a year, and with a rain season during the summer. The variation in the middle of the year also varies more than the beginning and end.

## Task 1b): Obtaining likelihood

To obtain the likelihood of $y_t$, which is depended upon on the parameters $\pi(\tau_t)$ for $t = 1, 2, ...366$ and is binomial distribution, as

$$p(y_t|\pi(\tau_t)) = \binom{n_t}{y_t} \pi(\tau_t)^{y_t} (1 - \pi(\tau_t))^{n_t - y_t}$$

which written out with the expression for $\pi(\tau_t)$ becomes

$$p(y_t \mid \pi(\tau_t)) = \prod_{i=1}^{T} \binom{n_t}{y_t} \left(\frac{1}{1+e^{-\tau_t}}\right)^{y_t} \left(1 - \frac{1}{1+e^{-\tau_t}}\right)^{n_t - y_t}$$

## Task 1c): Finding the conditional distribution of the variance term in the random walk

The task is to find the conditional $p(\sigma_u^2 \mid y, \tau)$, and name the distribution and the associated parameters, if it has a name. Starting with Bayes rule to get

$$p(\sigma_u^2 \mid \boldsymbol{\tau}, \boldsymbol{y}) =\propto p(\boldsymbol{y}, \boldsymbol{\tau} \mid \sigma_u^2)\ p(\sigma_u^2) = p(\sigma_u^2, \boldsymbol{\tau}, \boldsymbol{y}) = p(\boldsymbol{y}|\sigma_u^2, \boldsymbol{\tau})\ p(\boldsymbol{\tau}|\sigma_u^2)\ p(\sigma_u^2)$$

where the last term comes from utilizing the chain-rule. Given that $y_t|\tau_t$ is independent of $\sigma_u^2$, one get simply

$$p(\sigma_u^2 \mid \boldsymbol{\tau}, \boldsymbol{y}) \propto\ p(\boldsymbol{\tau}|\sigma_u^2)\ p(\sigma_u^2).$$

Having these two expressions defined in the task description, one can plug them in to get

$$p(\sigma_u^2 \mid \boldsymbol{\tau}, \boldsymbol{y}) \propto \prod_{i=2}^{T} \frac{1}{\sigma_u} \exp\Big\{-\frac{1}{2\sigma_u^2}(\tau_t - \tau_{t-1})^2\Big\} \cdot \frac{\beta^\alpha}{\Gamma(\alpha)} (1/\sigma_u^2)^{\alpha+1} \exp\big\{-\beta/\sigma_u^2\big\}$$

$$\propto (1/\sigma_u^2)^{\alpha+1} \exp\{-\beta/\sigma_u^2\} \exp\Big\{\sum_{t=2}^{T} -\frac{1}{2\sigma_u^2}(\tau_t - \tau_{t-1})^2\Big\} = (1/\sigma_u^2)^{(\frac{T-1}{2}+\alpha+1)} \exp\Big\{-\frac{1}{\sigma_u^2}\big(\beta + 1/2 \sum_{t=2}^{T}(\tau_t - \tau_{t-1})^2\big)\Big\}$$

$$\propto \prod_{i=2}^{T} \frac{1}{\sigma_u} \exp\Big\{-\frac{1}{2\sigma_u^2}(\tau_t - \tau_{t-1})^2\Big\} \cdot \frac{\beta^\alpha}{\Gamma(\alpha)} (1/\sigma_u^2)^{\alpha+1} \exp\big\{-\beta/\sigma_u^2\big\}$$

$$\propto (1/\sigma_u^2)^{\alpha+1} \exp\{-\beta/\sigma_u^2\}\ \exp\Big\{\sum_{t=2}^{T} -\frac{1}{2\sigma_u^2}(\tau_t - \tau t - 1)^2\Big\}$$

which with the parameters $\alpha^* = \alpha + \frac{T-1}{2}$ and $\beta^* = \beta + 1/2 \sum_{t=2}^{T}(\tau_t - \tau_{t-1})^2$ becomes the inverse gamma distribution:

$$\sigma_u^2 \mid \boldsymbol{\tau}, \boldsymbol{y}\ \sim\ \text{Inverse Gamma}\Big(\alpha + \frac{T-1}{2}, \beta + \frac{1}{2}\sum_{t=2}^{T}(\tau_t - \tau_{t-1})^2\Big)$$

## Task 1d): Obtaining the acceptance probability

Consider the conditional prior proposal distribution, $Q(\boldsymbol{\tau'_{\mathcal{I}}} \mid \boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2, \boldsymbol{y}) = p(\boldsymbol{\tau'_{\mathcal{I}}} \mid \boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)$, where $\boldsymbol{\tau'_{\mathcal{I}}}$ is the proposed values for $\boldsymbol{\tau_{\mathcal{I}}}, \mathcal{I} \subseteq 1, ..., 366$ is a set of time indices, and $\boldsymbol{\tau_{-\mathcal{I}}} = \boldsymbol{\tau}_{\{1,...,366\}/\mathcal{I}}$ is $\boldsymbol{\tau}$ subset to include all indices other than those in $\mathcal{I}$. Show that the resulting acceptance probability is given by the ratio of likelihoods:

$$\alpha(\boldsymbol{\tau'_{\mathcal{I}}} \mid \boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2, \boldsymbol{y}) = \min\Big\{1,\ \frac{p(\boldsymbol{y}|\boldsymbol{\tau'_{\mathcal{I}}})}{p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}})}\Big\}.$$

To start one must use the formula for acceptance probability

$$\alpha(y|x) = \min\Big\{1,\ \frac{\pi(y)}{\pi(x)}\ \frac{Q(x \mid y)}{Q(y \mid x)}\Big\}.$$

Inserting into the general formula, one gets

$$\alpha(\boldsymbol{\tau'_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2, \boldsymbol{y}) = \min\Big\{1, \frac{\pi(\tau'_{\mathcal{I}}|\tau_{-\mathcal{I}}, \sigma_u^2, \boldsymbol{y})}{\pi(\tau_{\mathcal{I}}|\tau_{-\mathcal{I}}, \sigma_u^2, \boldsymbol{y})} \frac{Q(\tau_{\mathcal{I}}|\tau'_{\mathcal{I}}, \tau_{-\mathcal{I}}, \sigma_u^2, \boldsymbol{y})}{Q(\tau'_{\mathcal{I}}|\tau_{\mathcal{I}}, \tau_{-\mathcal{I}}, \sigma_u^2, \boldsymbol{y})}\Big\}.$$

Since this eqution holds $Q(\boldsymbol{\tau'_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2, \boldsymbol{y}) = p(\boldsymbol{\tau'_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)$, one can start by calculating an expression for $p(\boldsymbol{\tau_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)$ using statistical properties

$$p(\boldsymbol{\tau_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2, \boldsymbol{y}) = \frac{p(\boldsymbol{\tau_{\mathcal{I}}}, \boldsymbol{y}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}{p(\boldsymbol{y}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)} = \frac{p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}}, \boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2) \cdot p(\boldsymbol{\tau_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}{p(\boldsymbol{y}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}$$

and from the assumption of conditional independence in the problem description of Problem A that $y_t|\tau_t$ for all $t = 1, ..., 366$ and does not depend upon $\sigma_u^2$, one get

$$p(\boldsymbol{\tau_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \boldsymbol{y}) = \frac{p(\boldsymbol{y_{\mathcal{I}}}|\boldsymbol{\tau_{\mathcal{I}}}) \cdot p(\boldsymbol{y_{-\mathcal{I}}}|\boldsymbol{\tau_{-\mathcal{I}}}) \cdot p(\boldsymbol{\tau_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}{p(\boldsymbol{y}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}$$

Proceeding in the same manner for $p(\boldsymbol{\tau'_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \boldsymbol{y})$ one get

$$p(\boldsymbol{\tau'_{\mathcal{I}}}|\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2, \boldsymbol{y}) = \frac{p(\boldsymbol{y_{\mathcal{I}}}|\boldsymbol{\tau'_{\mathcal{I}}}) \cdot p(\boldsymbol{y_{-\mathcal{I}}}|\boldsymbol{\tau_{-\mathcal{I}}}) \cdot p(\boldsymbol{\tau'_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}{p(\boldsymbol{y}|\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}$$

Inserting the last two expression into the formula for the acceptance probability, one get

$$\alpha(\boldsymbol{\tau'_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2, \boldsymbol{y}) = \min\left\{1, \frac{\frac{p(\boldsymbol{y_{\mathcal{I}}}|\boldsymbol{\tau'_{\mathcal{I}}}) \cdot p(\boldsymbol{y_{-\mathcal{I}}}|\boldsymbol{\tau_{-\mathcal{I}}}) \cdot p(\boldsymbol{\tau'_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}{p(\boldsymbol{y}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}}{\frac{p(\boldsymbol{y_{\mathcal{I}}}|\boldsymbol{\tau_{\mathcal{I}}}) \cdot p(\boldsymbol{y_{-\mathcal{I}}}|\boldsymbol{\tau_{-\mathcal{I}}}) \cdot p(\boldsymbol{\tau_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}{p(\boldsymbol{y}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2)}} \ \frac{p(\boldsymbol{\tau_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \boldsymbol{y})}{p(\boldsymbol{\tau'_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \boldsymbol{y})}\right\}.$$

which reduces neatly to the desired expression

$$\alpha(\boldsymbol{\tau'_{\mathcal{I}}}\ |\boldsymbol{\tau_{-\mathcal{I}}}, \sigma_u^2, \boldsymbol{y}) = \min\Big\{1, \ \frac{p(\boldsymbol{y}|\boldsymbol{\tau'_{\mathcal{I}}})}{p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}})}\Big\}.$$

## Task 1e): Implementation of single updating MCMC

One can implement an MCMC sampler for the posterior $p(\boldsymbol{\pi}, \sigma_u^2|\boldsymbol{y})$ using MH steps for individual $\tau_t$ parameters using the conditional prior, $p(\tau_t|\boldsymbol{\tau_{-t}}, \sigma_u^2)$, and Gibbs steps for $\sigma^2$. Using the inverse gamma distribution, one can sample $\sigma_u^2$ for the Gibbs step. With the MH step, one must acquire the conditional prior $p(\tau_t|\boldsymbol{\tau_{-t}}, \sigma_u^2)$, which can be done by using a RW(1) model

$$\tau_t \sim \tau_{t-1} + u_t$$

with $u_t \sim \mathcal{N}(0, \sigma_u^2)$.

$$P(\boldsymbol{\tau}|\sigma_u^2) = \prod_{i=2}^{T} \frac{1}{\sigma_u} \exp\Big\{-\frac{1}{2\sigma_u^2}(\tau_t - \tau_{t-1})^2\Big\} \ \propto \ \exp\Big\{-\frac{1}{2\sigma_u^2}(\tau_t - \tau_{t-1})^2\Big\}$$

The last term can be rewritten as

$$\exp\Big\{-\frac{1}{2\sigma_u^2}\boldsymbol{\tau^T Q \tau}\Big\}$$

where $\boldsymbol{Q}$ is defined as

$$\boldsymbol{Q} = \begin{bmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 1 \end{bmatrix}$$

Now one can show that $\boldsymbol{\tau}|\sigma_u^2 \sim MVN(\boldsymbol{\mu}, \sigma_u^2 \boldsymbol{Q^{-1}})$ with $\boldsymbol{\mu}$ being the assumed constant mean vector.

Some useful facts that is helpful for the task is knowing that for a multivariate Gaussian,

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{x_A} \\ \boldsymbol{x_B} \end{pmatrix} \sim MVN\left( \left( \begin{pmatrix} \boldsymbol{\mu_A} \\ \boldsymbol{\mu_B} \end{pmatrix} \right), \left( \begin{pmatrix} \boldsymbol{Q_{AA}} & \boldsymbol{Q_{AB}} \\ \boldsymbol{Q_{BA}} & \boldsymbol{Q_{BB}} \end{pmatrix} \right)^{-1} \right),$$

with the conditional mean and precision for $\boldsymbol{x_A}|\boldsymbol{x_B}$ is given by:

$$\boldsymbol{\mu}_{A|B} = \boldsymbol{\mu}_A - \boldsymbol{Q}_{AA}^{-1}\boldsymbol{Q}_{AB}\ (\boldsymbol{x}_B - \boldsymbol{\mu}_B) \quad \text{and} \quad \boldsymbol{Q}_{A|B} = \boldsymbol{Q}_{AA}.$$

Given that $\boldsymbol{\mu}$ is assumed to be constant, making $\boldsymbol{\mu}_A$ and $\boldsymbol{\mu}_B$ constant. The conditional mean thereby becomes

$$\boldsymbol{\mu}_{A|B} = -\boldsymbol{Q}_{AA}^{-1}\boldsymbol{Q}_{AB}\ (\boldsymbol{x}_B)$$

Due to only a singly value of $\tau$ is needed for each step for the single site MCMC sampler, then one get this simplified expression of the conditional distribution.

$$\tau_t'|\boldsymbol{\tau_{-t}} \sim \mathcal{N}\left( \begin{cases} \tau_2\ , & t = 1 \\ \frac{1}{2}(\tau_{t+1} + \tau_{t-1})\ , & t \in \{2, ..., 365\} \\ \tau_{365}\ , & t = 366 \end{cases} , \sigma_u^2 \cdot \begin{cases} 1, & t = 1, 366 \\ \frac{1}{2}, & t \in \{2, ..., 365\} \end{cases} \right)$$

The next step is to find the fraction

$$\frac{p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}'})}{p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}})}$$

in the expression for the acceptance probability $\alpha$, one can again simplify. One begins with the expression for $p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}'})$ from task 1b) and then take the log

$$\ln(p(\boldsymbol{y}\ |\boldsymbol{\tau_{\mathcal{I}}'})) = y_t \ln(\pi(\tau\_t)) + (y_t - n_t)\ln(1 - \pi(\tau\_t))$$

and substituting the expression for $\pi(\cdot)$

$$\ln(p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}'})) = y_t[\ln(e^{\tau_t}) - \ln(e^{1+\tau_t})] + (y_t - n_t)[-\ln(1 + e^{\tau_t})]$$

which simplified finally becomes

$$\ln(p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}'})) = y_t\tau_t - n_t\ \ln(1 + e^{\tau_t})$$

Returning to the fraction, one can now write

$$\frac{p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}'})}{p(\boldsymbol{y}|\boldsymbol{\tau_{\mathcal{I}}})} = \frac{\exp\{y_t\tau_t^{\text{t}} - n_t\ \ln(1 + e^{\tau_t^{\text{t}}})\}}{\exp\{y_t\tau_t - n_t\ \ln(1 + e^{\tau_t})\}} = \exp\{y_t(\tau_t^{\text{t}} - \tau_t) + n_t \ln\frac{1 + e^{\tau_t}}{1 + e^{\tau_t^{\text{t}}}}\}$$

Finally one can write the code for the algorithm:

```r
set.seed(123)

alpha_fraction = function(n, y, tau.prop, tau) {
    return(exp(y * (tau.prop - tau) + n * log((1 + exp(tau))/(1 + exp(tau.prop)))))
}

MH_step <- function(tau, sigma2, t) {
    # The 3 different cases for values of t
    if (t == 1) {
        μ_tau <- tau[2]
    } else if (t == 366) {
        μ_tau <- tau[365]
    } else {
        μ_tau <- (tau[t + 1] + tau[t - 1])/2
        sigma2 <- sigma2/2
    }

    new_tau <- rnorm(1, μ_tau, sqrt(sigma2))
    accept_prob <- min(c(1, alpha_fraction(rain$n.years[t], rain$n.rain[t], new_tau,
        tau[t])))

    if (runif(1) < accept_prob) {
        # u< acceptance criteria(alpha)
        return(list(tau = new_tau, accept = 1))  #Accepted with new value
    } else {
        return(list(tau = tau[t], accept = 0))  #Rejected with copy of previous value
    }
}

MCMC <- function(alpha, beta, n) {
    t_max <- 366
    sigma2_vec <- numeric(n)  #Init sigma vec
    # Get initial first value based on the prior
    sigma2_vec[1] <- sqrt(1/rgamma(1, alpha, rate = beta))
    tau <- matrix(0, nrow = n, ncol = 366)
    tau[1, ] <- rnorm(t_max)  #Init first year sample
    accept <- 0

    for (i in 2:n) {
        tau_old <- tau[i - 1, ]
        sigma2 <- sigma2_vec[i - 1]
        for (t in 1:t_max) {
            res <- MH_step(tau_old, sigma2, t)
            tau[i, t] <- res$tau
            accept <- accept + res$accept
        }
        beta_star <- beta + sum((tau[i, -t_max] - tau[i, -1])^2)/2  #Beta as defined in task 1c)
        sigma2_vec[i] <- 1/rgamma(1, alpha + (t_max - 1)/2, scale = beta_star)  #Gibbs step
    }
    cat("Acceptance rate:", round(accept/(n * t_max) * 100, 3), "%\n")
    return(list(tau = tau, sigma = sigma2_vec))
}
```

```r
set.seed(123)
n <- 50000
alpha = 2
beta = 0.05

t <- proc.time()[3]
tau.list <- MCMC(alpha, beta, n)
```

```
## Acceptance rate: 93.277 %
```

```r
runtime_single <- proc.time()[3] - t
cat("Running time:", runtime_single, "seconds \n")
```

```
## Running time: 143.886 seconds
```

```r
set.seed(123)
tau <- tau.list$tau
sigma2 <- tau.list$sigma
```

```r
set.seed(123)
pi <- function(tau) {
    return(1/(1 + exp(-tau)))
}
```

```r
set.seed(123)
par(mfrow = c(2, 2), mar = c(5, 5, 0, 0))
plot(1:length(pi(tau[, 201])), pi(tau[, 1]), xlab = "Sample", type = "l", ylab = expression(pi[1]))
plot(1:length(pi(tau[, 201])), pi(tau[, 201]), xlab = "Sample", type = "l", ylab = expression(pi[201]))
plot(1:length(pi(tau[, 366])), pi(tau[, 366]), xlab = "Sample", type = "l", ylab = expression(pi[366]))
plot(1:length(sigma2), sigma2, xlab = "Sample", type = "l", ylim = c(0, 0.11), ylab = expression(sigma[
```

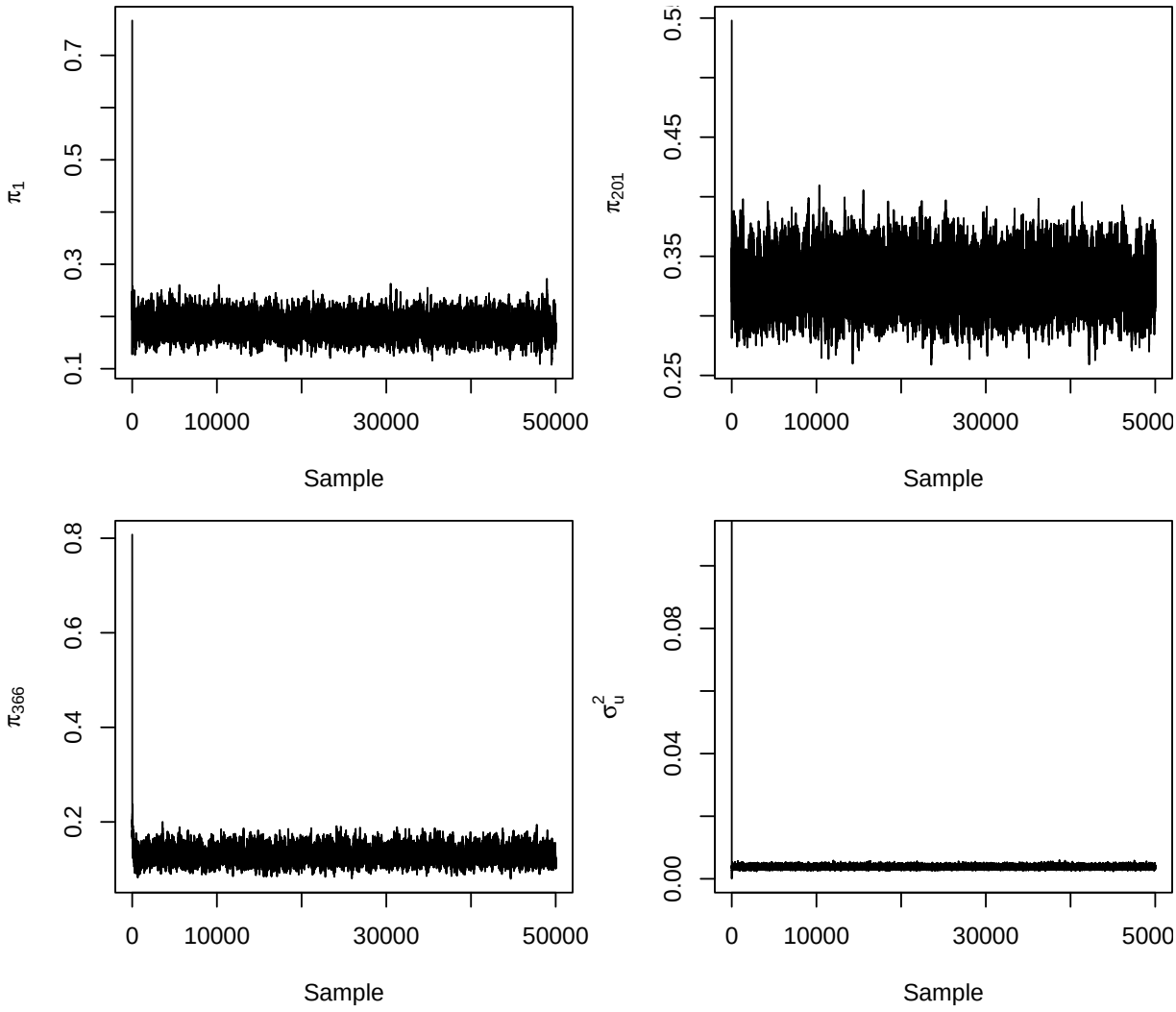


Figure 2: Trace plots of $\pi_1$, $\pi_{201}$, $\pi_{366}$ and $\sigma_u^2$ with 50 000 samples produced by a MCMC where every element of $\boldsymbol{\tau}$ is updated per iteration.

The four plots shows that there is a lot of variation in the beginning, before a state of apparent random noise takes place and therefore be converged. If one for example having the first 100 samples removed as burn-in samples, one get converged traceplots:

```
set.seed(123)
burn_period = 1000
par(mfrow = c(2, 2), mar = c(5, 5, 0, 0))
plot(1:length(pi(tau[burn_period:n, 1])), pi(tau[burn_period:n, 1]), type = "l",
    ylab = expression(pi[1]), xlab = "Sample")
plot(1:length(pi(tau[burn_period:n, 201])), pi(tau[burn_period:n, 201]), type = "l",
    ylab = expression(pi[201]), xlab = "Sample")
plot(1:length(pi(tau[burn_period:n, 366])), pi(tau[burn_period:n, 366]), type = "l",
    ylab = expression(pi[366]), xlab = "Sample")
plot(1:length(sigma2[burn_period:n]), sigma2[burn_period:n], type = "l", ylab = expression(sigma["u"]^2
    xlab = "Sample")
```

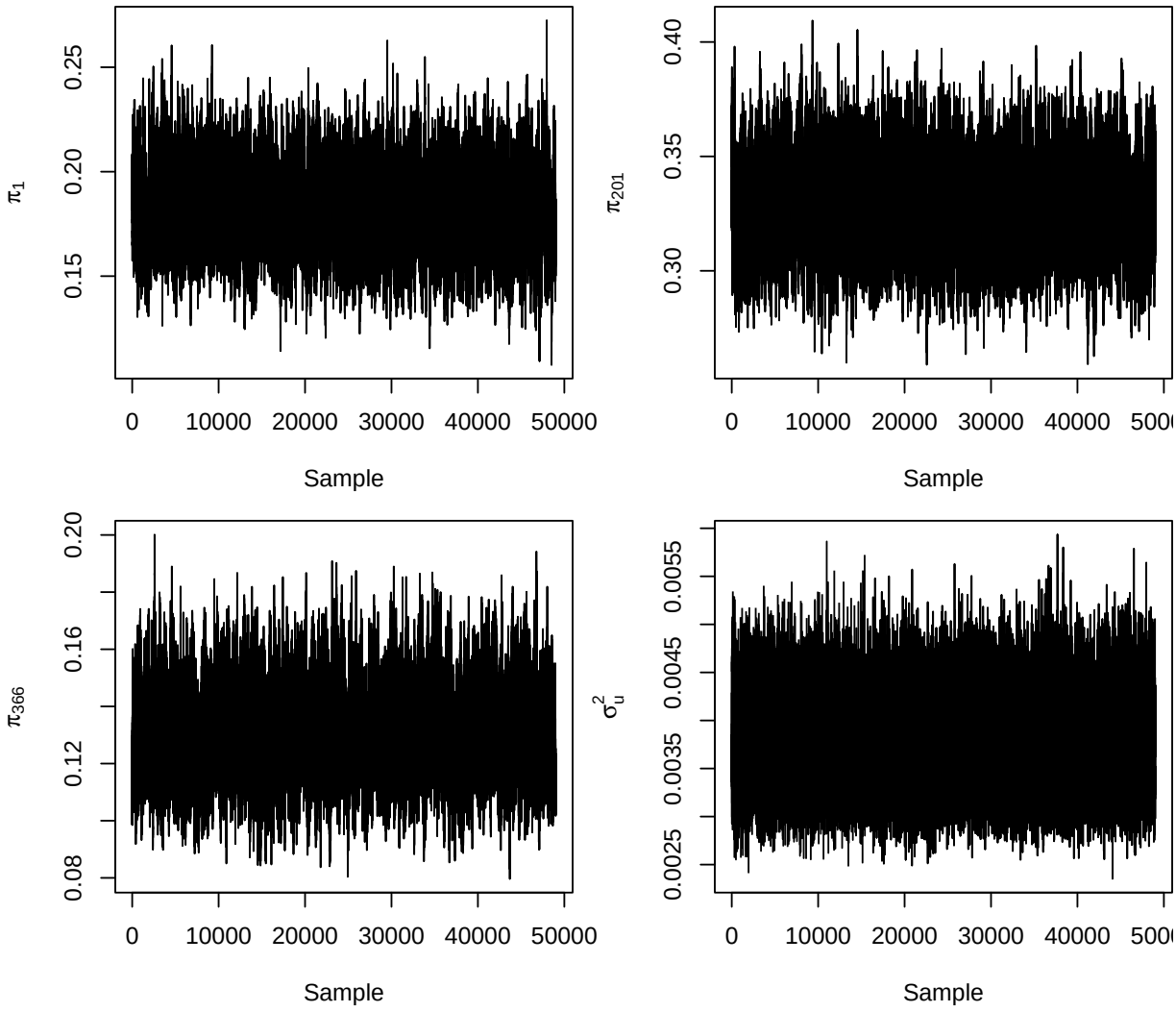


Figure 3: Trace plots of $\pi_1$, $\pi_{201}$, $\pi_3 66$ and $\sigma_u^2$ with 50 000 samples produced by a MCMC where every element of $\boldsymbol{\tau}$ is updated per iteration, but the first 1000 samples are discarded as burn-in samples.

```
set.seed(123)
# Plots a histogram of the vector along with a 95% credible interval
histogram_func <- function(vec, nint = 40, xlim = c(0, 1), xlab = NULL) {
    # Obtain the credible interval
    lower_upper = quantile(vec, probs = c(0.025, 0.975))
    # Plot histogram along with vertical lines for the credible interval and
    # the mean value of the vector
    hist(vec, col = "blue", probability = T, breaks = nint, xlab = xlab, xlim = xlim,
        main = NULL)
    abline(v = lower_upper[1], col = "red")
    abline(v = lower_upper[2], col = "red")
    abline(v = mean(vec), col = "green", lwd = 2)  #Mean
    legend("topright", legend = c("95% CI", "Mean", "Samples"), col = c("red", "green",
        "blue"), lty = 1)
    mean_val = mean(vec)
}
```

```
par(mfrow = c(2, 2), mar = c(5, 5, 0, 0))
histogram_func(sigma2, xlim = c(0.002, 0.006), nint = 4000, xlab = expression(sigma^2))
histogram_func(pi(tau[burn_period:n, 1]), xlim = c(0.1, 0.45), xlab = expression(pi[1]))
histogram_func(pi(tau[burn_period:n, 201]), xlim = c(0.1, 0.6), xlab = expression(pi[201]))
histogram_func(pi(tau[burn_period:n, 366]), xlim = c(0, 0.45), xlab = expression(pi[366]))
```

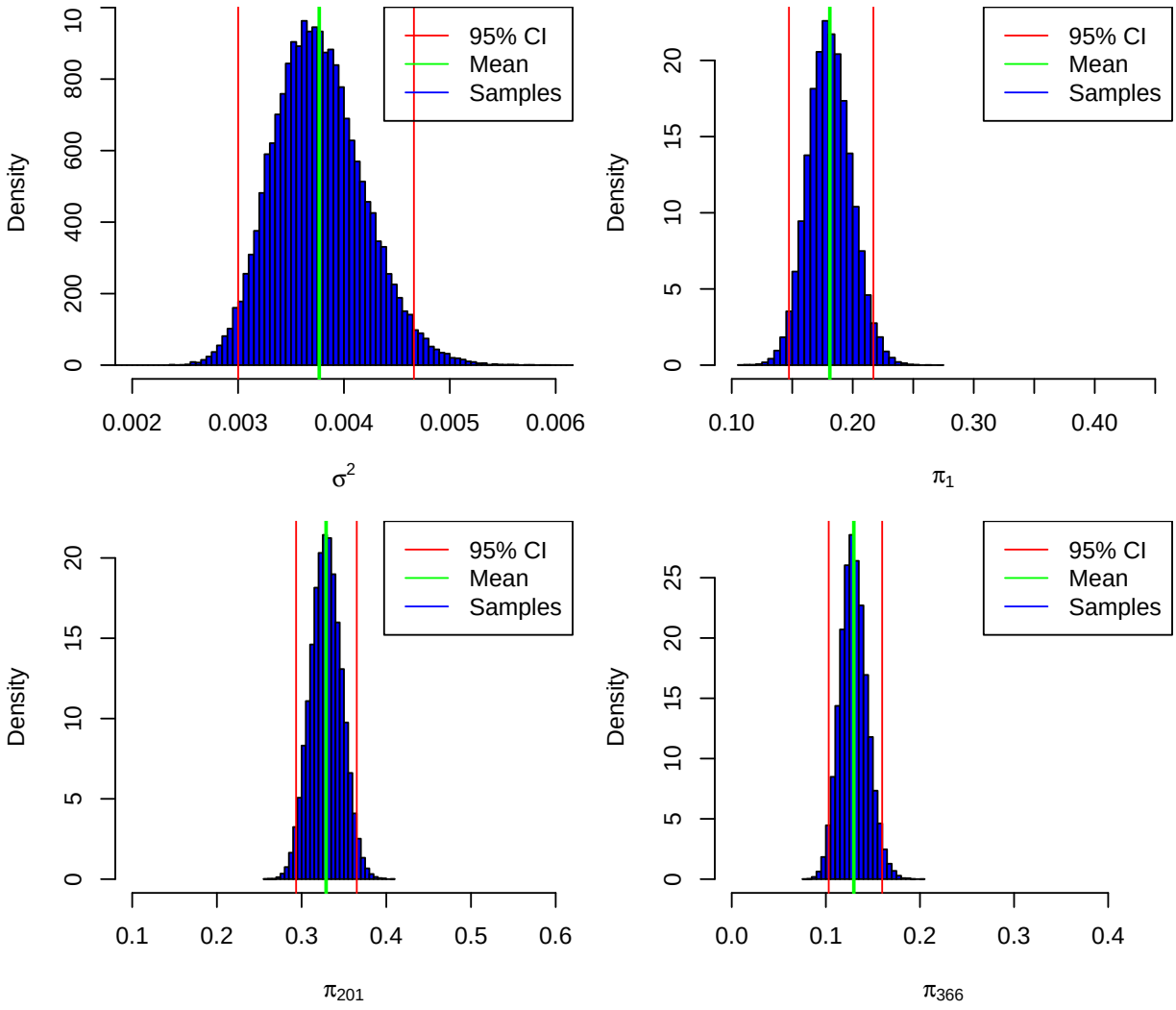


Figure 4: Histogram plots with 95% credible intervals and mean of $\pi_1$, $\pi_{201}$, $\pi_{366}$ and $\sigma_u^2$ with 50 000 samples produced by a MCMC where every element of $\boldsymbol{\tau}$ is updated per iteration, but the first 1000 samples are discarded as burn-in samples.

All four plots shows nicely distributed posterior distributions. The mean to the four distributions are given below:

```
set.seed(123)
cat("Mean of sigma:", round(mean(sigma2[burn_period:n]), 3), "\n")
```

```
## Mean of sigma: 0.004
```

```r
cat("Mean of pi_t=1:", round(mean(pi(tau[burn_period:n, 1])), 3), "\n")
```

```
## Mean of pi_t=1: 0.181
```

```r
cat("Mean of pi_t=201:", round(mean(pi(tau[burn_period:n, 201])), 3), "\n")
```

```
## Mean of pi_t=201: 0.329
```

```r
cat("Mean of pi_t=366:", round(mean(pi(tau[burn_period:n, 366])), 3), "\n")
```

```
## Mean of pi_t=366: 0.13
```

```r
set.seed(123)
# Plot autocorrelation function using acf from the stats package
par(mfrow = c(2, 2), mar = c(5, 5, 0, 0))
acf(sigma2[100:n], ylab = expression(paste("ACF ", sigma2["u"]^2)))
acf(pi(tau[, 1]), ylab = expression(paste("ACF ", pi[1])))
acf(pi(tau[, 201]), ylab = expression(paste("ACF ", pi[201])))
acf(pi(tau[, 366]), ylab = expression(paste("ACF ", pi[366])))
```

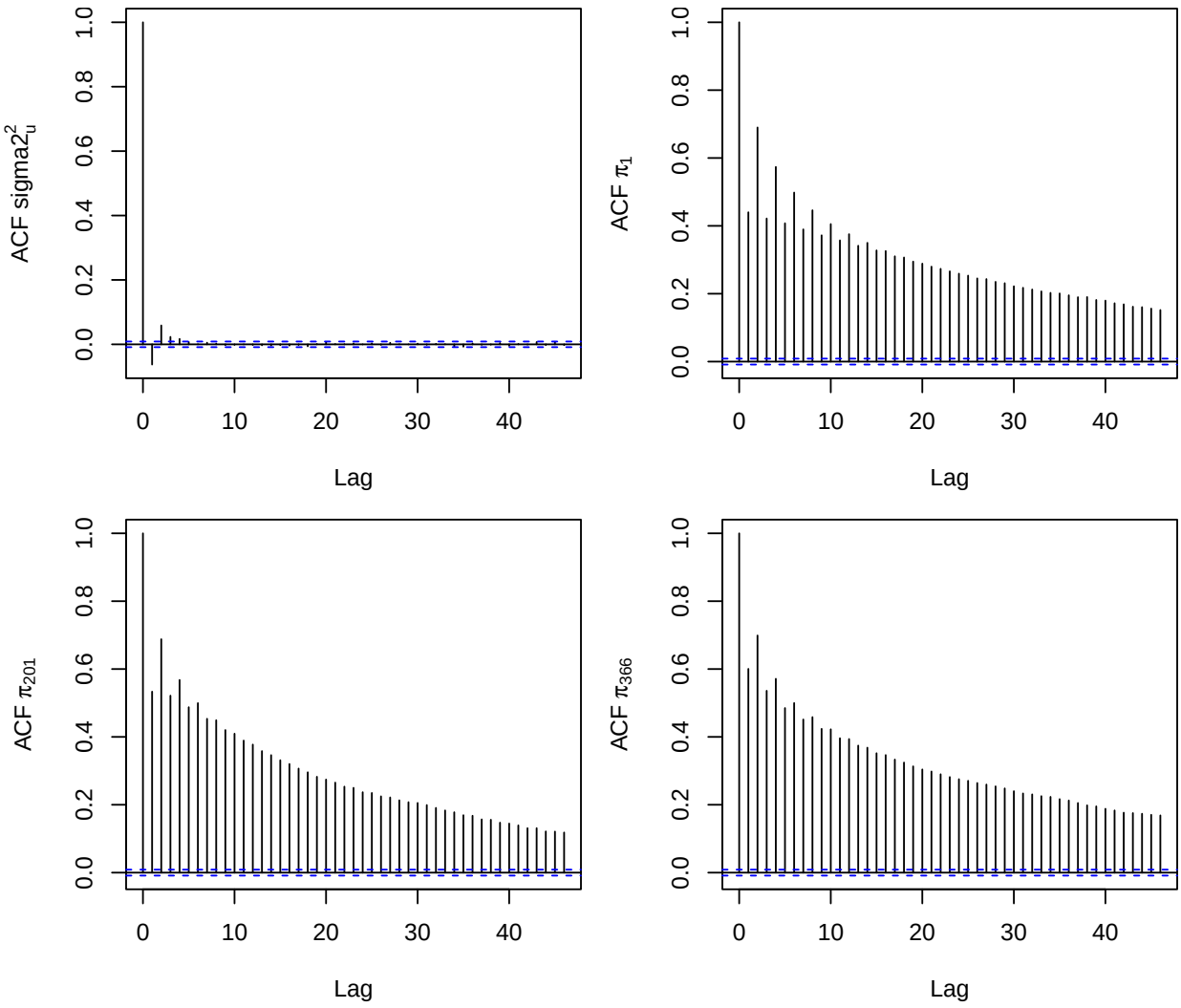


Figure 5: ACF plots for $\sigma_u^2$, $\pi_1$, $\pi_{201}$ and $\pi_{366}$ with 50 000 samples produced by a MCMC where every element of $\boldsymbol{\tau}$ is updated per iteration, but the first 1000 samples are discarded as burn-in samples.

The autocorrelation plots shows that there is no significant autocorrelation for $\sigma_u^2$, while for $\pi_1$, $\pi_{201}$ and $\pi_{366}$ there is significant correlation lags past 50 lags. This is because of the nature of a random walk being dependent upon the previous value.

```
set.seed(123)
tau_burn <- tau[burn_period:n, ]
sigma_burn <- sigma2[burn_period:n]

day = rain$day
n_rain = rain$n.rain
n_years = rain$n.years

compare_plot <- function(samp_matrix) {
    lower = c()
    upper = c()
    for (i in 1:366) {
```

```
        lower_upper = quantile(samp_matrix[, i], probs = c(0.025, 0.975))
        lower = append(lower, lower_upper[1])
        upper = append(upper, lower_upper[2])
    }
    plot(day, n_rain/n_years, cex = 0, pch = 1, ylab = expression(pi["t"]), xlab = "t",
        ylim = c(0, 1), type = "l", col = "gray", cex.lab = 1.5)
    lines(day, colMeans(samp_matrix), col = "green")
    lines(1:366, lower, col = "red")
    lines(1:366, upper, col = "red")
    legend("topright", legend = c("Observation model", "MCMC", "95% CI"), col = c("gray",
        "green", "red"), lty = 1)
}

compare_plot(pi(tau_burn))
```

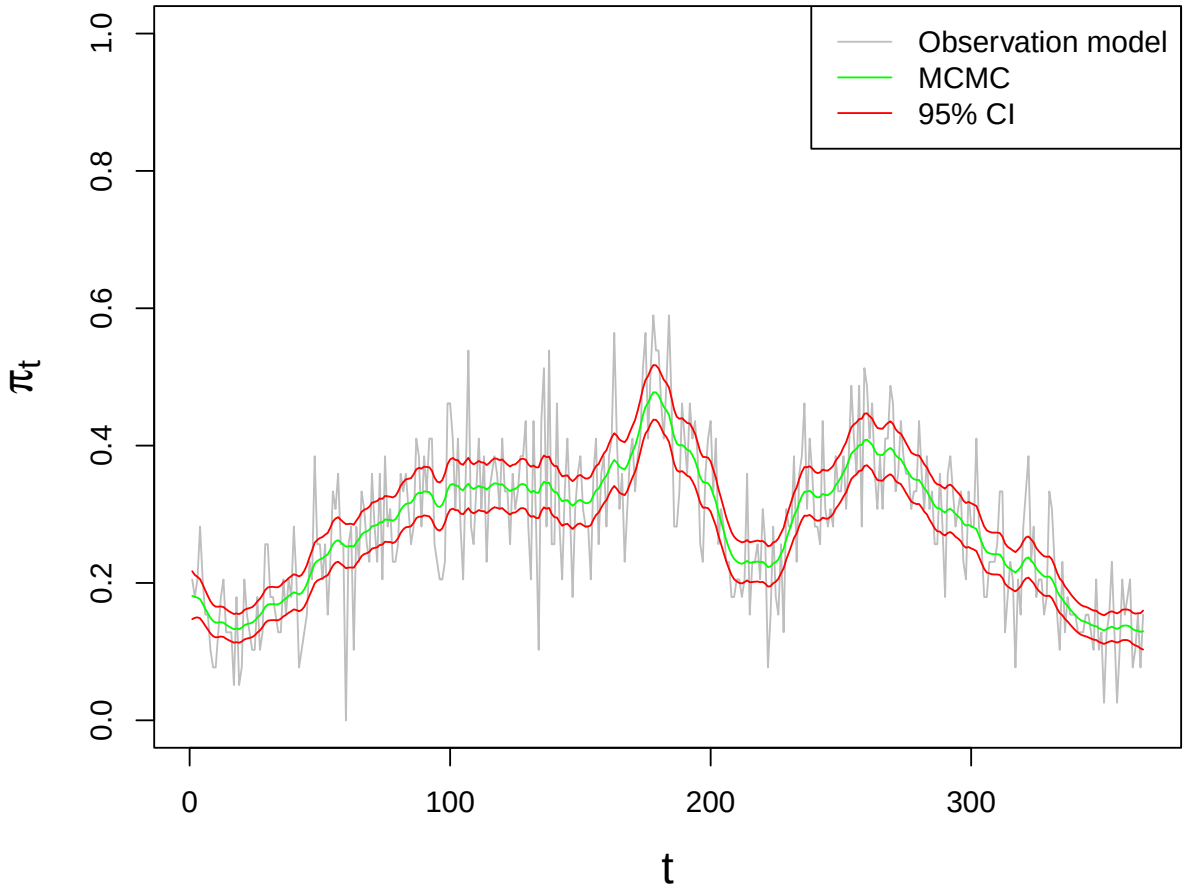


Figure 6: The figure shows the fraction of n.rain/n.years for each day $t$ in grey, and the green line is $\pi(\boldsymbol{\tau})$ from the MCMC with 95% credible interval in red.

The green line from the MCMC seems to be a smoothed line of the given data. This is due to the smoothing effect of a random walk with an order of 1. If the prediction where more spiked, and thereby follow the given data to larger extent, it would have been a sign of over-fitting.

```
set.seed(123)


ci_tab_single <- matrix(rep(0, 16), ncol = 4, byrow = TRUE)

# define column names and row names of matrix
colnames(ci_tab_single) <- c("sigma", "pi1", "pi201", "pi366")
rownames(ci_tab_single) <- c("Estimate", "CI low", "CI high", "Sd")

values_sigma = c(mean(sigma_burn), quantile(sigma_burn, probs = c(0.025, 0.975))[1],
    quantile(sigma_burn, probs = c(0.025, 0.975))[2], sd(sigma_burn))

values_pi1 = c(mean(pi(tau_burn[, 1])), quantile(pi(tau_burn[, 1]), probs = c(0.025,
    0.975))[1], quantile(pi(tau_burn[, 1]), probs = c(0.025, 0.975))[2], sd(pi(tau_burn[,
    1])))

values_pi201 = c(mean(pi(tau_burn[, 201])), quantile(pi(tau_burn[, 201]), probs = c(0.025,
    0.975))[1], quantile(pi(tau_burn[, 201]), probs = c(0.025, 0.975))[2], sd(pi(tau_burn[,
    201])))

values_pi366 = c(mean(pi(tau_burn[, 366])), quantile(pi(tau_burn[, 366]), probs = c(0.025,
    0.975))[1], quantile(pi(tau_burn[, 366]), probs = c(0.025, 0.975))[2], sd(pi(tau_burn[,
    366])))


ci_tab_single[, 1] = values_sigma
ci_tab_single[, 2] = values_pi1
ci_tab_single[, 3] = values_pi201
ci_tab_single[, 4] = values_pi366

# convert matrix to table
ci_tab_single <- as.table(ci_tab_single)

ci_tab_single
```

```
##                 sigma          pi1        pi201        pi366
## Estimate 0.0037621863 0.1810531520 0.3289337633 0.1297218418
## CI low   0.0030023835 0.1472934677 0.2935183749 0.1030400636
## CI high  0.0046609952 0.2170794677 0.3651352490 0.1598494425
## Sd       0.0004240812 0.0178124480 0.0183457217 0.0144085339
```

Table 1: The table gives the 95% credible interval values for $\pi_1$, $\pi_{201}$, $\pi_{366}$ and $\sigma_u^2$ with 50 000 samples produced by a MCMC where every element of $\boldsymbol{\tau}$ is updated per iteration, but the first 1000 samples are discarded as burn-in samples.

## Task 1f): Implementation of the block updating MCMC

Now, instead of updating each element of $\tau$ per iteration, we block together multiple values $\tau_{[a,b]}$ on an interval $[a, b]$ and update all elements in each block per iteration. This introduces a tuning parameter, namely the number of elements in each block, $M$. Intuitively, blocking elements together should speed up the process, but keep in mind that if the blocks grow to big it will require large matrix multiplications. We must also

investigate the effective sample size and the correlation of samples. First, we begin by deriving the theory for the code:

The sampling of $\sigma_u^2$, the Gibbs step, remains the same as before. As for the Metropolis Hastings step we no longer consider each element in $\tau$ but as stated previously, blocks. This means that our conditional prior proposal function now becomes

$$p(\tau_{[a,b]}|\tau_{-[a,b]}, \sigma_u^2)$$

so we will need to derive a new acceptance probability. Due to the conditional independence of $y_t|\tau_t$ for each $t \in [1, 366]$ and the result in $d$), the acceptance probability for blocks can be expressed as

$$\alpha(\tau'_{[a,b]}|\tau_{-[a,b]}, \sigma_u^2, y) = \min\left(1, \prod_{i=a}^{b} \frac{P(y|\tau_i')}{P(y|\tau_i)}\right) = \min\left(1, \exp\left(\sum_{i=a}^{b}(y_i(\tau_i' - \tau_i) + n_i \ln\left(\frac{1+e^{\tau_i}}{1+e^{\tau_i'}}\right))\right)\right)$$

We note that the acceptance probability now is a product of the probability for each element $\tau_i \in \tau_{[a,b]}$ so it is reasonable to assume that this acceptance probability will be inversely proportional to $M$. Let $\mathcal{I}$ denote the interval of interest, now we know that the conditional distribution of

$$\tau_{\mathcal{I}}|\tau_{-\mathcal{I}}$$

follows a multivariate normal distribution with the conditional mean and precision given by

$$\mu_{\mathcal{I}|-\mathcal{I}} = -Q_{\mathcal{I},\mathcal{I}}^{-1} Q_{\mathcal{I},-\mathcal{I}} \tau_{-\mathcal{I}} \qquad\qquad Q_{\mathcal{I}|-\mathcal{I}} = Q_{\mathcal{I},\mathcal{I}}$$

by the same reasoning as in $e$) before. We use the cholesky decomposition of $Q$ to draw the multivariate normal samples as $\tau' = \mu_{\mathcal{I}|-\mathcal{I}} + Lv$ where we obtain $L$ from the Cholesky decomposition of $Q_{\mathcal{I}|-\mathcal{I}}^{-1} = \Sigma_{\mathcal{I}|-\mathcal{I}} = LL^T$ and $v \sim \mathcal{N}(0, 1)$. In this case, with reasonable interval length $1 < M < 365$, we get three different precision matrices, one containing the first but not the last day, one without the first and the last day and lastly one containing the last but not the first. The implementation is found below

```
set.seed(123)
# Collect data
load(file = "rain.rda")
day = rain$day
n_rain = rain$n.rain
n_years = rain$n.years

# Define som functions to make life easier
expit <- function(tau) {
    return(1/(1 + exp(-tau)))
}


trace_plot <- function(plot_vector, ylab = expression(pi), ylim = c(0, 1)) {
    plot(1:length(plot_vector), plot_vector, ylim = ylim, cex = 0, ylab = ylab, type = "l",
        xlab = "sample", cex.lab = 1.5)
}


create_Q <- function(n, M) {
    # This works
    main = rep(2, n)
    lower = rep(-1, n - 1)
    upper = rep(-1, n - 1)
    main[1] = 1
    main[n] = 1
```

```
    return(tridiag(upper, lower, main))
}


accept_block <- function(interval, tau_prop, tau_int) {
    prob = exp(sum(n_rain[interval] * (tau_prop - tau_int)) + sum(n_years[interval] *
        log(((1 + exp(tau_int))/(1 + exp(tau_prop))))))
    return(min(1, prob))
}


# Implementation of the Metropolis Hastings block algorithm Default value for M
# added after analysis below
MCMC_block <- function(alpha, beta, n_samples, M = 10) {

    accepted = 0
    N = length(day)

    # For one iteration through all blocks last_block = ifelse(N%%M==0, M,
    # N%%M)
    iter = ceiling(N/M)

    # Precalculate all the matrices needed
    Q = create_Q(N)

    Q_AA_1 = Q[1:M, 1:M]
    Q_AB_1 = Q[1:M, -(1:M)]
    Q_AA_1_inv = solve(Q_AA_1)  #Inverse
    Q_AA_1_chol = t(chol(Q_AA_1_inv))  #Cholesky of Q_AA_inv
    Q_prod1 = -Q_AA_1_inv %*% Q_AB_1  #Product used for conditional mean

    Q_AA_2 = Q[2:(M + 1), 2:(M + 1)]
    # Q_AB_2 needs to be computed with the corresponding index inside the
    # innermost loop
    Q_AA_2_inv = solve(Q_AA_2)  #Inverse
    Q_AA_2_chol = t(chol(Q_AA_2_inv))  #Cholesky of Q_AA_inv

    # All matrices for the middle part will be calculated with the
    # corresponding index inside the loop over internal blocks

    Q_AA_3 = Q[(N + 1 - M):N, (N + 1 - M):N]
    Q_AB_3 = Q[(N + 1 - M):N, -((N + 1 - M):N)]
    Q_AA_3_inv = solve(Q_AA_3)  #Inverse
    Q_AA_3_chol = t(chol(Q_AA_3_inv))  #Cholesky of Q_AA_inv
    Q_prod3 = -Q_AA_3_inv %*% Q_AB_3  #Product used for conditional mean

    # Allocate memory for samples
    tau_mat = matrix(0, nrow = n_samples, ncol = N)
    tau_mat[1, ] = rnorm(N, sd = 0.5)

    # Sample sigma from the prior distribution
    sigma = rep(0, n_samples)
    sigma[1] = 1/rgamma(1, 2, rate = 0.05)
```

```
    # Iterate n samples
    for (i in 2:n_samples) {
        # First block
        a = 1
        b = M
        int = c(a:b)

        blocked = Metropolis_Block(int, tau_mat[i - 1, ], sigma[i - 1], Q_prod1,
            Q_AA_1_chol)
        tau_mat[i, int] = blocked$tau
        accepted = accepted + blocked$accepted

        for (j in 2:(iter - 1)) {
            # Second block
            a = (j - 1) * M + 1
            b = j * M
            int = c(a:b)

            Q_AB_2 = Q[int, -int]
            Q_prod2 = -Q_AA_2_inv %*% Q_AB_2  #Product used in the Metropolis step

            blocked = Metropolis_Block(int, tau_mat[i - 1, ], sigma[i - 1], Q_prod2,
                Q_AA_2_chol)
            tau_mat[i, int] = blocked$tau
            accepted = accepted + blocked$accepted

        }

        # Last block
        a = (N - M) + 1
        b = N
        int = c(a:b)

        blocked = Metropolis_Block(int, tau_mat[i - 1, ], sigma[i - 1], Q_prod3,
            Q_AA_3_chol)
        tau_mat[i, int] = blocked$tau
        accepted = accepted + blocked$accepted

        # Function diff() gives successive differences of array
        tau_Q_tau = sum(diff((tau_mat[i, ]))^2)
        # sigma[i] = (1/rgamma(1, alpha + (N-1)/2, rate = (beta +
        # (1/2)*tau_Q_tau)))

        # sigma[i] = rinvgamma(1, shape=(alpha+ (N-1/2)), scale = (beta +
        # (1/2)*tau_Q_tau))
        sigma[i] = 1/(rgamma(1, shape = (alpha + (N - 1)/2), rate = (beta + (1/2) *
            tau_Q_tau)))

    }
    return(list(pi = expit(tau_mat), tau = tau_mat, sigma = sigma, acceptance_ratio = accepted/(N *
        n_samples)))
}
```

```
# The Metropolis step
Metropolis_Block <- function(int, prev_tau, sigma, Q_prod, Q_chol) {

    # Make the proposed tau values from a conditional multivariate normal
    # distribution
    cond_mu = Q_prod %*% prev_tau[-int]
    tau_prop = cond_mu + sqrt(sigma) * Q_chol %*% rnorm(length(prev_tau[int]))

    # Accept/reject
    if (runif(1) < accept_block(int, tau_prop, prev_tau[int])) {
        return(list(tau = tau_prop, accepted = length(int), mu = cond_mu))
    } else {
        return(list(tau = prev_tau[int], accepted = 0, mu = cond_mu))
    }
}
```

A note on the implementation is that even though it is suggested that we precompute all the precision matrices, we found this to be difficult. Since the matrix $Q_{AB}$ will differ based on the index in the innermost for loop, we chose to implement it inside. Our attempts to not to it this way proved to give unreasonable results, so we chose quality over saving a little bit of computational stress.

Going forward we will explore the properties of the tuning parameter $M$ and then do the same analysis as before for a suitable value of $M$. We use the set $\{1, 5, 10, 15, 20, 25, 30, 35, 42, 50, 60, 70, 80, 100\}$ as our possible $M$ values.

```
set.seed(123)
# Multiple M values and plot some properties as functions of the different
# values

M_arr = c(1, 5, 10, 15, 20, 25, 30, 35, 42, 50, 60, 70, 80, 100)

runtime_arr = c()
accept_arr = c()
mean_samp = c()
min_samp = c()

n_samples_2 = 5000
alpha = 2
beta = 0.05

for (i in 1:length(M_arr)) {
    start_curr = proc.time()[3]
    current_test = MCMC_block(alpha, beta, n_samples_2, M_arr[i])
    runtime_arr = append(runtime_arr, (proc.time()[3] - start_curr))
    accept_arr = append(accept_arr, current_test$acceptance_ratio)
    eff_size = effectiveSize(current_test$pi[1000:n_samples_2, ])
    mean_samp = append(mean_samp, mean(eff_size))
    min_samp = append(min_samp, min(eff_size))
}

par(mfrow = c(2, 2), mar = c(5, 5, 0, 0))
plot(M_arr, runtime_arr, type = "b", ylab = "Runtime (s)", xlab = "M")
abline(v = 10, col = "red")
plot(M_arr, accept_arr, type = "b", ylab = "Acceptance rate", xlab = "M")
abline(v = 10, col = "red")
```

```
plot(M_arr, mean_samp/runtime_arr, type = "b", ylab = "Mean ESS/second", xlab = "M")
abline(v = 10, col = "red")
plot(M_arr, min_samp/runtime_arr, type = "b", ylab = "Min ESS/second", xlab = "M")
abline(v = 10, col = "red")
```

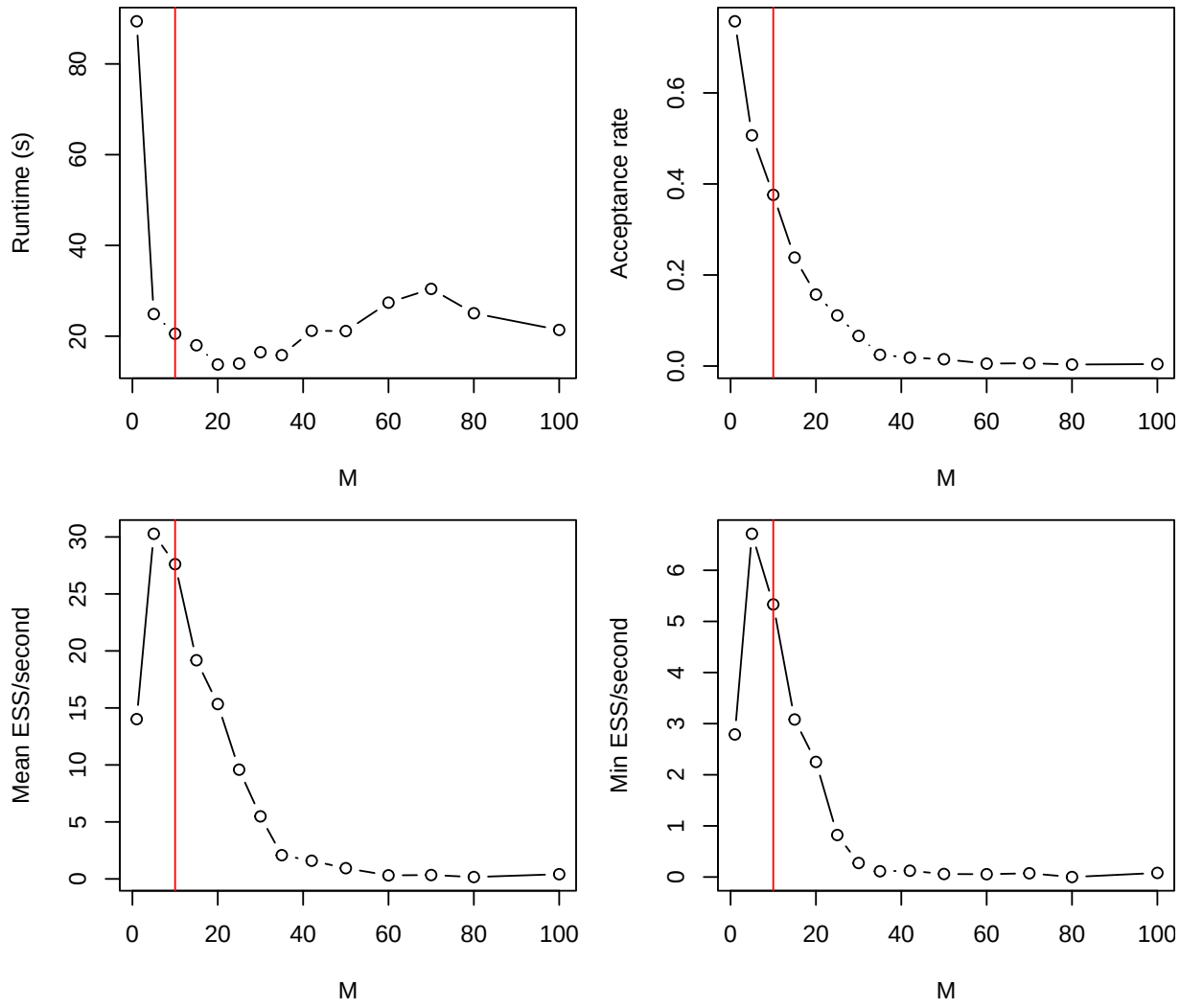


Figure 7: Some properties of the MCMC blocking algorithm as a function of block size M, along with the properties when M=10 which we find to be a suitable value for M based on these properties. The number of samples are reduced to 5000 due to runtime issues.

In figure 7 we see that the runtime decreases as the block sizes increases to around 25 before it increases again. This is reasonable as to begin with we reduce the number of iterations by blocking the $\tau_t$ parameters, but after a while it is likely that the large matrices makes for a longer computation time and the effect of blocking is lost. The acceptance rates seem to be steadily going downwards as we expected. We choose to view the mean and minimum ESS per second instead of only the mean and minimum ESS, as the definition of ESS does not take computational stress into account. From the plots we see that the optimal value is $M = 5$ or $M = 10$. The increase from $M = 1$ to $M = 5$ is likely due to the major decrease in runtime, while the decrease after $M = 5$ is likely due to the drop in acceptance rate increasing the correlation of our samples. With all these considerations in mind, $M = 10$ seems a good value for block size and it is with this value we

will compare the single site and the blocked MCMC algorithm.

```
# Collect results from suitable M=10
set.seed(123)
n_samples = 50000
alpha = 2
beta = 0.05

before_MCMC_block = proc.time()[3]
test_results <- MCMC_block(alpha, beta, n_samples)
after_MCMC_block = proc.time()[3]
MCMC_block_runtime = after_MCMC_block - before_MCMC_block

cat("Running time:", MCMC_block_runtime, "seconds \n")
```

```
## Running time: 205.475 seconds
```

We see that we have an increase in runtime from task $1e)$ to $1f)$. This is likely because we must for each block create the matrices $Q$ according to the index we are at, which we must not in task $1e)$. This problem lies in the implementation of our blocking algorithm and we have seen in the above discussion that if $M = 1$ the blocking algorithm uses far more time than for $M = 10$ so we still conclude that the blocking reduces runtime if all matrices are precomputed.

```
set.seed(123)

# Traceplot, ACF and histograms of the samples obtained from an M value that
# gives nice properties

sigma_samples = test_results$sigma
pi_samples = test_results$pi


# Trace plots
par(mfrow = c(2, 2), mar = c(5, 5, 0, 0))
trace_plot(sigma_samples, ylab = expression(sigma["u"]^2))
trace_plot(pi_samples[, 1], ylab = expression(pi[1]))
trace_plot(pi_samples[, 201], ylab = expression(pi[201]))
trace_plot(pi_samples[, 366], ylab = expression(pi[366]))
```

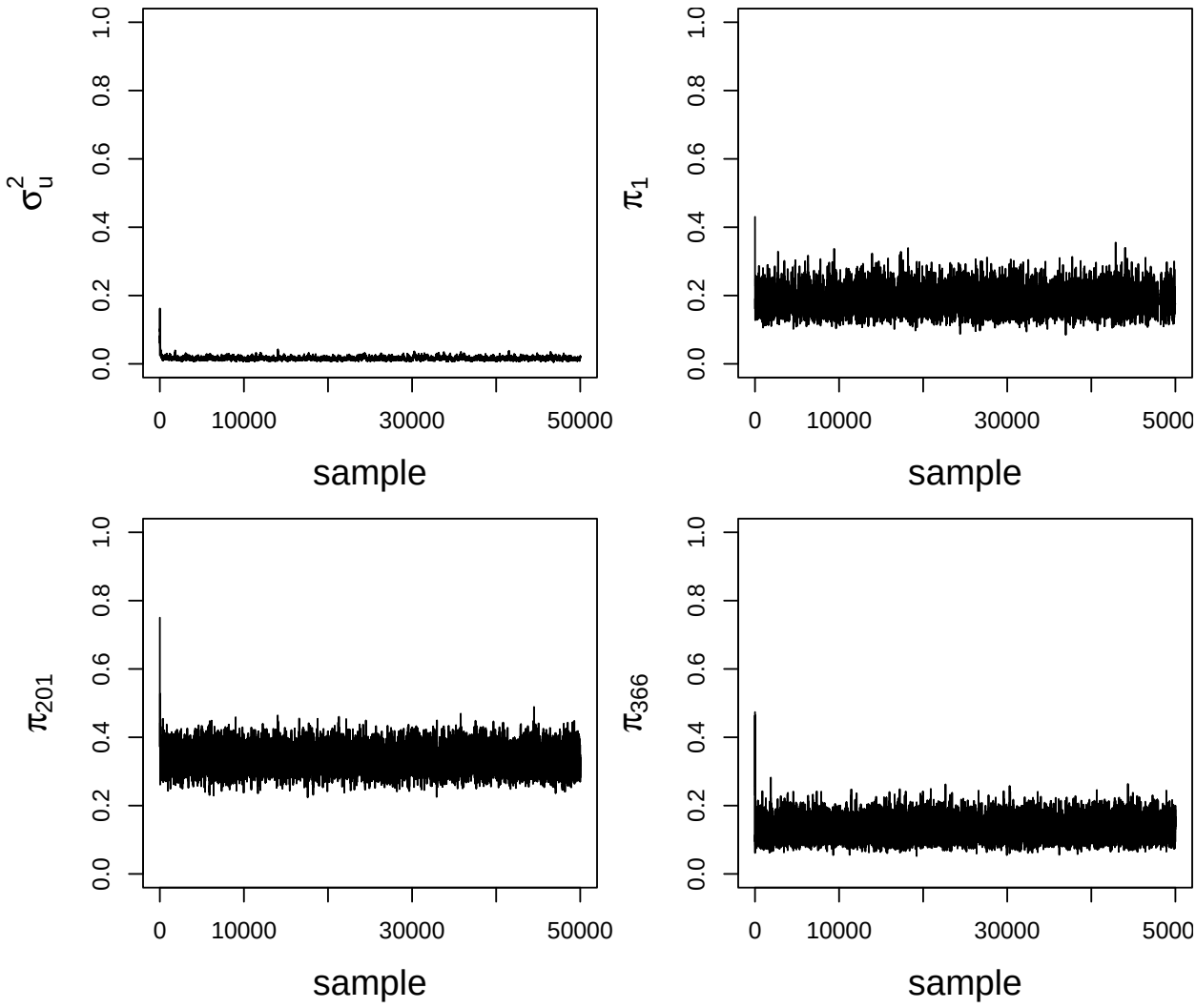


Figure 8: Trace plots of $\pi_1$, $\pi_{201}$, $\pi_3 66$ and $\sigma_u^2$ with 50 000 samples produced by a MCMC where elements of $\boldsymbol{\tau}$ are updated blockwise with block size M=10 per iteration

```
# Discard some burn in samples
samples_block = test_results$pi[1000:n_samples, ]
```

The trace plots seem to stabilize quickly, so we continue with burning the first 1000 samples and do the analysis with them.

```
sigma_samples_burn = test_results$sigma[1000:n_samples]
pi_samples_burn = test_results$pi[1000:n_samples, ]


par(mfrow = c(2, 2), mar = c(5, 5, 0, 0))
histogram_func(sigma_samples_burn, xlim = c(0, 0.05), xlab = expression(sigma^2))
histogram_func(pi_samples_burn[, 1], xlim = c(0, 0.45), xlab = expression(pi[1]))
histogram_func(pi_samples_burn[, 201], xlim = c(0.1, 0.6), xlab = expression(pi[201]))
histogram_func(pi_samples_burn[, 366], xlim = c(0, 0.45), xlab = expression(pi[366]))
```

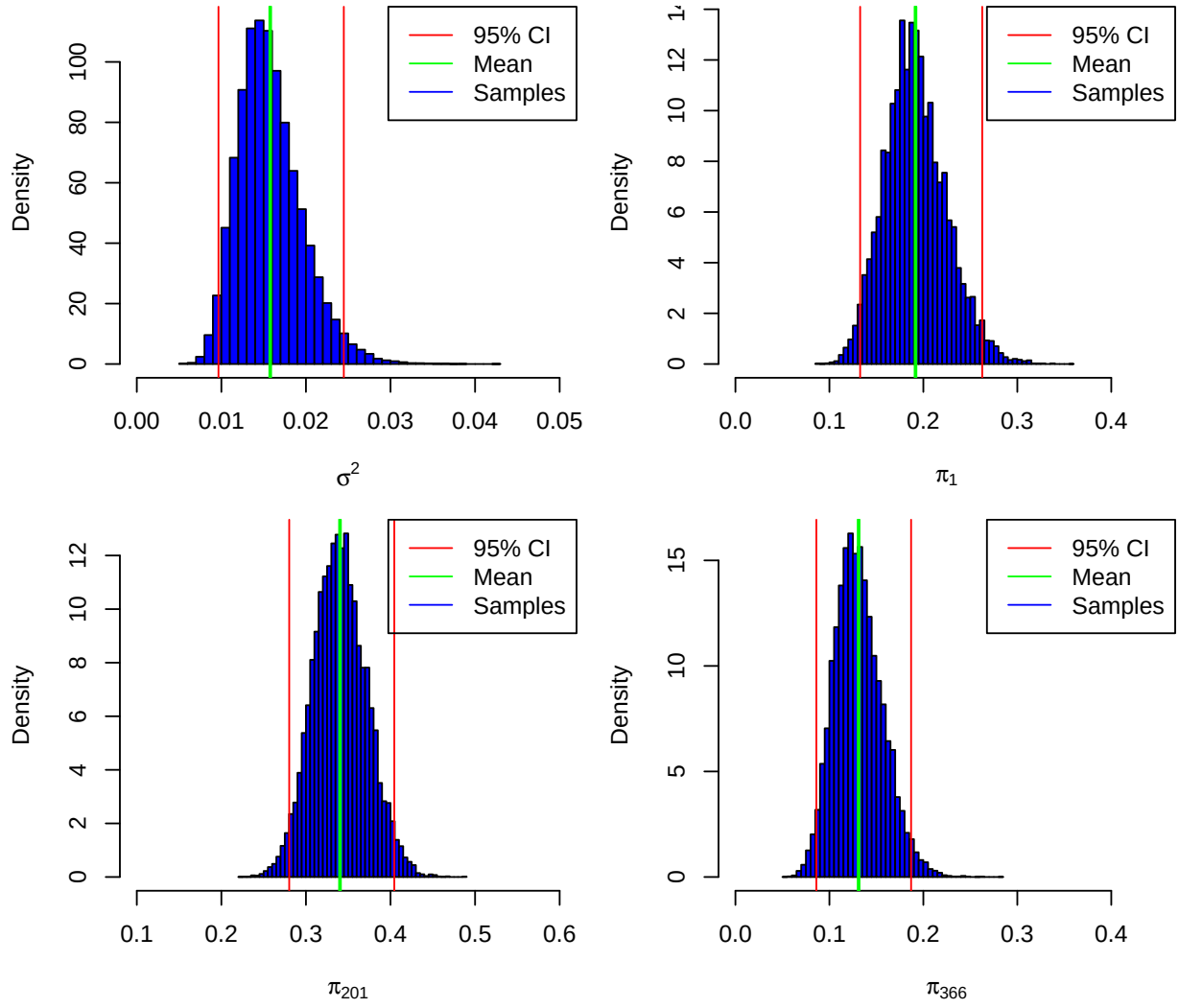


Figure 9: Histogram plots with 95% credible intervals and mean of $\pi_1$, $\pi_{201}$, $\pi_{366}$ and $\sigma_u^2$ with 50 000 samples produced by a MCMC where elements of $\boldsymbol{\tau}$ are updated blockwise with block size M=10 per iteration, but the first 1000 samples are discarded as burn-in samples.

```
mean_sigma = mean(sigma_samples_burn)
mean_pi1 = mean(pi_samples_burn[, 1])
mean_pi201 = mean(pi_samples_burn[, 201])
mean_pi366 = mean(pi_samples_burn[, 366])

cat("Mean of sigma:", round(mean_sigma, 3), "\n")
```

```
## Mean of sigma: 0.016
```

```
cat("Mean of pi_t=1:", round(mean_pi1, 3), "\n")
```

```
## Mean of pi_t=1: 0.192
```

```
cat("Mean of pi_t=201:", round(mean_pi201, 3), "\n")
```

```
## Mean of pi_t=201: 0.34
```

```r
cat("Mean of pi_t=366:", round(mean_pi366, 3), "\n")
```

```
## Mean of pi_t=366: 0.131
```

The samples now appear to be very similar to that of the single site algorithm. The means do not differ greatly and the histograms show nice distributions. They could perhaps have been a bit smoother so maybe one should ideally sample more than 50000 samples.

```r
# Autocorrelation
par(mfrow = c(2, 2), mar = c(5, 5, 0, 0))
acf(sigma_samples_burn, ylab = expression(paste("ACF for ", sigma["u"]^2)))
acf(samples_block[1, ], ylab = expression(paste("ACF for ", pi[1])), lag.max = 40)
acf(samples_block[201, ], ylab = expression(paste("ACF for ", pi[201])), lag.max = 40)
acf(samples_block[366, ], ylab = expression(paste("ACF for ", pi[366])), lag.max = 40)
```

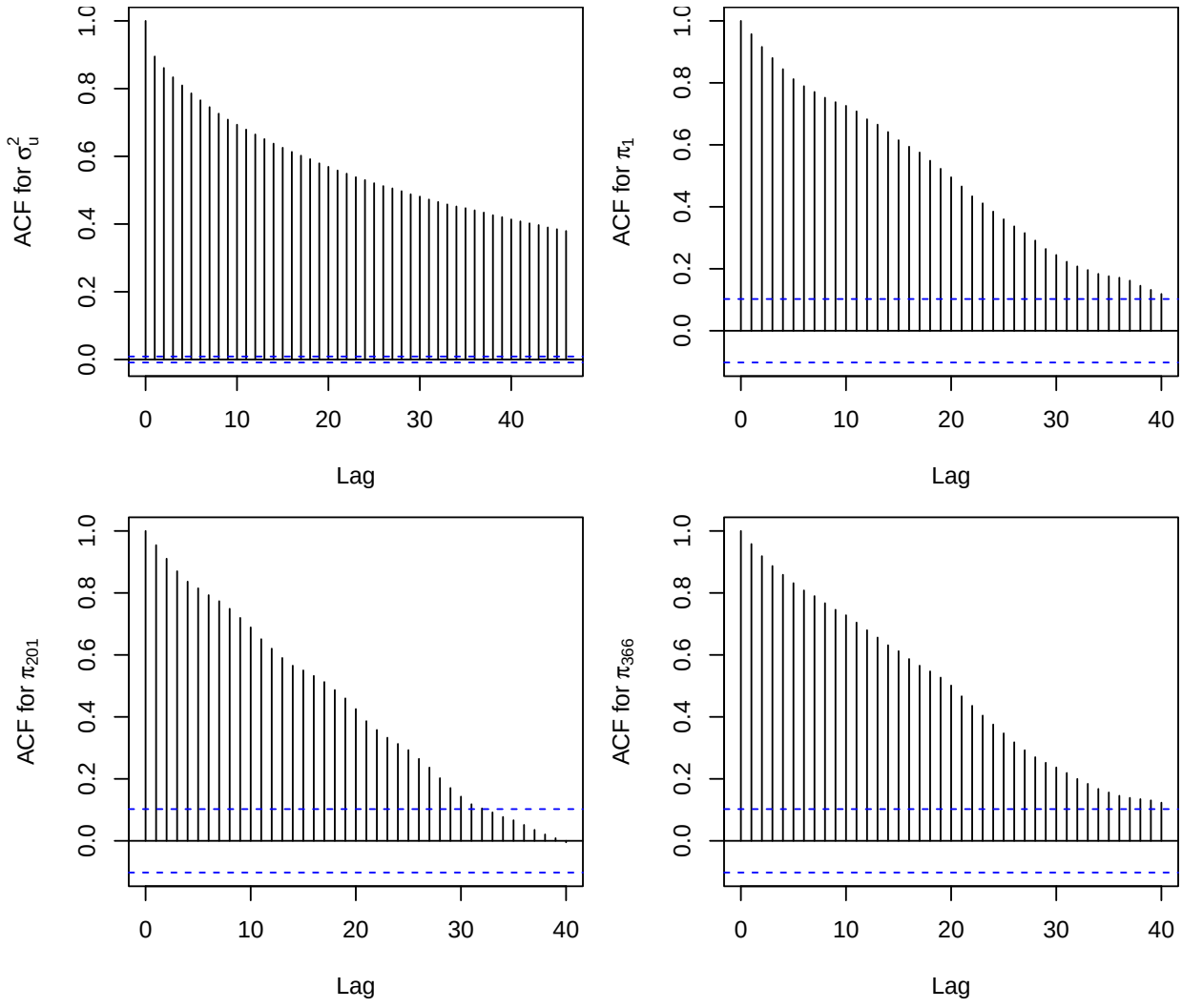


Figure 10: ACF plots for $\sigma_u^2$, $\pi_1$, $\pi_{201}$ and $\pi_{366}$ with 50 000 samples produced by a MCMC where elements of $\boldsymbol{\tau}$ are updated blockwise with block size M=10 per iteration, but the first 1000 samples are discarded as burn-in samples.

For the ACF plots, the correlation of the $\sigma_u^2$'s are higher than for the single update, but the others seem to

drop to zero a bit faster than in the single update, especially the one for $\pi_{201}$. This is also expected, as the correlation in lags should be lower when blocking is introduced.

```
set.seed(123)
# Comparison with data values


par(mfrow = c(1, 1), mar = c(5, 5, 0, 0))
compare_plot(samples_block)
```

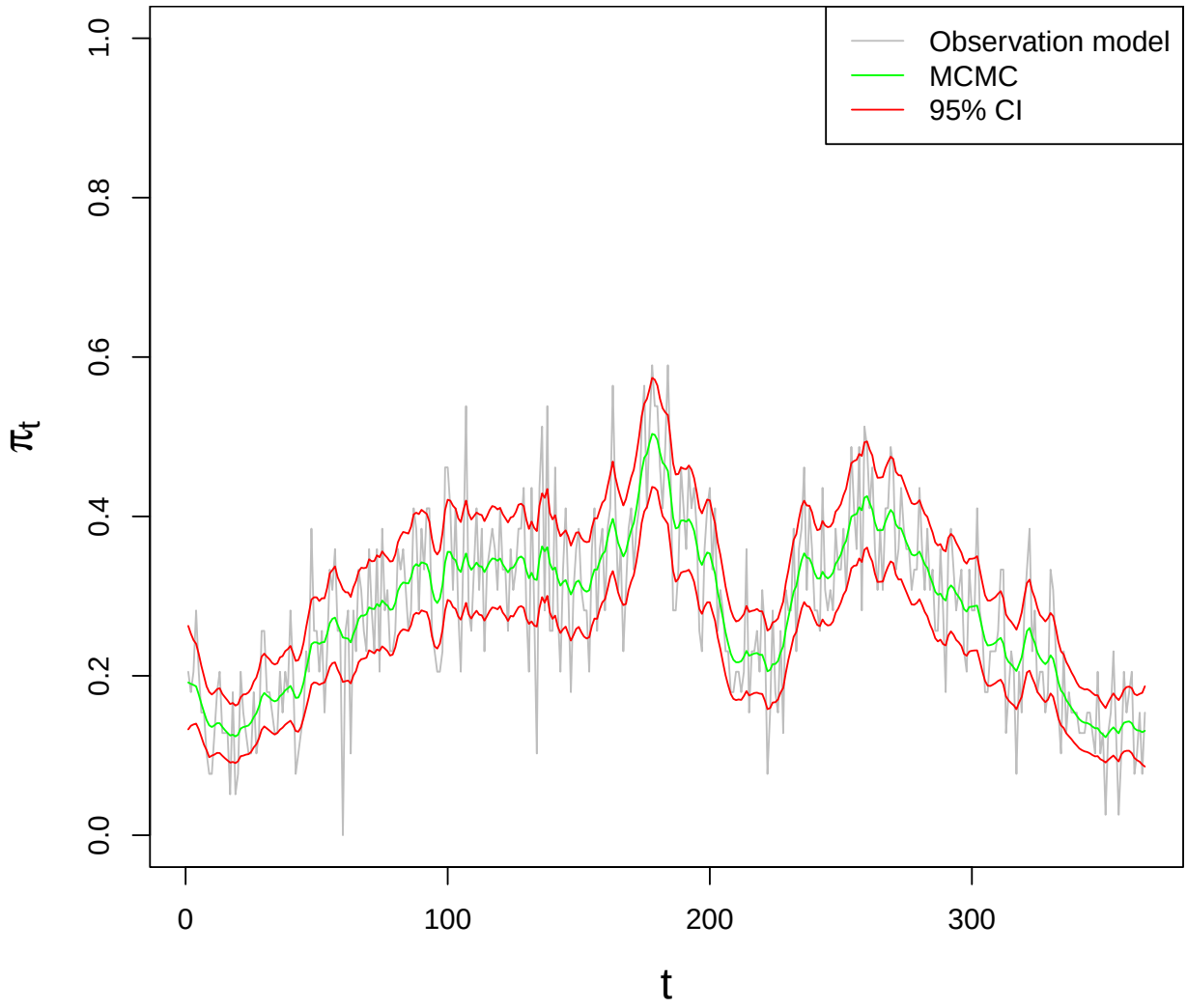


Figure 11: The figure shows the fraction of n.rain/n.years for each day $t$ in gray, and the green line is $\pi(\boldsymbol{\tau})$ from the blocked MCMC with M=10 and a 95% credible interval in red.

Judging by the comparison of the blocked MCMC to the observed data and the above analysis, it seems our Markov chain has converged, the predictions are plausible and the confidence intervals seem fairly tight. The method aligns well with the observed data and we have reason to believe that the implementation has been successful based on this and the previous analysis.

```
set.seed(123)
```

```r
ci_tab_block <- matrix(rep(0, 16), ncol = 4, byrow = TRUE)

colnames(ci_tab_block) <- c("sigma", "pi1", "pi201", "pi366")
rownames(ci_tab_block) <- c("Estimate", "CI low", "CI high", "Sd")

values_sigma_b = c(mean(sigma_samples_burn), quantile(sigma_samples_burn, probs = c(0.025,
    0.975))[1], quantile(sigma_samples_burn, probs = c(0.025, 0.975))[2], sd(sigma_samples_burn))

values_pi1_b = c(mean(pi_samples_burn[, 1]), quantile(pi_samples_burn[, 1], probs = c(0.025,
    0.975))[1], quantile(pi_samples_burn[, 1], probs = c(0.025, 0.975))[2], sd(pi_samples_burn[,
    1]))

values_pi201_b = c(mean(pi_samples_burn[, 201]), quantile(pi_samples_burn[, 201],
    probs = c(0.025, 0.975))[1], quantile(pi_samples_burn[, 201], probs = c(0.025,
    0.975))[2], sd(pi_samples_burn[, 201]))

values_pi366_b = c(mean(pi_samples_burn[, 366]), quantile(pi_samples_burn[, 366],
    probs = c(0.025, 0.975))[1], quantile(pi_samples_burn[, 366], probs = c(0.025,
    0.975))[2], sd(pi_samples_burn[, 366]))


ci_tab_block[, 1] = values_sigma_b
ci_tab_block[, 2] = values_pi1_b
ci_tab_block[, 3] = values_pi201_b
ci_tab_block[, 4] = values_pi366_b

ci_tab_block <- as.table(ci_tab_block)

ci_tab_block
```

```
##                sigma         pi1       pi201       pi366
## Estimate 0.015779303 0.191644240 0.340260124 0.131100170
## CI low   0.009657273 0.132731125 0.280215317 0.086131327
## CI high  0.024476862 0.262651275 0.404477024 0.186964410
## Sd       0.003805159 0.032883737 0.031661555 0.025662465
```

Table 2: The table gives the 95% credible interval values for $\pi_1$, $\pi_{201}$, $\pi_{366}$ and $\sigma_u^2$ with 50 000 samples produced by a MCMC where elements of $\boldsymbol{\tau}$ are updated blockwise with blocksize $M = 10$ per iteration, but the first 1000 samples are discarded as burn-in samples.

```r
set.seed(123)
# Comparison of single site and block

tau.df <- data.frame(t(tau_burn))
tau.df$mean <- rowMeans(tau.df[, c(1:ncol(tau.df) - 1)], na.rm = TRUE)

plot(day, n_rain/n_years, cex = 0, pch = 1, ylab = expression(pi["t"]), xlab = "t",
    ylim = c(0, 0.7), type = "l", col = "gray", cex.lab = 1.5)
lines(1:366, pi(tau.df$mean), type = "l", col = "red", xlab = "t", ylab = expression(pi["t"]))
lines(1:366, colMeans(samples_block), type = "l", col = "green")
legend("topright", col = c("red", "green"), legend = c("Single Site", "Blocking"),
    lty = 1)
```

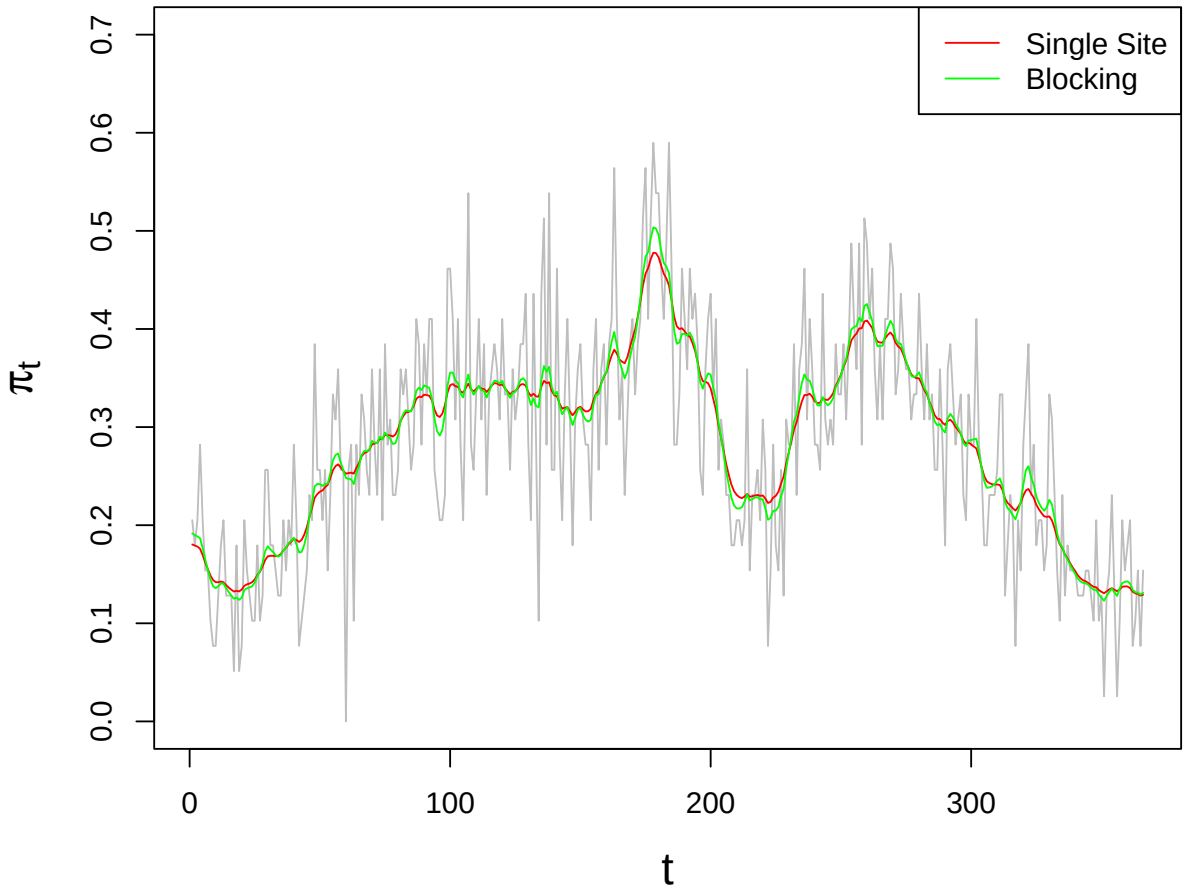


Figure 12: The figure compares the single site MCMC means of the fraction of n.rain/n.years for each day $t$, to the means of the blocked MCMC method with M=10

```
mean_ESS_single = mean(effectiveSize(pi(tau[burn_period:n, ])))
mean_ESS_block = mean(effectiveSize(samples_block))

cat("Mean ESS/second for the single update algorithm:", mean_ESS_single/runtime_single,
    "\n")
```

```
## Mean ESS/second for the single update algorithm: 10.43783
```

```
cat("Mean ESS/second for the block update algorithm:", mean_ESS_block/MCMC_block_runtime,
    "\n")
```

```
## Mean ESS/second for the block update algorithm: 28.75034
```

```
cat("The single update algorithm is ", abs(runtime_single - MCMC_block_runtime),
    " seconds faster than blocking algorithm \n")
```

```
## The single update algorithm is  61.589  seconds faster than blocking algorithm
```

When comparing the means of the single site versus the blocked means, the blocked means are more jagged and it seems the blocked algorithm has a sort of higher amplitude. The blocked algorithm seems to give both higher and lower $\pi_t$ values based on the surrounding trends. It is hard to say whether or not this is always desirable and if it suggests overfitting or simply a better model, though the more correlated $\sigma_u^2$ values might cause the random walk to lose a bit of smoothing effect, thus causing the more jagged line. It does suggest that blocking causes the $\pi_t$ values to hug the observed values tighter, so we will not discard it as a negative property in this setting. It is also interesting that even though the blocking algorithm runs slightly slower than the single update algorithm, it produces has an ESS per second which is roughly 3 times larger than that of the single update algorithm for the $\pi(\tau_t)$ samples. This is a major advantage of the blocking algorithm and indicates that it is both more efficient and/or less correlated.

# Task 2: INLA implementation, analysis and comparison to MCMC on the Tokyo rainfall dataset

We know wish to use the INLA method to do the same calculations as before. This implementation is far easier in R and we expect it to yield faster results.

```
set.seed(123)
# install.packages('INLA',repos=c(getOption('repos'),
# INLA='https://inla.r-inla-download.org/R/stable'), dep=TRUE)
library("INLA")
```

```
set.seed(123)
plot_INLA <- function(inla_mod) {
    # Function to plot the INLA model in a similar way as we plotted the MCMC
    # model
    plot(day, n_rain/n_years, cex = 0, pch = 1, ylab = expression(pi["t"]), xlab = "t",
        ylim = c(0, 1), type = "l", col = "gray", cex.lab = 1.5)
    lines(inla_mod$summary.fitted.values$mean, type = "l", col = "green", ylim = c(0,
        1), xlab = "Day", ylab = expression(pi["t"]))
    lines(inla_mod$summary.fitted.values$"0.025quant", col = "red")
    lines(inla_mod$summary.fitted.values$"0.975quant", col = "red")
    legend("topright", legend = c("Predictions", "95% Credible Interval"), col = c("green",
        "red"), lty = 1)
}
```

## Task 2a): Comparison with MCMC

Before we fit any models, we must make sure that we fit the same model as we did for the MCMC implementation. To do this we must note that INLA places a prior on the logarithmic precision instead of on the variance. In our case this leads to

$$\log(\rho) = \log\left(\frac{1}{\sigma_u^2}\right) \sim \log\left(\frac{1}{\text{InvGamma}(\text{shape} = 2, \text{scale} = 0.05)}\right) = \log\left(\text{Gamma}(\text{shape} = 2, \text{rate} = 0.05)\right)$$

The implementation of the first INLA model and a comparison to the MCMC model is found below

```
set.seed(123)
# Prior parameters
alpha = 2
beta = 0.05
```

```r
# Fit the first INLA model
before = proc.time()[3]
control.inla = list(strategy = "simplified.laplace", int.strategy = "ccd")
mod_1 <- inla(n.rain ~ -1 + f(day, model = "rw1", constr = FALSE, hyper = list(prec = list(prior = "log
    param = c(alpha, beta)))), data = rain, Ntrials = n.years, control.compute = list(config = TRUE),
    family = "binomial", verbose = TRUE, control.inla = control.inla)

after = proc.time()[3]
runtime_INLA_1 = after[[1]] - before[[1]]

compare_INLA <- function(samp_matrix, inla_mod) {
    # Compares INLA to MCMC
    lower = c()
    upper = c()
    for (i in 1:366) {
        lower_upper = quantile(samp_matrix[, i], probs = c(0.025, 0.975))
        lower = append(lower, lower_upper[1])
        upper = append(upper, lower_upper[2])
    }
    plot(day, n_rain/n_years, cex = 0, pch = 1, ylab = expression(pi["t"]), xlab = "t",
        ylim = c(0, 1), type = "l", col = "gray", cex.lab = 1.5)
    lines(day, colMeans(samp_matrix), col = "green")
    lines(1:366, lower, col = "red")
    lines(1:366, upper, col = "red")
    lines(inla_mod$summary.fitted.values$mean, type = "l", col = "yellow", ylim = c(0,
        1), xlab = "Day", ylab = expression(pi["t"]))
    lines(inla_mod$summary.fitted.values$"0.025quant", col = "blue")
    lines(inla_mod$summary.fitted.values$"0.975quant", col = "blue")
    legend("topright", legend = c("Observation model", "MCMC", "INLA", "95% CI MCMC",
        "95% CI INLA"), col = c("gray", "green", "yellow", "red", "blue"), lty = 1)
}

compare_INLA(samples_block, mod_1)
```

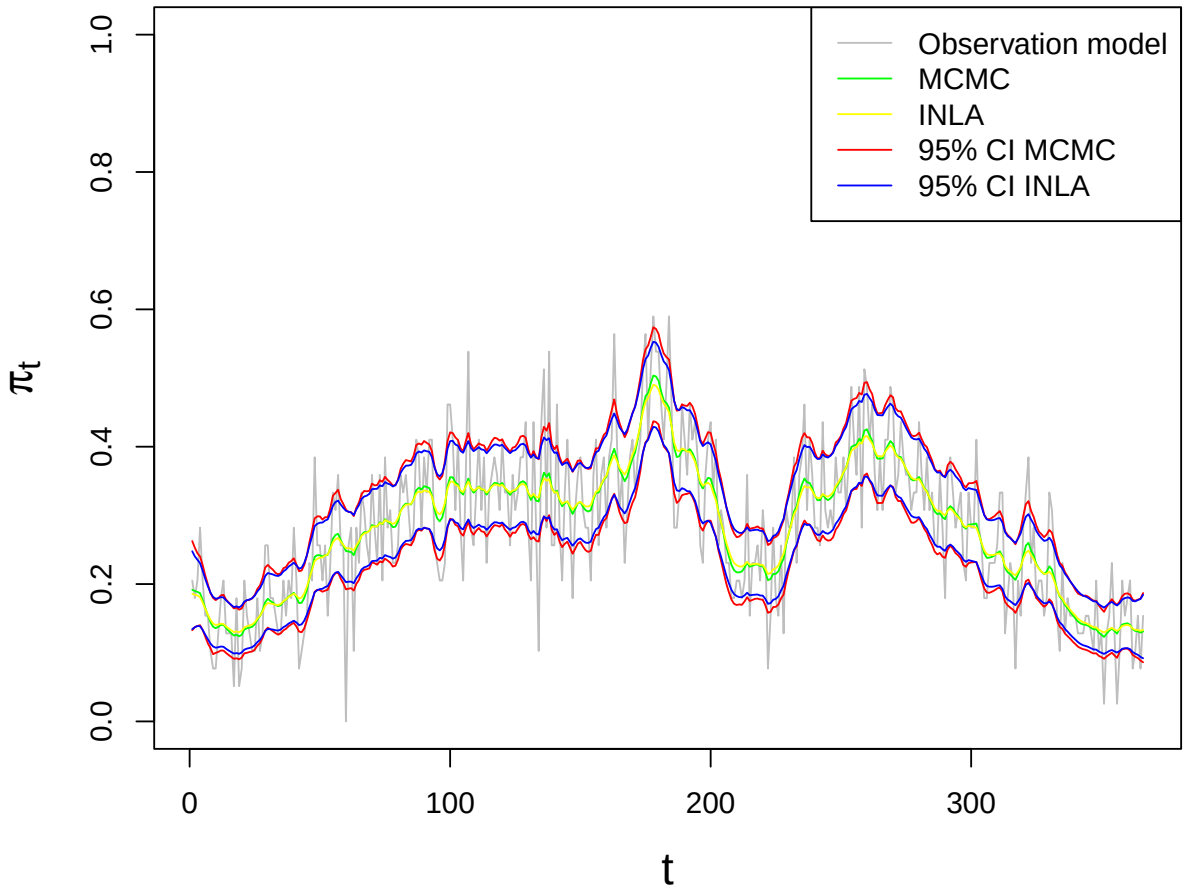


Figure 13: Comparison of the MCMC model and the INLA model

```
cat("Runtime INLA: ", runtime_INLA_1, "\n")
```

```
## Runtime INLA:  4.081
```

```
cat("Runtime INLA: ", MCMC_block_runtime, "\n")
```

```
## Runtime INLA:  205.475
```

As we can see from the plot, the differences in both the confidence intervals and the predictions are insignificant and very small. The fact that the models do not match exactly can be explained by the random component of the MCMC method. The major difference is the runtime, which is roughly 205 seconds for the MCMC method and close to 5 seconds for the INLA method, i.e. INLA is 41 times faster. Such a large difference clearly makes the case that INLA is the preferred method, even though both methods theoretically should converge to the same values.

### Task 2b): Robustness of the INLA schemes

Now to explore the robustness of the INLA method we must compare the approximation(Gaussian, Simplified Laplace and Adaptive) and integration strategies(ccd, grid and eb). This is a very simple modification to the

code and is found below

```
set.seed(123)

# Strategies and integration strategies
strategies = c("gaussian", "simplified.laplace", "adaptive")
int.strategies = c("ccd", "grid", "eb")

INLA_versions = list()
runtime_vec = c()
name_vec = c()
# Test every strategy with the integration strategy
for (i in 1:length(strategies)) {
    for (j in 1:length(int.strategies)) {
        before = proc.time()[3]
        control.inla = list(strategy = strategies[i], int.strategy = int.strategies[j])
        # Fit each model
        mod <- inla(n.rain ~ -1 + f(day, model = "rw1", constr = FALSE, hyper = list(prec = list(prior =
            param = c(alpha, beta)))), data = rain, Ntrials = n.years, control.compute = list(config = T
            family = "binomial", verbose = F, control.inla = control.inla)
        # Collect all information and models
        runtime = proc.time()[3] - before[[1]]
        INLA_versions[paste(strategies[i], int.strategies[j])] = list(mod)
        runtime_vec = append(runtime_vec, runtime)
        name_vec = append(name_vec, paste(strategies[i], int.strategies[j]))
    }
}


colors = c("#FF00FF", "#00FFFF", "red", "#0000FF", "purple", "green", "#FFFF00",
    "brown", "white", "orange")


# Plotting the different versions to get a glimpse of how robust they are
plot(INLA_versions[[1]]$summary.fitted.values$mean, type = "l", col = colors[1],
    ylim = c(0, 1), xlab = "Day", ylab = expression(pi["t"]))
for (i in 2:8) {
    lines(INLA_versions[[i]]$summary.fitted.values$mean, col = colors[i])
}
```

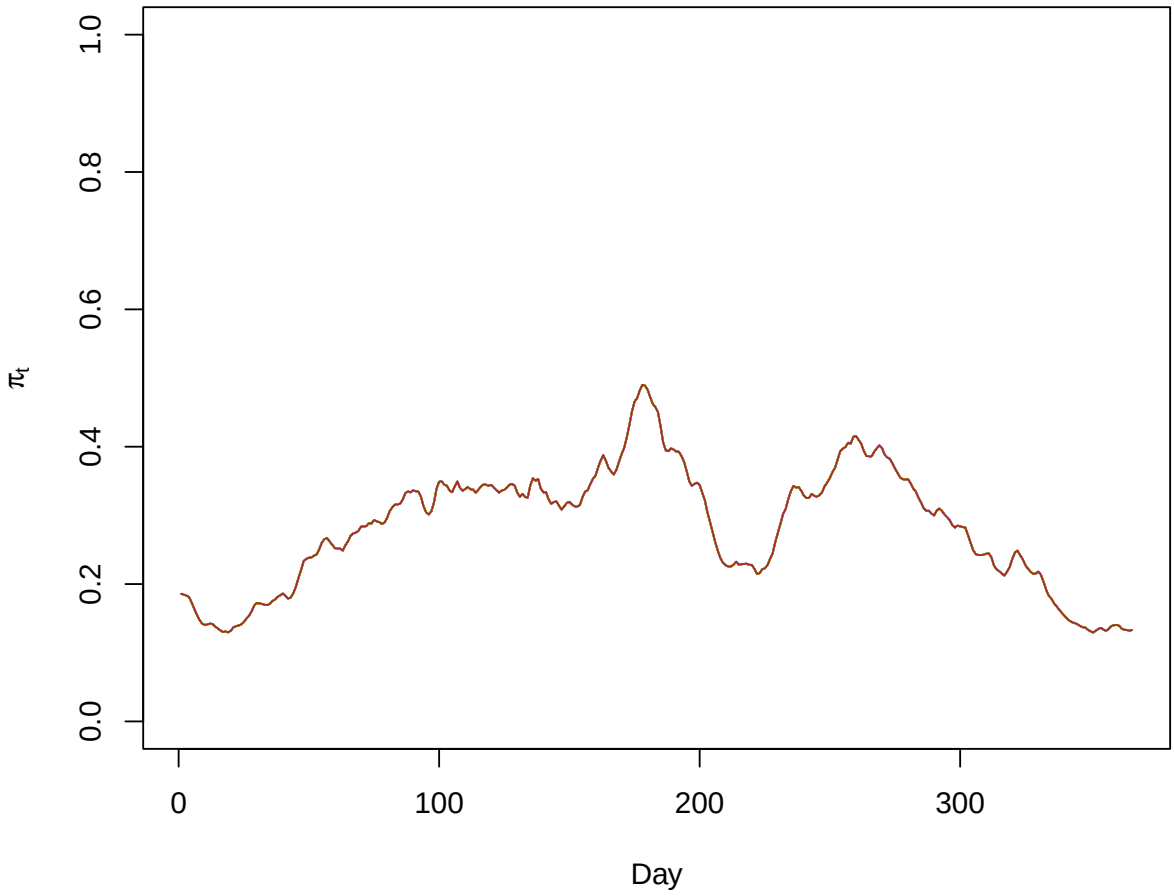


Figure 14: Different approximation and integration strategies for the INLA model

```
cat("Max runtime of models : ", max(runtime_vec), "\n")
```

```
## Max runtime of models :  4.072
```

```
cat("Min runtime of models : ", min(runtime_vec), "\n")
```

```
## Min runtime of models :  3.789
```

Judging from the plot, the differences between strategies are negligible, meaning that the INLA method is very robust based on the control.inla parameters. All methods also have a stable and very short computation time.

### Task 2c): Comparing different INLA models

We now compare three different models. The one we have already fitted without an intercept and a sum to zero constraint, one with an intercept and a sum to zero constraint, and one with neither an intercept nor a sum to zero constraint. Again, the INLA package makes this a very easy task to implement as we have done below in two parts.

```r
set.seed(123)

# Fit the second INLA model with intercept

control.inla = list(strategy = "simplified.laplace", int.strategy = "ccd")
before = proc.time()[3]

mod_2 <- inla(n.rain ~ f(day, model = "rw1", constr = T, hyper = list(prec = list(prior = "loggamma",
    param = c(alpha, beta)))), data = rain, Ntrials = n.years, control.compute = list(config = TRUE),
    family = "binomial", verbose = F, control.inla = control.inla)
# Calculate the runtime

runtime_INLA_2 = proc.time()[3] - before[[1]]

# Plot a comparison with and without intercept
plot(mod_1$summary.fitted.values$mean, type = "l", col = "#FF00FF", ylim = c(0, 1),
    xlab = "Day", ylab = expression(pi[t]))
lines(mod_2$summary.fitted.values$mean, col = "#00FFFF")
legend("topright", legend = c("INLA with intercept", "INLA without intercept"), col = c("#FF00FF",
    "#00FFFF"), lty = 1)
```

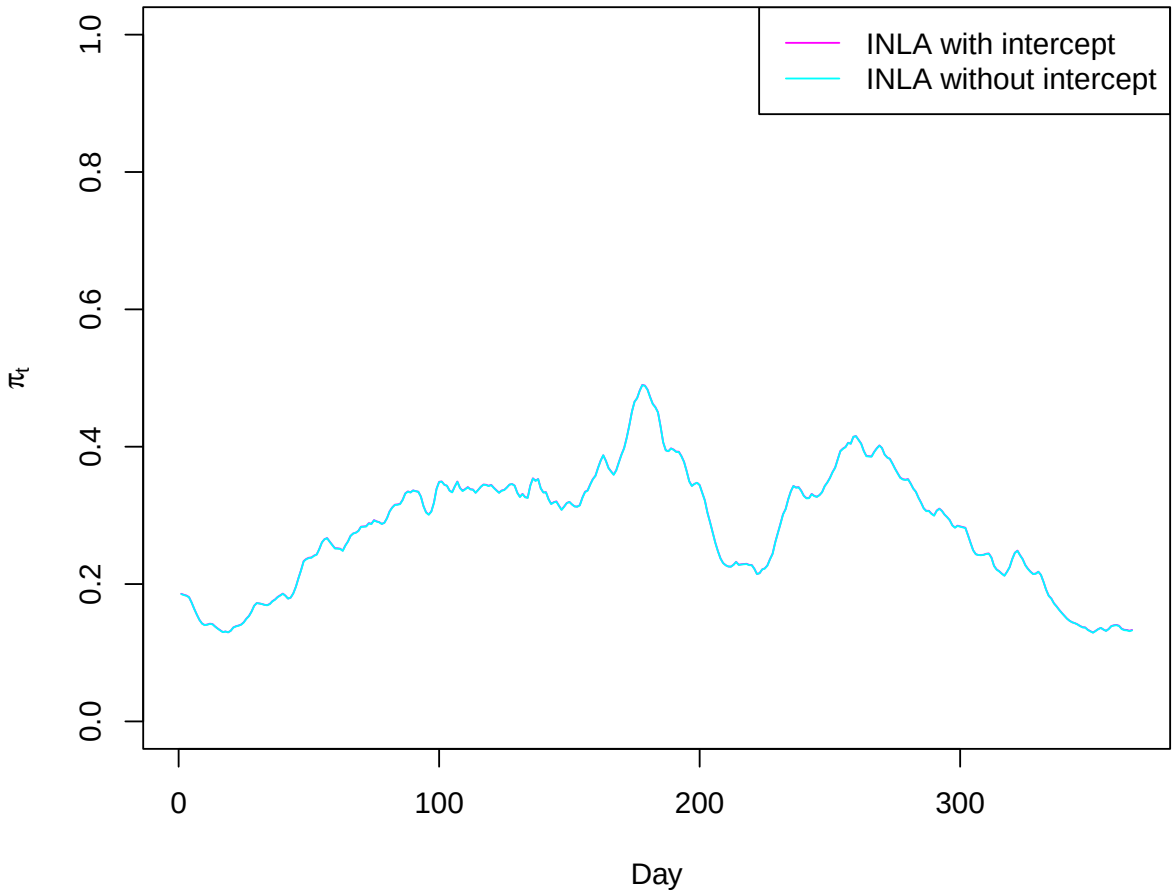


Figure 15: INLA model with and without intercept, both with sum to zero constraint

```
cat("Runtime : ", runtime_INLA_2, "\n")
```

```
## Runtime :  4.064
```

It appears that these models do not show any difference at all. The first model which is the same as the MCMC model in task 1, is mathematically written as

$$y_t|\tau_t \sim \text{Bin}(n_t, \pi(\tau_t))$$

and if we introduce an intercept we can write it as a multiple regression model

$$y_t|\tau_t \sim \text{Bin}(n_t, \pi(\eta_t))$$

where

$$\eta_t = \mu + \tau_t$$

This is a different model, but in our case we specify the proposal density to be symmetric, i.e. a random walk proposal(RW1). The symmetry makes the model mean invariant so the models should in fact yield the same results. This is consistent with what can be seen in figure 15.

```r
set.seed(123)
# Fit the third INLA model without intercept and without sum to zero constraint
control.inla = list(strategy = "simplified.laplace", int.strategy = "ccd")
before = proc.time()[3]

mod_3 <- inla(n.rain ~ f(day, model = "rw1", constr = F, hyper = list(prec = list(prior = "loggamma",
    param = c(alpha, beta)))), data = rain, Ntrials = n.years, control.compute = list(config = TRUE),
    family = "binomial", verbose = F, control.inla = control.inla)

runtime_INLA_3 = proc.time()[3] - before[[1]]

# Plot a comparison with and without sum to zero constraint
plot(mod_2$summary.fitted.values$mean, type = "l", col = "#00FFFF", ylim = c(0, 1),
    xlab = "Day", ylab = expression(pi[t]))
lines(mod_3$summary.fitted.values$mean, col = "#00FF00")
legend("topright", legend = c("INLA with sum-to-zero constraint", "INLA without sum-to-zero-constrain")
    col = c("#00FFFF", "#00FF00"), lty = 1)
```

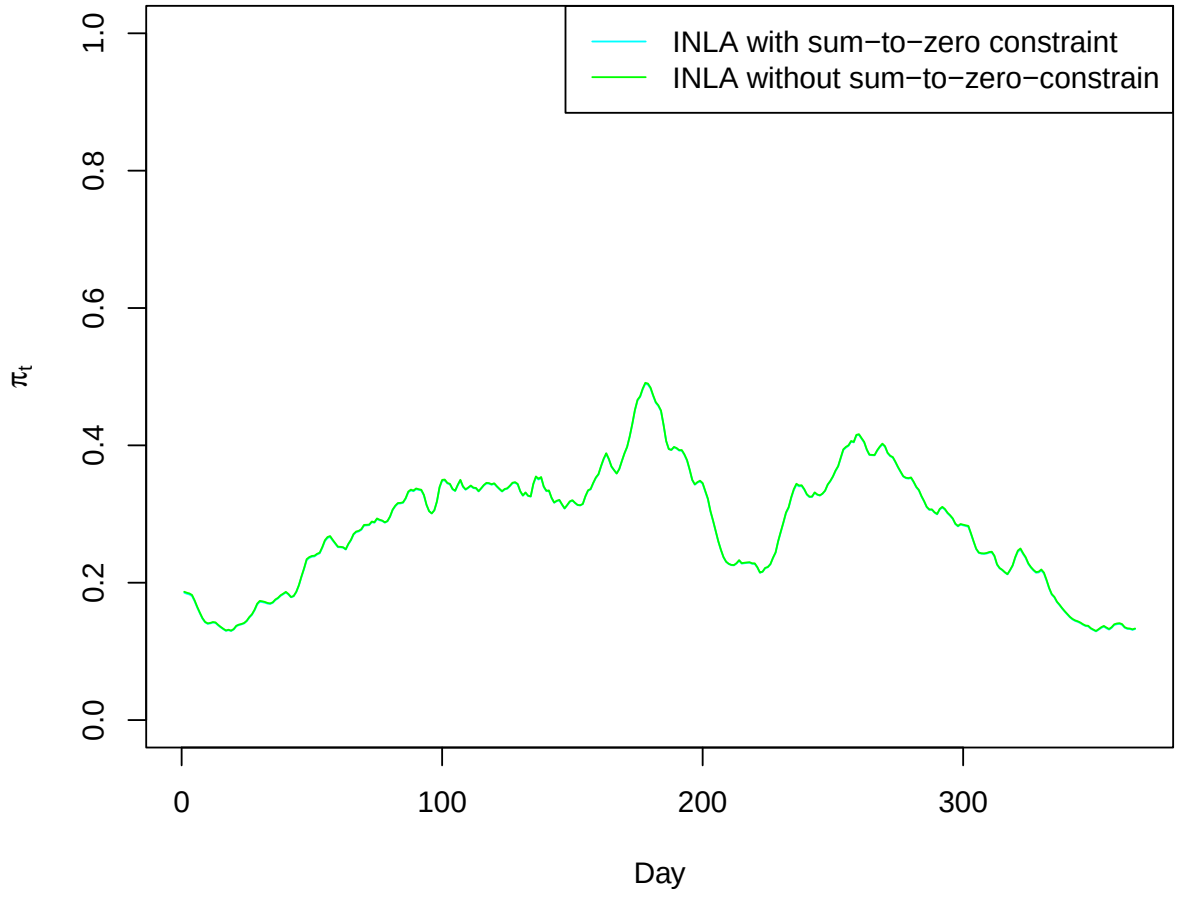


Figure 16: INLA model with and without sum to zero constraint, both with intercept

```
cat("Runtime : ", runtime_INLA_3, "\n")
```

```
## Runtime :  4.091
```

```
mod_2$summary.fixed
```

```
##                    mean         sd 0.025quant   0.5quant 0.975quant       mode
## (Intercept) -0.9877818 0.01951208   -1.02605 -0.9877812 -0.9495175 -0.9877798
##                         kld
## (Intercept) 5.533265e-11
```

```
mod_3$summary.fixed
```

```
##                    mean       sd 0.025quant   0.5quant 0.975quant       mode
## (Intercept) -0.9436314 11.29869   -23.1744 -0.9441745   21.29025 -0.9452261
##                         kld
## (Intercept) 5.985776e-09
```

When viewing the summary, we see that the values of the mean and mode differ somewhat and that the standard deviation increases significantly without the sum to zero constraint. The latter might be a consequence of the many possible values that $\tau_t$ can take on without the constraint. In figure 16 it appears there is no difference. This is a bit strange as we might expect the changes, even though they are small, to be noticeable. The conclusion is nonetheless that the sum to zero constraint does not effect the model in a significant matter and the intercept is not relevant for our case with the RW1 proposal.

Concluding we have implemented both MCMC for single and block update as well as different INLA models. We have seen that the major difference is how much faster the INLA method is. This combined with an easy implementation makes the INLA method a good alternative to MCMC methods when it is suitable.