Master project - backup

Halvard

2024-10-30

Simulation of a GMRF - random walks

Random walk 1

For the random walk 1 we have a diagonal precision matrix with 2 on the diagonal and -1 on the neighboring diagonals. The first and last elements on the diagonal are set to 1, so all rows sum to 0. k is some scale constant. We also assume a zero mean, but adding a different mean is just a location shift.

```r
#Creating the structure matrix
n = 4
R <- diag(2, n)
R[abs(row(R) - col(R)) == 1] <- -1
R[1, 1] <- 1
R[n, n] <- 1

R

k <- 1 #A constant connected top variance
Q <- k*R
mu <- rep(0, n)
```

Eigenvalues of R:

$$\lambda_i = 2 - 2\cos(\pi(i-1)/n), \quad i = 1, ..., n$$

Tried sampling with Algorithm 3.1, does not appear to work as expected. Maybe its missing some constraint, like the sum of x equals 0, or something with my implementation.

```r
# Eigenvalues
eigVecs <- eigen(Q, symmetric=TRUE) #Finds the eigenvectors

eigVals <- rep(0, n)
for(i in 2:n){
  eigVals[i] <- (2 - 2*cos(pi*(i - 1)/n))*k #Finds the eigenvalues
}

EVInv <- rev(sqrt(1/eigVals))  #Invert and square root for scaling

z <- rnorm(n-1) #Standard normal sample
y <- rep(0, n-1)
for(i in 1:(n-1)){
  y[i] <- z[i] * EVInv[i] #Computes the ys
}
x <- rep(0, n)
for(i in 1:(n-1)){
  x <- x + y[i]*eigVecs$vectors[1+i, 1:n]
}
```

```r
plot(1:n, x)
```

Another approach is to mimic the algorithms from the earlier sections, however our Q is singular

```r
A <- eigVecs$vectors[1, 1:n]
Q_inv <- solve(Q)
det(Q)

Q_new <- Q + A%*%t(A)

L <- t(chol(Q_new)) #The lower triangle cholesky decomp.
z <- rnorm(n)
v <- solve(t(L), z)



Q_inv <- solve(Q_new)
det(Q_new)

x_adj <- x - Q_inv%*%A *solve(t(A)%*%Q_inv%*%A)*(t(A)%*%x)
```

For now only using code below this point, the above code does not work.

```r
#Libraries
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.3.3
```

```r
library(tidyr)
```

```
## Warning: package 'tidyr' was built under R version 4.3.3
```

```r
library(ggpubr)
library(INLA)
```

```
## Warning: package 'INLA' was built under R version 4.3.3

## Loading required package: Matrix

##
## Attaching package: 'Matrix'

## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
## Loading required package: sp
```

```
## Warning: package 'sp' was built under R version 4.3.3
```

```
## This is INLA_24.05.01-1 built 2024-05-01 18:49:50 UTC.
##  - See www.r-inla.org/contact-us for how to get help.
##  - List available models/likelihoods/etc with inla.list.models()
##  - Use inla.doc(<NAME>) to access documentation
```

Basic random walk

The easiest way to simulate a random walk is through the assumption of independent normal distributed increments. We know that '

$$x_{t+1}|x_1,...,x_t,\sigma = x_{t+1}|x_t,\sigma \sim N(x_t,\sigma^2)$$

Thus, all we need to do is sample from the standard normal, scale them by σ and add them sequentially. Standard to assert that $x_0 = 0$.

Defining a basic plotting function that will come in handy

```
plot_realizations <- function(df, title, xlabel = "t", ylabel = "y", legend = TRUE){
  #Need to give in a dataframe with fitting colnames, used by the legend
  df$t <- 1:nrow(df)  # Create a time index from 1 to n
  df_long <- df %>% pivot_longer(cols = -t, names_to = "variable", values_to = "value")

  ggplot(df_long, aes(x = t, y = value, color = variable)) +
  geom_line() +
  labs(title = title,
       x = xlabel, y = ylabel) +
    if (legend) {theme(legend.title = element_blank())}
    else {theme(legend.position = "none")}
}
```

Now, lets define functionality for the random walk.

```
RW1 <- function(sigma, N){
  # sigma^2 is the variance for the transitions and N is the number of points
  x <- rep(0, N)
  z <- sigma * rnorm(N-1)
  for(j in 2:N){
    x[j] <- x[j-1] + z[j-1]
  }
  return(x)
}

#Also making a normalized RW1, ie. it sums to zero
Norm_RW1 <- function(sigma, N){
  # sigma^2 is the variance for the transitions and N is the number of points
  x <- rep(0, N)
  z <- sigma * rnorm(N-1)
  for(j in 2:N){
    x[j] <- x[j-1] + z[j-1]
  }
```

```r
  return(x -  mean(x)) #makes the mean zero
}

#Parameters for simulation of RW1
n <- 10
N <- 100
sigma <- 1

df <- data.frame(matrix(NA, nrow = N, ncol = n))
set.seed(0)
for (i in 1:n){
  df[, i] <- RW1(sigma, N)
}

# Plot all lines using ggplot
RW1_plot <- plot_realizations(df, "RW1 with N=100", legend = FALSE)
```

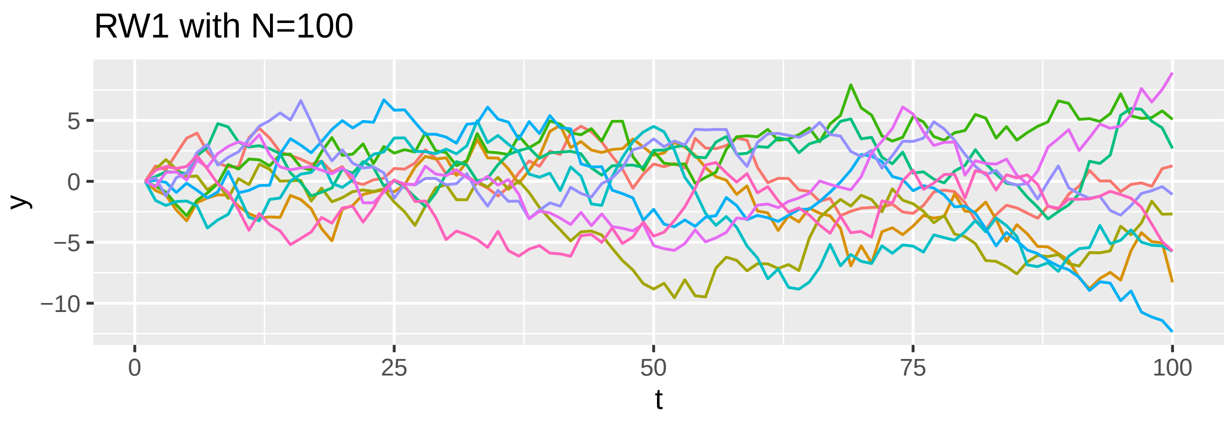
Random walk 2

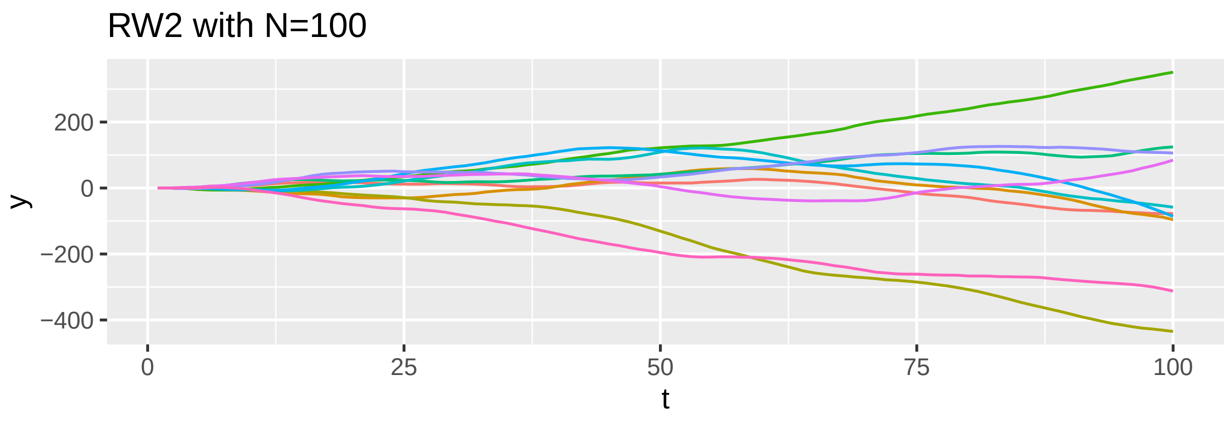
$$x_t - 2x_{t+1} + x_{t+2} \sim N(0, \sigma^2) x_{t+2} \sim N(2x_{t+1} - x_t, \sigma^2) x_{t+2} = 2x_{t+1} - x_t + \epsilon_t \epsilon_t \sim N(0, \sigma^2)$$

```r
RW2 <- function(sigma, N){
  # sigma^2 is the variance for the transitions and N is the number of points
  x <- rep(0, N)
  z <- sigma * rnorm(N-1)
  for(j in 3:N){
    x[j] <- 2*x[j-1] - x[j-2] + z[j-1]
  }
  return(x)
}

#Parameters for simulation of RW2
n <- 10
N <- 100
sigma <- 1

df2 <- data.frame(matrix(NA, nrow = N, ncol = n))
set.seed(0)
for (i in 1:n){
  df2[, i] <- RW2(sigma, N)
}

# Plot all lines using ggplot
RW2_plot <- plot_realizations(df2, "RW2 with N=100", legend = FALSE)
```

Visualize the plots.

```r
RW_figure <- ggarrange(RW1_plot, RW2_plot, ncol = 1)

RW_figure
```





Simulation study

We want to conduct a small simulation study to see if the adaptive models improve the standard models in situations with shocks.First we will start with assessing the performance on non.shocked data.

Simulation of non-shocked Gaussian data

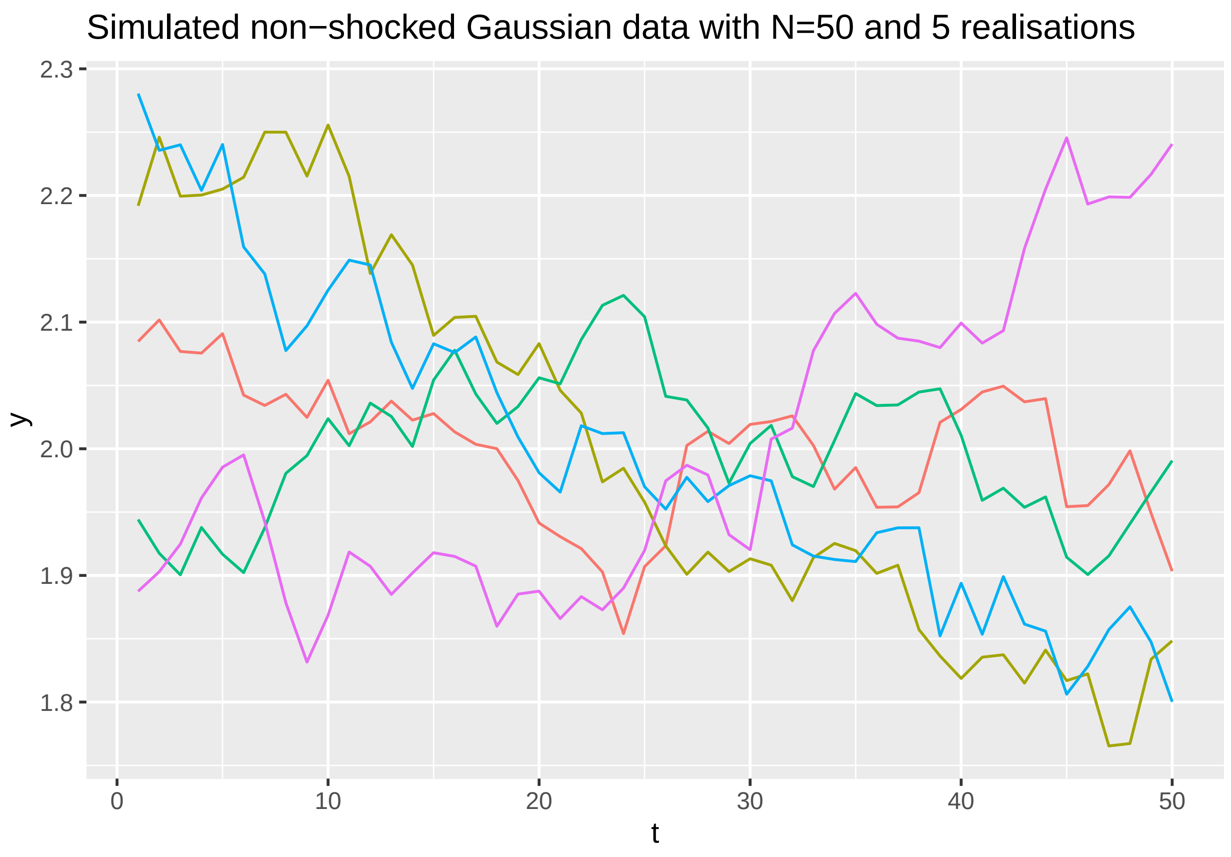
We will simulate data with a latent temporal structured random effect as a RW1, denoted $\mathbf{x}$. The total Bayesian hierarchical model can be described as

$$y_t|\eta_t \sim N(\eta_t, \sigma_t^2)\eta_t = \mu + x_t.$$

For the moment we assume constant σ_t for all timepoints, and choose some fixed σ_r for the random walk. First, lets make the general functions.

```
#function to simulate a realization y
sim_non_shocked_gaussian_data <- function(N, mu, sigma_obs, sigma_rw){
  #N timepoints, mean mu, and standard deviations observations and the RW1
  eta <- mu + Norm_RW1(sigma_rw, N)
  y <- sapply(eta, function(r) rnorm(1, mean = r, sd = sigma_obs))
  return(y)
}

sim_non_shocked_gaussian_dataframe <- function(N, mu, sigma_obs, sigma_rw, n, seed = 50){
```

```r
  set.seed(seed)
  df <- data.frame(matrix(NA, nrow = N, ncol = n))
  for(i in 1:n){
    df[, i] <- sim_non_shocked_gaussian_data(N, mu, sigma_obs, sigma_rw)
  }
  return(df)
}

#The dataframe for all non-shocked Gaussian data
N <- 50
n <- 100
sigma_obs <- 0.001
sigma_rw <- 0.03
mu <- 2
NSG_dataframe <- sim_non_shocked_gaussian_dataframe(N, mu, sigma_obs, sigma_rw, n)

#Visualizing some simulated data
plot_realizations(NSG_dataframe[, 1:5], "Simulated non-shocked Gaussian data with N=50 and 5 realisatio
```



Model evaluation of the the direct model for non-shocked Gaussian data

For our actual simulation study we are interested in $N = 50$ timepoints and 100 realizations. We will then fit a direct smooth model and an adaptive model to each realization and compare some model criteria, like DIC etc.

Root mean square eroor

A common model criteria is the root mean square error, or RMSE. We define a function to calculate this below.

```
RMSE <- function(data, preds){
  return(sqrt( sum((data - preds)**2) / length(data))) #definition of RMSE
}
```

Average proper logarithmic scoring rule(LS)

From a paper by Gneiting and Raftery (2007).

$$\mathrm{LS}(p, \omega) = \log p(\omega)$$

where p is the predicted distribution of a point, which we get from INLA, and ω is the observed value, which is the specific datapoint. We then take the average of the result for all the data points which unique prediction densities.

it does not work yet

```
LS <- function(res, data){
  #res is an inla object from calling a model on data, a vector of datapoints
  log_sum <- 0
  for (i in 1:length(data)){
    p <- (res$...something)(data[i]) # need some way to get the denisty of preds
    log_sum <- log_sum + log(p)
  }
  return(log_sum / length(data)) #returns the average
}
```

The direct smooth model with INLA

We fill fit the simple model in INLA with a latent layer as

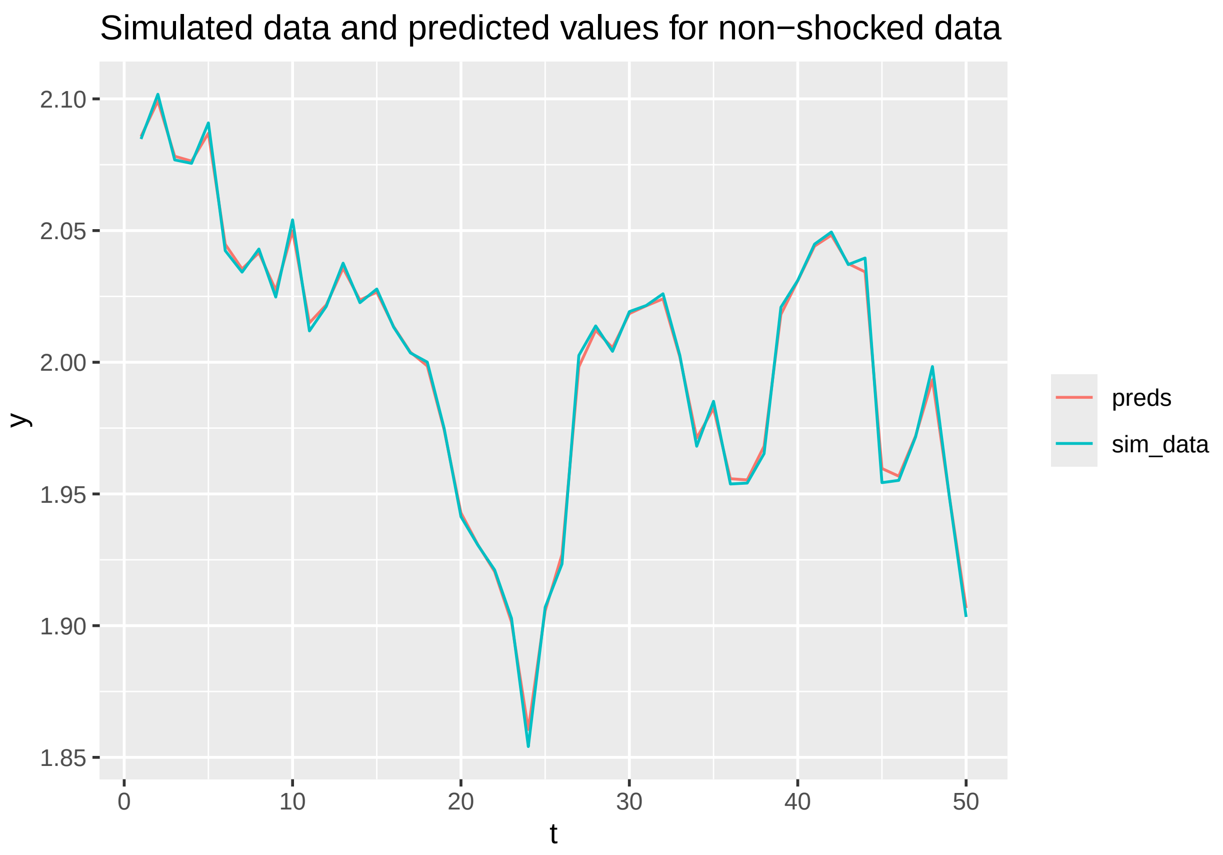
$$\eta_t = \mu + x_t$$

where x_t is a RW1 with some precision τ with a default prior. We use a Gaussian likelihood in the observation layer.

```
#Data preperation
NSG_data <- data.frame(matrix(c(NSG_dataframe[, 1], 1:N), nrow = N, ncol = 2))
colnames(NSG_data) <- c("y", "time") #makes the colnames match the formula

#The INLA model
formula <- y ~ f(time, model = "rw1") #intercept is included automatically
res <- inla(formula, family = "gaussian", data = NSG_data,
            control.compute = list(dic = TRUE, cpo = TRUE))
```

```
## Warning in .recacheSubclasses(def@className, def, env): undefined subclass
## "ndiMatrix" of class "replValueSp"; definition not updated
```

```r
#For plotting the data and the predicted values
preds <- res$summary.fitted.values$mean
plot_df <- data.frame(matrix(c(NSG_dataframe[, 1], preds), ncol = 2))
colnames(plot_df) <- c("sim_data", "preds") #for legends in the plot

plot_realizations(plot_df, "Simulated data and predicted values for non-shocked data")
```



We see that the model predictions align well with the data it is fit on.

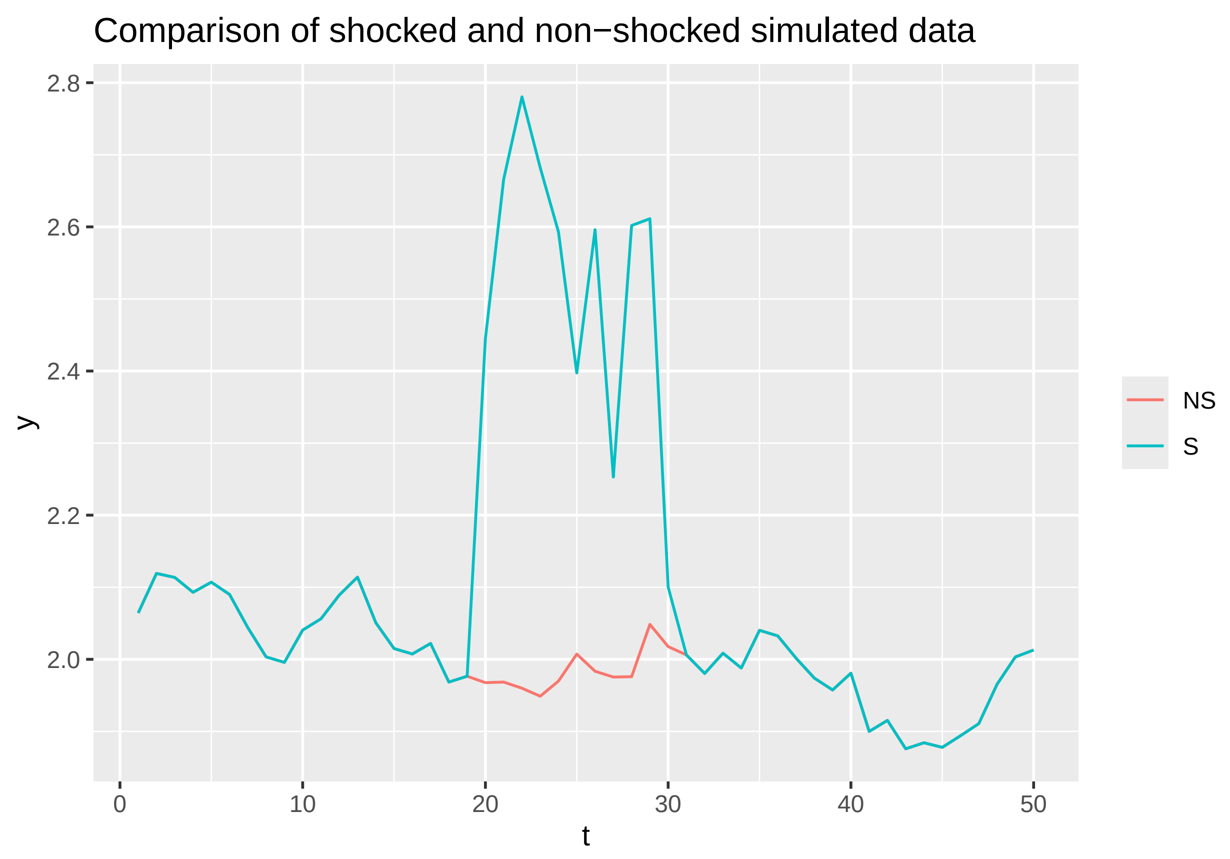
```r
Model_eval_NSG <- data.frame(matrix(NA, nrow = n, ncol = 2)) # need to increase 2 later
colnames(Model_eval_NSG) <- c("DIC_DS", "RMSE")

NSG_dataframe$t <- 1:N #adds the timecolumn

formula <- y ~ f(time, model = "rw1") #intercept is included automatically
for(i in 1:n){#iterate over each simulated realization
  test_data <- NSG_dataframe[, c(i, n + 1)] #gets the i-th realization and time
  colnames(test_data) <- c("y", "time") #makes the colnames match the formula
  res_DS <- inla(formula, family = "gaussian", data = test_data,
                 control.compute = list(dic = TRUE, cpo = TRUE))
  preds <- res_DS$summary.fitted.values$mean
  Model_eval_NSG[i, ] <- c(res_DS$dic$dic, RMSE(NSG_dataframe[, i], preds))
  #can get CPO by res_DS$cpo$cpo
}

#Model_eval_NSG
```

```r
#plot(res)
#summary(res)
#res$dic$dic
```

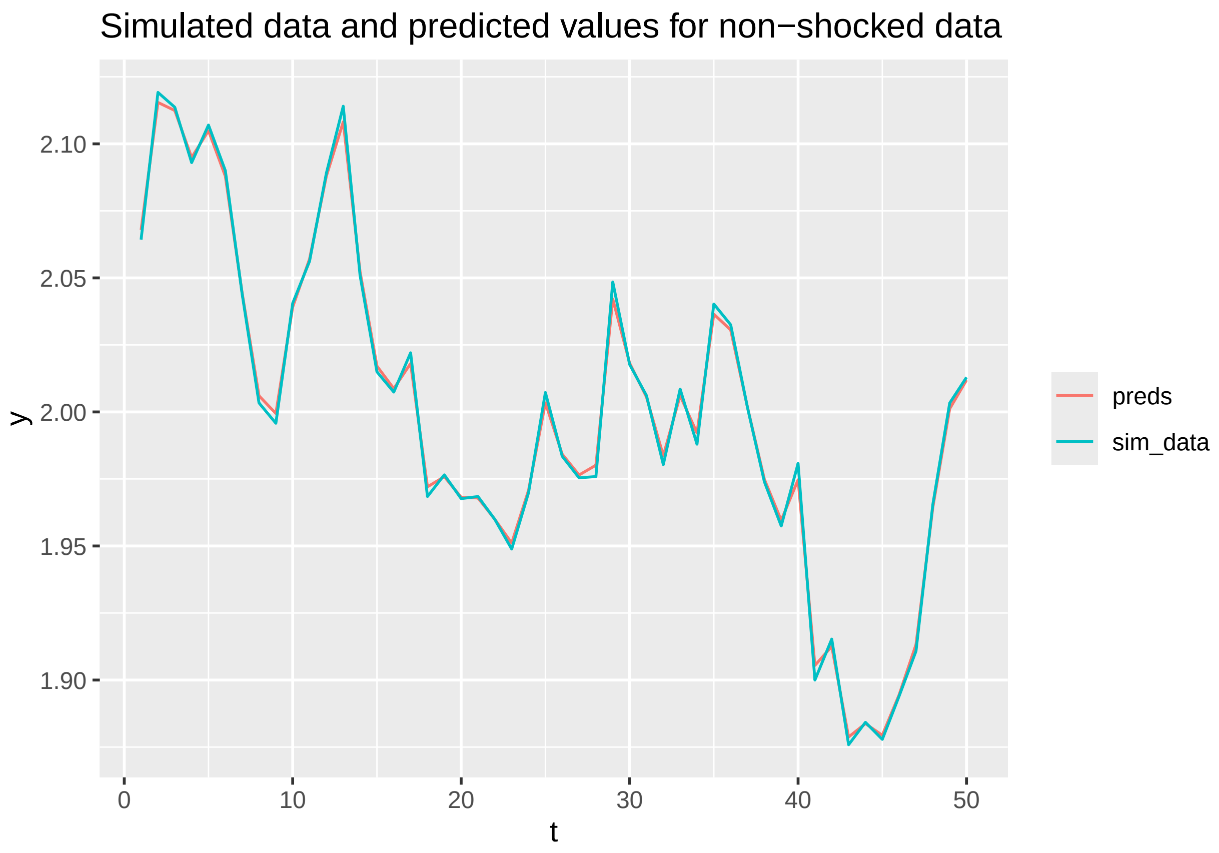
Simulation of shocked Gaussian data

We now want to simulate Gaussian data where we know that there are shocks on certain timepoints. This could be modeled by adding or subtracting a slightly randomized value from the chosen points. Lets say we want a shock from $t = 20$ to $t = 30$, which could be done by adding a $s_t \overset{iid}{\sim} N(0.4, 0.3)$ for instance.

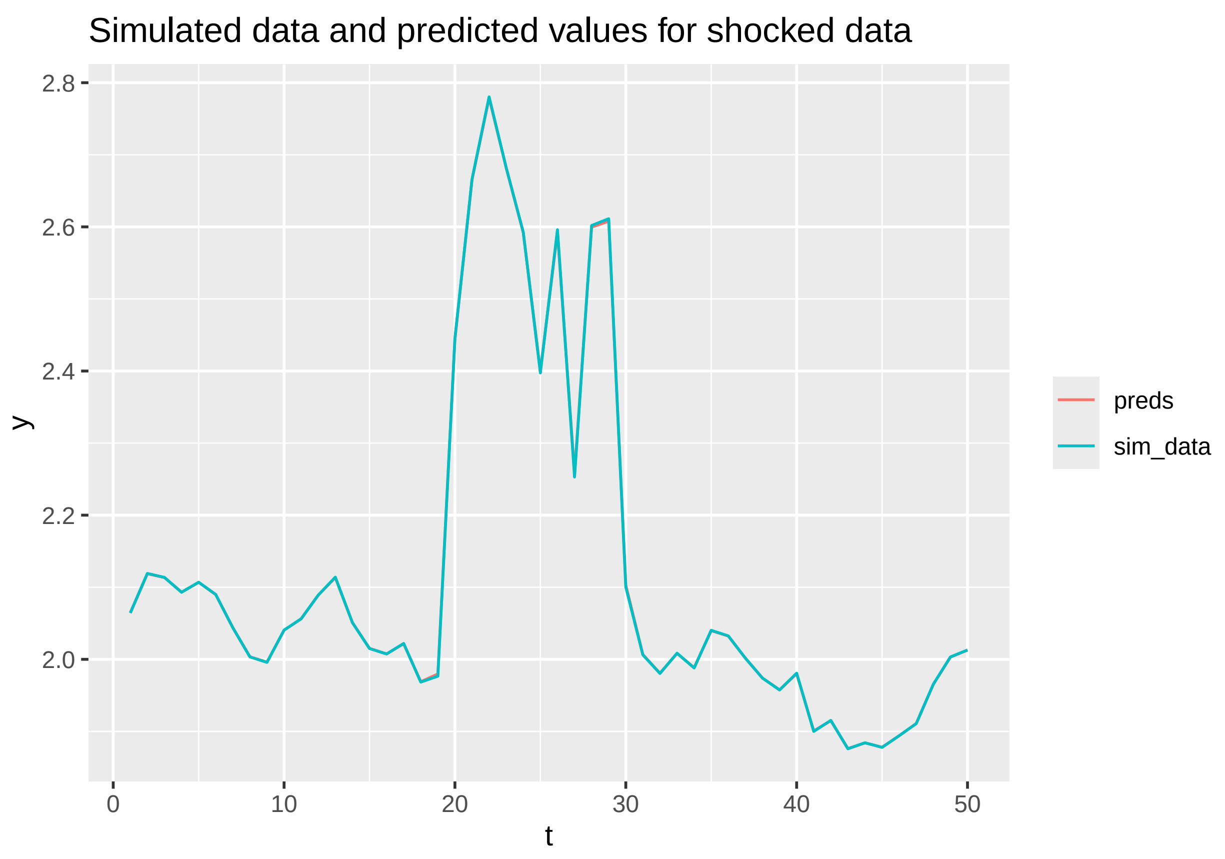
```r
#Parameters and constants
N <- 50
n <- 100
sigma_obs <- 0.001
sigma_rw <- 0.03
mu <- 2

#A comparison of non-shocked and shocked simulated data
NSG_data <- sim_non_shocked_gaussian_data(N, mu, sigma_obs, sigma_rw)
offset <- c(rep(0, 19),rnorm(11, 0.7, 0.3), rep(0, 20))
SG_data <- NSG_data + offset

example_dataframe <- data.frame(matrix(c(NSG_data, SG_data), nrow = N, ncol = 2))
colnames(example_dataframe) <- c("NS", "S") #Non-shocked and shocked

plot_realizations(example_dataframe, "Comparison of shocked and non-shocked simulated data", "t", "y")
```



```r
#Non-shocked data and inlas prediction for the smooth direct model
formula <- y ~ f(time, model = "rw1") #intercept is included automatically

test_data <- data.frame(matrix(c(NSG_data, 1:N), nrow = N, ncol = 2))
colnames(test_data) <- c("y", "time") #makes the colnames match the formula
res_NS <- inla(formula, family = "gaussian", data = test_data)
preds_NS <- res_NS$summary.fitted.values$mean

plot_df <- data.frame(matrix(c(NSG_data, preds_NS), ncol = 2))
colnames(plot_df) <- c("sim_data", "preds")

plot_realizations(plot_df, "Simulated data and predicted values for non-shocked data", "t", "y")
```

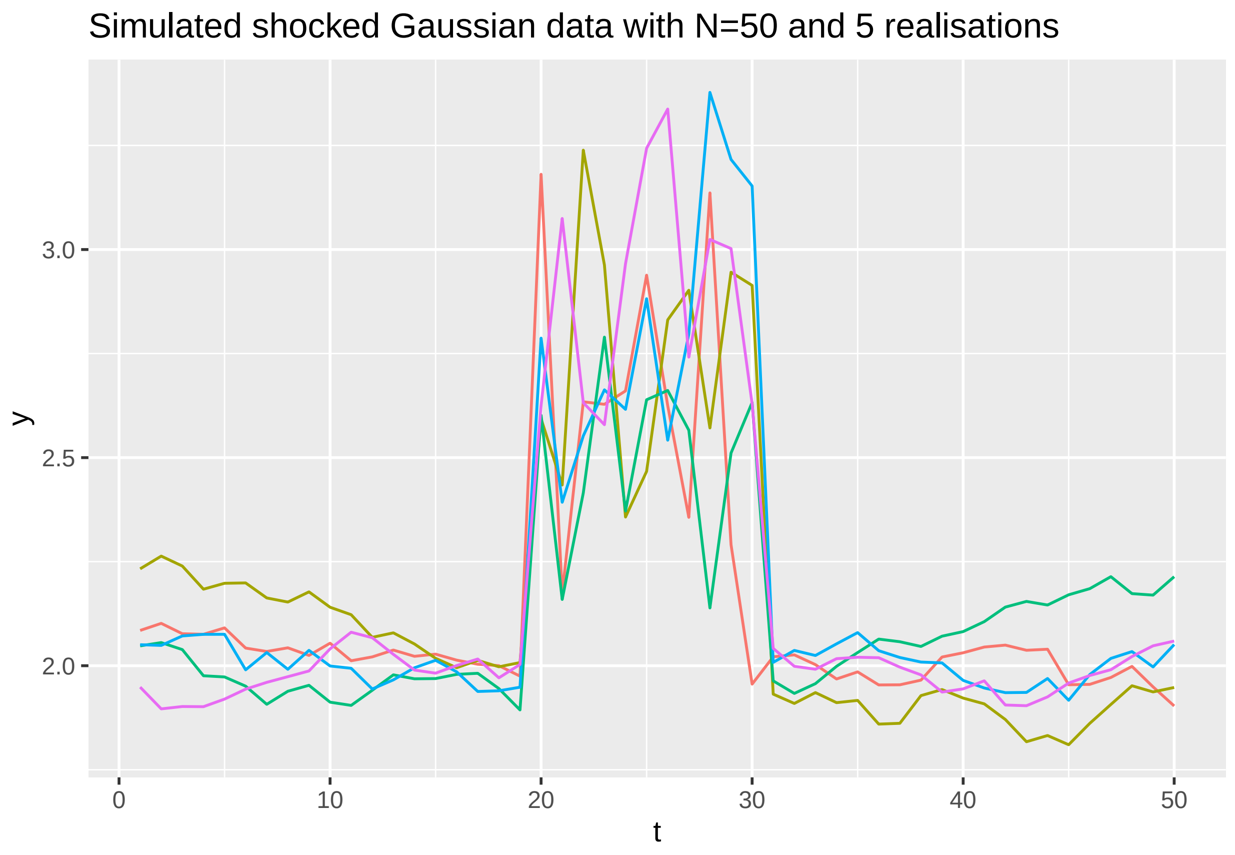


```r
#Shocked data and inlas prediction for the smooth direct model
test_data <- data.frame(matrix(c(SG_data, 1:N), nrow = N, ncol = 2))
colnames(test_data) <- c("y", "time") #makes the colnames match the formula
res_S <- inla(formula, family = "gaussian", data = test_data)
preds_S <- res_S$summary.fitted.values$mean

plot_df <- data.frame(matrix(c(SG_data, preds_S), ncol = 2))
colnames(plot_df) <- c("sim_data", "preds")

plot_realizations(plot_df, "Simulated data and predicted values for shocked data", "t", "y")
```



Lets make some general functions for simulating shocked Gaussian data.

```r
#function to simulate a shocked realization y
sim_shocked_gaussian_data <- function(N, mu, sigma_obs, sigma_rw, t_start = 20, t_end = 30, mu_offset =
  #N timepoints, mean mu, and standard deviations observations and the RW1
  eta <- mu + Norm_RW1(sigma_rw, N)
  y <- sapply(eta, function(r) rnorm(1, mean = r, sd = sigma_obs))
  offset <- c(rep(0, t_start - 1),rnorm(t_end - t_start + 1, mu_offset, sigma_offset), rep(0, N - t_end
  y_offset <- y + offset
  return(y_offset)
}

sim_shocked_gaussian_dataframe <- function(N, mu, sigma_obs, sigma_rw, t_start = 20, t_end = 30, mu_off
  set.seed(seed)
  df <- data.frame(matrix(NA, nrow = N, ncol = n))
  for(i in 1:n){
    df[, i] <- sim_shocked_gaussian_data(N, mu, sigma_obs, sigma_rw, t_start, t_end, mu_offset, sigma_o
  }
  return(df)
}

#Shocked Gaussian dataframe
SG_dataframe <- sim_shocked_gaussian_dataframe(N, mu, sigma_obs, sigma_rw, n=n)

#Visualizing some simulated data
plot_realizations(SG_dataframe[, 1:5], "Simulated shocked Gaussian data with N=50 and 5 realisations",
```



Implementing the adaptive random walk in the latent model in INLA

As this is somewhat complicated and new to me, I will start by a slightly easier example, namely the RW1. Can then also compare it to the already defined RW1 in INLA to ensure it works as intended. First some basic theory on defining random effects in INLA from https://becarioprecario.bitbucket.io/inla-gitbook/ch-newmodels.html .

Defining new latent random effects in INLA

New latent effects must be specified as GMRFs. So we need μ, Q and θ and its log-priors and initial values. Also need a graph, which I think can just be Q as well, and some log-normalizing constant. As θ is parametrized as $\theta_1 = \log(\tau)$ and $\theta_2 = \text{logit}(\rho)$ where τ is the precision and ρ is the spatial dependence, which I think we define to be 1.

```r
library(INLA)
sym_tridiag_matrix <- toeplitz(c(2, -1, rep(0, 3)))
sym_tridiag_matrix[1, 1] <- sym_tridiag_matrix[5, 5] <- 1 #first and last diag
inla.as.sparse(sym_tridiag_matrix)
```

```
## 5 x 5 sparse Matrix of class "dgTMatrix"
##
## [1,]  1 -1  .  .  .
## [2,] -1  2 -1  .  .
## [3,]  . -1  2 -1  .
## [4,]  .  . -1  2 -1
## [5,]  .  .  . -1  1
```

The general structure of the inla.rgeneric is shown below:

```
inla.rgeneric.somemodel = function(
  cmd = c("graph", "Q", "mu", "initial", "log.norm.const","log.prior", "quit"),
  theta = NULL)
{
  # for reference and potential storage for objects to
  # cache, this is the environment of this function
  # which holds arguments passed as `...` in
  # `inla.rgeneric.define()`.
  envir = parent.env(environment())
  graph = function(){ <to be completed> }
  Q = function() { <to be completed> }
  mu = function() { <to be completed> }
  log.norm.const = function() { <to be completed> }
  log.prior = function() { <to be completed> }
  initial = function() { <to be completed> }
  quit = function() { <to be completed> }

  # sometimes this is useful, as argument 'graph' and 'quit'
  # will pass theta=numeric(0) (or NULL in R-3.6...) as
  # the values of theta are NOT
  # required for defining the graph. however, this statement
  # will ensure that theta is always defined.

  if (!length(theta)) theta = initial()
  val = do.call(match.arg(cmd), args = list())
  return (val)
}

#if W is a needed argument
somemodel.model <- inla.rgeneric.define(inla.rgeneric.somemodel, W = W)
```

Implementing RW1 in INLA

In a RW1 we only have one hyperparameter, namely τ. So we get $\theta = \log(\tau)$ and the precision matrix is defined previously. Lets first define the inla.rgeneric function with all its necessary subfunctions.

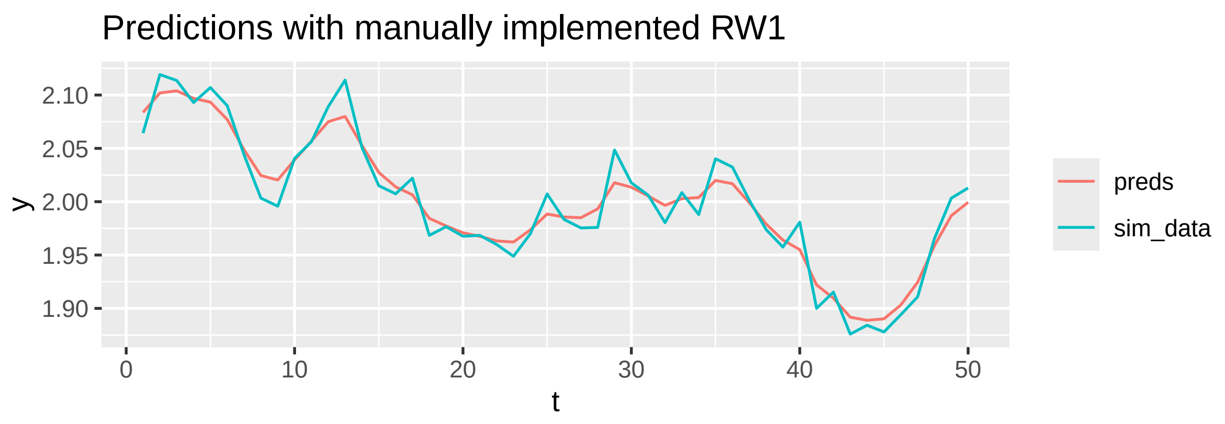
```
inla.rgeneric.RW1.model = function(
  cmd = c("graph", "Q", "mu", "initial", "log.norm.const","log.prior", "quit"),
  theta = NULL)
{
  #Input:
  #N is the number of timepoints

  envir = parent.env(environment())

  interpret_theta <- function() { return(list(prec = exp(theta[1L])))}

  graph <- function() {return(Q())}
```

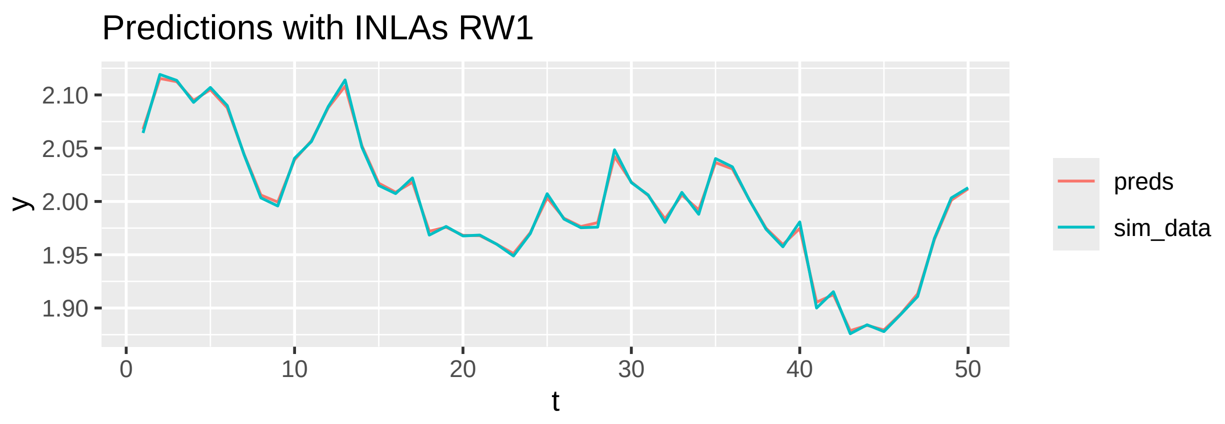
```r
  #Could only create the upper diagonal with Toeplitz in pracma,
  #or create it as sparse from the beginning
  Q <- function() {
    p <- interpret_theta()
    dense_R <- toeplitz(c(2, -1, rep(0, N - 2)))#2 on diag and -1 on firstdiags
    dense_R[1, 1] <- dense_R[N, N] <- 1 # 1 for first and last diag element
    return(inla.as.sparse(p$prec * dense_R)) #sparse representation
  }

  mu <- function() {return(numeric(0))}

  #Need to find a good initial value for theta on the log-scale
  initial <- function() {return(4)}#I think 4 is default initial for precisions

  #INLA will calculate the constant when numeric(0) is returned
  log.norm.const <- function() {return(numeric(0))}

  log.prior <- function() {#default: shape = 1, rate = 0.00005
    p <- interpret_theta()
    prior <- dgamma(p$prec, shape = 1 , rate = 0.00005 , log = TRUE) + p$prec
    return(prior)
  } #could maybe just add theta instead of log(prec)

  quit <- function() {return(invisible())}

  #to ensure theta is defined
  if (!length(theta)) theta = initial()

  vals <- do.call(match.arg(cmd), args = list())
  return(vals)
}

N <- 50 #is defined further up as well
RW1_model <- inla.rgeneric.define(inla.rgeneric.RW1.model, N = N)
```

Lets test if it works as intended on some data from earlier.

```r
#The INLA formula for a latent model with intercept and user defined RW1
formula_M <- y ~ f(time, model = RW1_model)

# From earlier
test_data <- data.frame(matrix(c(NSG_data, 1:N), nrow = N, ncol = 2))
colnames(test_data) <- c("y", "time") #makes the colnames match the formula

set.seed(46)
res_M <- inla(formula_M, family = "gaussian", data = test_data)

preds_M <- res_M$summary.fitted.values$mean
plot_df_M <- data.frame(matrix(c(NSG_data, preds_M), ncol = 2))
colnames(plot_df_M) <- c("sim_data", "preds")

plot_M <- plot_realizations(plot_df_M, "Predictions with manually implemented RW1")
```

```r
#The standard RW1 model from INLA
formula_I <- y ~ f(time, model = "rw1")
res_I <- inla(formula_I, family = "gaussian", data = test_data)

preds_I <- res_I$summary.fitted.values$mean
plot_df_I <- data.frame(matrix(c(NSG_data, preds_I), ncol = 2))
colnames(plot_df_I) <- c("sim_data", "preds")

plot_I <- plot_realizations(plot_df_I, "Predictions with INLAs RW1")

Comparison_figure <- ggarrange(plot_M, plot_I, ncol = 1)

Comparison_figure
```





```r
summary(res_M)
```

```
## Time used:
##     Pre = 0.488, Running = 4.68, Post = 0.117, Total = 5.28
## Fixed effects:
##              mean      sd 0.025quant 0.5quant 0.975quant  mode   kld
## (Intercept) 1.901 1078298   -2178575    1.301    2178585 1.629 0.001
##
## Random effects:
##   Name     Model
##     time RGeneric2
```

```
##
## Model hyperparameters:
##                                           mean    sd 0.025quant 0.5quant
## Precision for the Gaussian observations 109.67 1.682     106.28   109.69
## Theta1 for time                           4.72 0.013       4.69     4.72
##                                         0.975quant   mode
## Precision for the Gaussian observations     112.90 109.84
## Theta1 for time                               4.74   4.72
##
## Marginal log-Likelihood:  143.18
##  is computed
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

```r
summary(res_I)
```

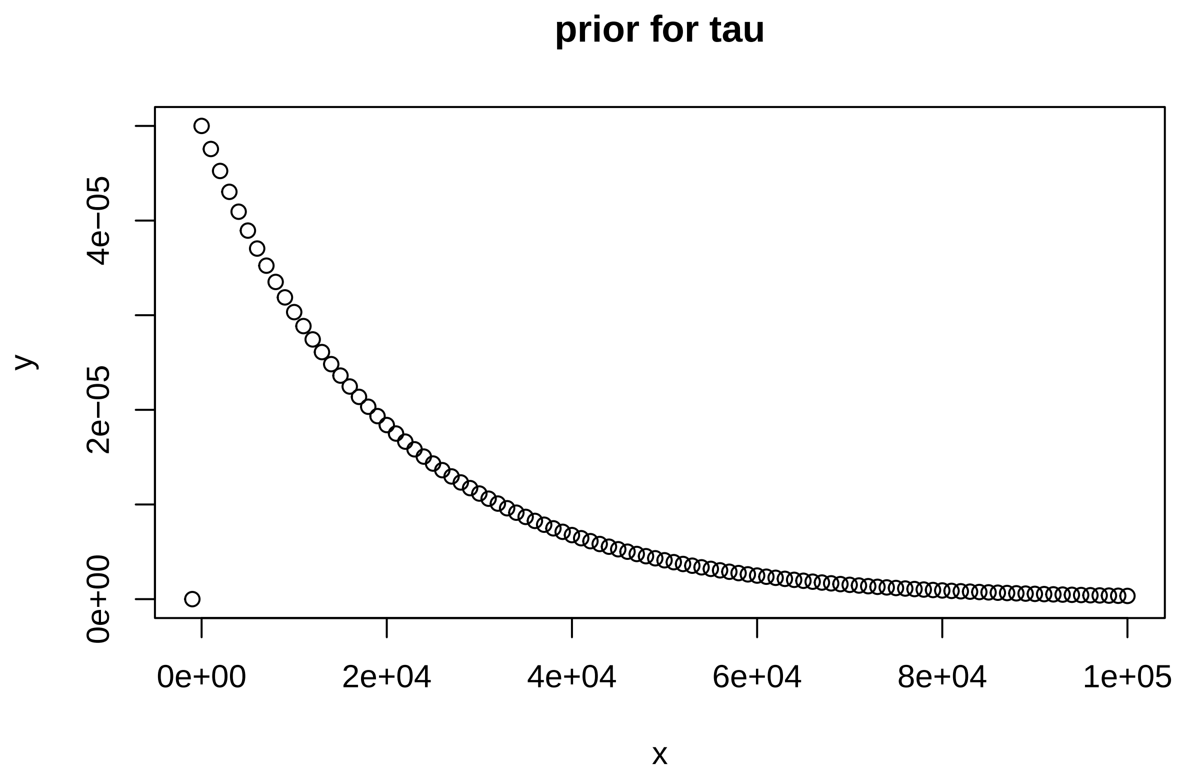
```
## Time used:
##     Pre = 0.516, Running = 0.577, Post = 0.135, Total = 1.23
## Fixed effects:
##              mean    sd 0.025quant 0.5quant 0.975quant mode kld
## (Intercept)    2 0.001      1.998        2      2.002    2   0
##
## Random effects:
##   Name     Model
##     time RW1 model
##
## Model hyperparameters:
##                                             mean       sd 0.025quant 0.5quant
## Precision for the Gaussian observations 27064.53 29391.38    2659.07 18233.27
## Precision for time                       1147.45   270.83     702.76  1117.88
##                                         0.975quant    mode
## Precision for the Gaussian observations  104505.87 7280.04
## Precision for time                         1762.76 1062.29
##
## Marginal log-Likelihood:  93.49
##  is computed
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

The INLA with manual RW1 sometimes breaks down, maybe 1 in 4 times, so I am trying to fix a seed to guarantee I get a result. Seems to work everytime with set.seed(46)

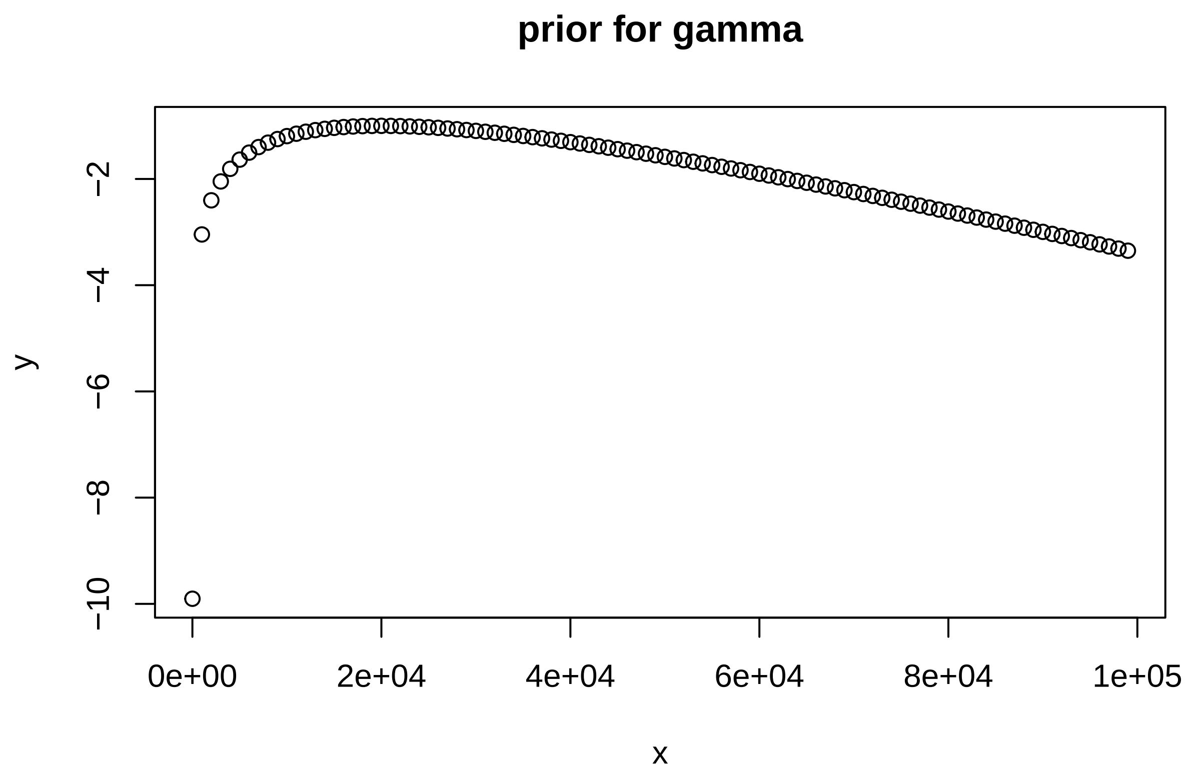
It seems to fit the data a lot worse than the default implementation. From the summaries we see that the intercept for the manual model is way off with a mean of 0.324 and a standard deviation of 933358, where as the INLA model gets a mean of 2 with low standard deviation. Which is what it is supposed to be. I assume something went wrong with interpret theta or the prior specification. The precision in the INLA model is also a lot higher, which is desired.

Some testing of where the mistakes are:

```r
x <- seq(from = -1000, to = 100000, by = 1000)
y <- dgamma(x, shape = 1, rate = 0.00005)
plot(x, y, main = "prior for tau")
```



```
x <- seq(from = 1, to = 100000, by = 1000)
y <- dgamma(x, shape = 1, rate = 0.00005, log = TRUE) + log(x)
plot(x, y, main = "prior for gamma")
```



The priors seem quite good in my opinion, might also have misunderstood completely. Very possible.

Sprøsmål: Should I scale the R-matrix when defining the precision matrix Q in the RW1? By geometric variance and so on

To do: Make use of the plot function everywhere - done Implement adaptive RW, maybe look at github and wakefield et al Choose model criteria - somewhat done Write about INLA - not started Use set.seed() to make results reproducible - continuous need for this

Code for ARW1 from github Alekshin Guendel

```
#The rgeneric inla function for user defined random effects, i e. a ARW1
inla.rgeneric.bym2.model = function(
    cmd = c("graph", "Q", "mu", "initial", "log.norm.const", "log.prior",
            "quit"),
    theta = NULL)
{
    # Assume we are passed the following inputs
    # n: the number of subdivisions (i.e. time points or regions)
    # Q_star: the scaled structure matrix for the structured component
    # gamma_tilde: the inverse eigenvalues of Q_star
    # U_prec, alpha_prec: U and alpha parameters for precision PC prior
    # U_phi, alpha_phi: U and alpha parameters for mixing parameter PC prior

    # for reference and potential storage for objects to
    # cache, this is the environment of this function
```

```r
    # which holds arguments passed as `...` in
    # `inla.rgeneric.define()`.
    envir = parent.env(environment())

    interpret.theta = function() {
        return(list(prec = exp(theta[1L]),
                    phi = 1 / (1 + exp(-theta[2L]))))
    }

    graph = function(){ return (Q()) }

    Q = function() {
        p = interpret.theta()
        D = (1 / (1 - p$phi)) * diag(n)
        QQ = rbind(cbind(p$prec * D, -sqrt(p$phi * p$prec) * D),
                   cbind(-sqrt(p$phi * p$prec) * D, Q_star + p$phi * D))
        return (inla.as.sparse(QQ))
    }

    mu = function() { return(numeric(0)) }

    log.norm.const = function() { return (numeric(0)) }

    log.prior = function() {
        p = interpret.theta()

        # Construct prior for the precision parameter
        lambda_prec = -log(alpha_prec) / U_prec
        prec_prior = log(lambda_prec) - log(2) - theta[1L] / 2 -
            lambda_prec / sqrt(p$prec)

        # Construct prior for the mixing parameter phi
        dU = sqrt(U_phi * sum(gamma_tilde - 1) -
                  sum(log(1 + U_phi * (gamma_tilde - 1))))
        lambda_phi = -log(1 - alpha_phi) / dU
        d2 = p$phi * sum(gamma_tilde - 1) -
            sum(log(1 + p$phi * (gamma_tilde - 1)))
        phi_prior = log(lambda_phi) + 2 * theta[2L] -
            2 * log1p(exp(theta[2L])) - log(2) - log(d2) / 2 -
            lambda_phi * sqrt(d2) +
            log(abs(sum( (gamma_tilde - 1) ^ 2 /
                         (1 + exp(theta[2L]) * gamma_tilde))))

        return(prec_prior + phi_prior)

    }

    initial = function() { return(c(4, 0)) }

    quit = function() { return (invisible()) }

    if (!length(theta)) theta = initial()
    val = do.call(match.arg(cmd), args = list())
    return (val)
}

# ------ The following is my work trying to understand the code -------
?do.call
```

```
## starting httpd help server ... done
```

```
?match.arg
#I believe the match.arg(cmd) are all the values we are interested in
# which were defined as functions with no arguments further up
# or by arguments passed globally to the function as specified at the top,
# so, val = c("graph","Q","mu","initial","log.norm.const","log.prior","quit")
#I assume these quantities are needed later?
?inla.rgeneric.bym2.model
```

```
## No documentation for 'inla.rgeneric.bym2.model' in specified packages and libraries:
## you could try '??inla.rgeneric.bym2.model'
```

```
#might need these, dont know
library(SUMMER)
library(readstata13)
library(dplyr)

vignette()

#### Get smoothed direct estimates w/ time fixed effect, adaptive bym2 ####
# Create structure matrices
conflict_years <- 1993:1999 - 1984
conflict_years_long <- rep(0, num_years)
conflict_years_long[conflict_years] <- 1

R_conflict <- matrix(0, num_years, num_years)
R_nonconflict <- matrix(0, num_years, num_years)
for(i in 1:num_years){
  if(i == 1){
    if(conflict_years_long[i] || conflict_years_long[i + 1]){
      R_conflict[i, i] <- 1
    }
    else{
      R_nonconflict[i, i] <- 1
    }
  }
  else if(i == num_years){
    if(conflict_years_long[i] || conflict_years_long[i - 1]){
      R_conflict[i, i] <- 1
    }
    else{
      R_nonconflict[i, i] <- 1
    }
  }
  else{
    if(conflict_years_long[i]|| (conflict_years_long[i - 1] &&
                                 conflict_years_long[i + 1])){
      R_conflict[i, i] <- 2

    }
    else if(conflict_years_long[i - 1] || conflict_years_long[i + 1]){
      R_conflict[i, i] <- 1
      R_nonconflict[i, i] <- 1
    }
    else{
      R_nonconflict[i, i] <- 2
    }
  }

  for(j in 1:num_years){
    if(abs(i - j) == 1){
      if(conflict_years_long[i] || conflict_years_long[j]){
        R_conflict[i, j] <- -1
      }
      else{
        R_nonconflict[i, j] <- -1
      }
    }
  }
}

R_1 <- R_nonconflict
R_2 <- R_conflict
scaled_Q <- INLA:::inla.scale.model.bym.internal(R_1 + R_2,
                                                 adjust.for.con.comp = TRUE)$Q
gv <- scaled_Q[1, 1] / (R_1 + R_2)[1, 1]
R_1_star <- gv * R_1
R_2_star <- gv * R_2
vals <- (1:num_years)[-num_years]
R_1_star_hat <- R_1_star[vals, vals]
R_2_star_hat <- R_2_star[vals, vals]
eps <- eigen(solve(R_1_star_hat + R_2_star_hat) %*% R_2_star_hat)$values
gamma_tilde <- c(1 / eigen(R_1_star + R_2_star)$values[1:(num_years - 1)], 0)
save(R_1_star, R_2_star, eps, gamma_tilde,
     file = "../Data/generated_data/structure_matrices.RData")

# Specify PC prior hyperparameters
pc.u.theta <- 0.75
pc.alpha.theta <- 0.75

# Fit model
adaptive_bym2_model <-
  INLA::inla.rgeneric.define(model = inla.rgeneric.adaptive.bym2.model,
                             n = num_years, R_1_star = R_1_star,
                             R_2_star = R_2_star, gamma_tilde = gamma_tilde,
                             eps = eps, U_prec = pc.u, alpha_prec = pc.alpha,
                             U_phi = pc.u.phi, alpha_phi = pc.alpha.phi,
                             U_theta = pc.u.theta,
                              alpha_theta = pc.alpha.theta)
constr <- list(A = matrix(c(rep(0, num_years), rep(1, num_years)),
                           nrow = 1, ncol = 2 * num_years), e = 0)
mod <- logit.est ~ time  +
  f(region.struct, model = adaptive_bym2_model,
    diagonal = 1e-06, extraconstr = constr, n = 2 * num_years) +
  f(survey.id,  model = "iid", hyper = hyperpc1)
options <- list(dic = TRUE, mlik = TRUE, cpo = TRUE,
                openmp.strategy = "default", return.marginals.predictor = TRUE)
control.inla <- list(strategy = "adaptive", int.strategy = "auto")
fit_adaptive_linear <-
  INLA::inla(mod, family = "gaussian", control.compute = options,
             data = dat,
             control.predictor = list(compute = TRUE),
             control.family =
               list(hyper =  list(prec = list(initial = log(1),
                                              fixed = TRUE))),
             scale = dat$logit.prec,
             control.inla = control.inla, verbose = FALSE)

out_adaptive_linear <- dat %>% select(region, years) %>%
  mutate(median = NA, lower = NA, upper = NA, logit.median = NA,
         logit.lower = NA, logit.upper = NA)
for (i in 1:nrow(dat)) {
  tmp.logit <-
    INLA::inla.rmarginal(1e+05,
                         fit_adaptive_linear$marginals.fitted.values[[i]])
  tmp <- expit(tmp.logit)
  out_adaptive_linear$median[i] <- median(tmp)
  out_adaptive_linear$lower[i] <- quantile(tmp, probs = 0.05)
  out_adaptive_linear$upper[i] <- quantile(tmp, probs = 0.95)
  out_adaptive_linear$logit.median[i] <- median(tmp.logit)
  out_adaptive_linear$logit.lower[i] <- quantile(tmp.logit, probs = 0.05)
  out_adaptive_linear$logit.upper[i] <- quantile(tmp.logit, probs = 0.95)
}

out_adaptive_bym2 <- out_adaptive_linear %>%
  filter(is.na(years))

smoothed_direct_adaptive_bym2$fit <- fit_adaptive_linear
smoothed_direct_adaptive_bym2$model <- mod

out_combined <- rbind(cbind(out_bym2, prior = "bym2"),
                      cbind(out_adaptive_bym2, prior = "adaptive bym2"))


# ------ The following is my work trying to understand the code -------
#first part is computing the structure and precision matrices, and scaled
# A lot of parameters and priors etc, then we define the ARW1 and include
# it in the formula for the inla call. Also a sum to zero constraint maybe?
# After the model call I am not sure whats happening.
```

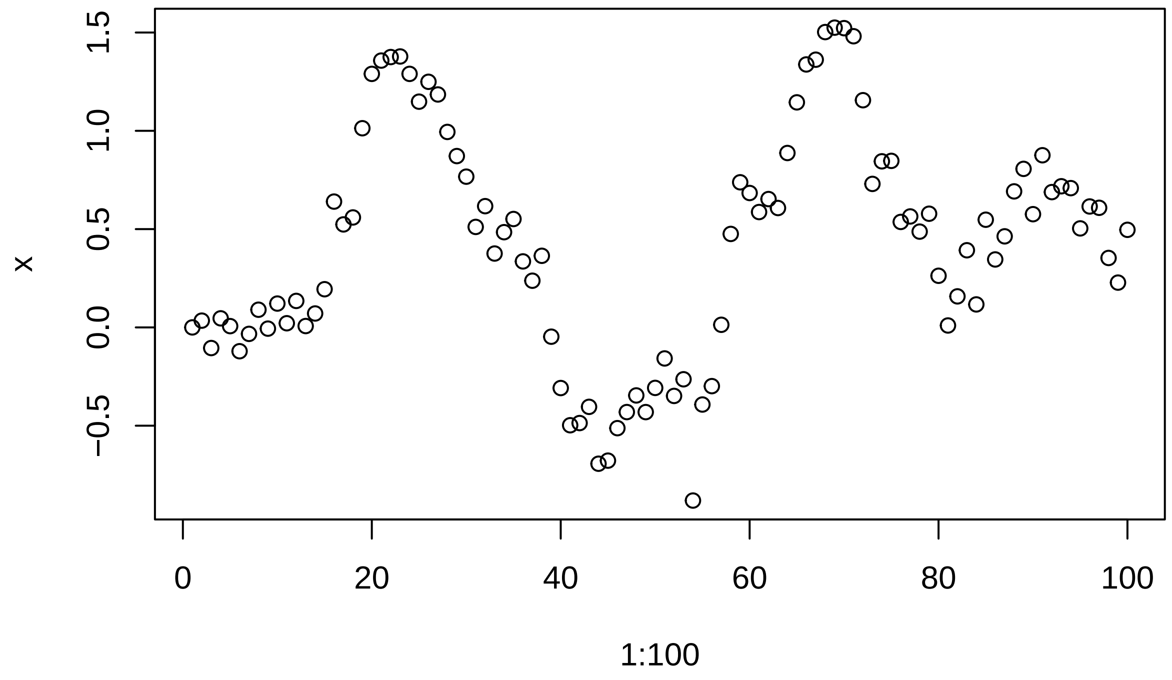
Simulation of non-shocked Poisson data

We will simulate data with a latent temporal structured random effect as a RW1. The total Bayesian hierarchical model can be described as

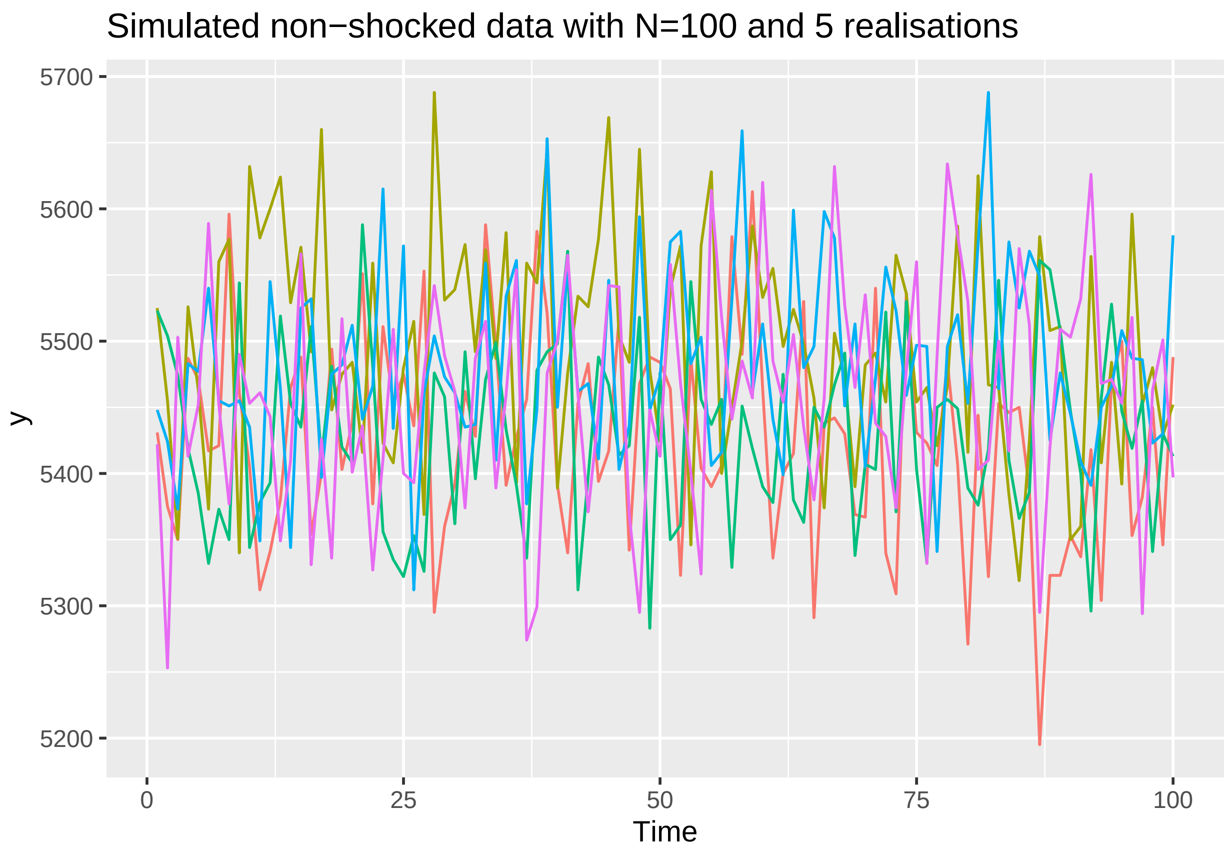
$$Y|\lambda \sim Poisson(E\lambda)\log\lambda_t = \mu + x_t$$

Where $\mathbf{x} \sim RW1(\tau)$ where we fix E, μ and τ for the simulations. When fitting models we will need to assign priors to them.

```
#Simulating non-shocked data

#Parameters
E = 100
mu = 1
sigma = 0.2
T = 100 #Number of time points

#A single simulation
x <- RW1(sigma, T)
rates <- E * exp(mu + x) #rates is E*lambda or E*exp(mu + x)
y <- sapply(rates, function(r) rpois(1, r)) #samples from the Poisson

plot(1:100, x)
```



```r
sim_non_shocked_data <- function(E, mu, sigma, T){
  x <- RW1(sigma, T)
  rates <- E * exp(mu + x) #rates is E*lambda or E*exp(mu + x)
  y <- sapply(rates, function(r) rpois(1, r)) #samples from the Poisson
  return(y) #The observed data
}

sim_non_shocked_dataframe <- function(E, mu, sigma, T, n, seed = 44){
  set.seed(seed)
  df <- data.frame(matrix(NA, nrow = T, ncol = n))
  for(i in 1:n){df[, i] <- sim_non_shocked_data(E, mu, sigma, T) }
  return(df)
}

test <- sim_non_shocked_dataframe(100, 4, 0.001, 100, 5)

test$t <- 1:nrow(test)  # Create a time index from 1 to n

# Reshape the dataframe to long format
test_long <- test %>% pivot_longer(cols = starts_with("X"), names_to = "variable", values_to = "value")

# Plot all lines using ggplot
ggplot(test_long, aes(x = t, y = value, color = variable)) +
  geom_line() +
  labs(title = "Simulated non-shocked data with N=100 and 5 realisations",
       x = "Time", y = "y") + theme(legend.position = "none")
```



The code above seems to work fine, but should possibly tune some of the parameters, or maybe the plot of the ys should look insane.

Now the next step is to fit some models with INLA, we will start with the direct smoothed model. ## Fitting the model with INLA

```
#need to input the data as a dataframe with a y column and time column
formula <- y ~ f(time, model = "rw1") #intercept is included automatically

test_data <- data.frame(matrix(c(test[, 2], 1:100, rep(E, 100)), nrow = 100, ncol = 3)) # should make t
colnames(test_data) <- c("y", "time", "E")

res <- inla(formula, E = E, family = "poisson", data = test_data)
plot(res)
summary(res)
```

```
## Time used:
##     Pre = 0.444, Running = 0.549, Post = 0.107, Total = 1.1
## Fixed effects:
##               mean    sd 0.025quant 0.5quant 0.975quant  mode kld
## (Intercept) 4.007 0.001      4.004    4.007      4.009 4.007   0
##
## Random effects:
##   Name     Model
##     time RW1 model
##
```

```
## Model hyperparameters:
##                        mean       sd 0.025quant 0.5quant 0.975quant     mode
## Precision for time 77408.79 33068.67   29533.42 71866.94  156982.65 60908.68
##
## Marginal log-Likelihood:  -595.86
##  is computed
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

```
?inla
```

Seems like it works quite well, the posterior distributions align rather well with the true parameters for mean and precision.

The following is code from a PowerPoint on Inla by Andrea Riebler, as well as some additional testing and modification by myself.

```
library(INLA)
```

A simple linear regression

```
# Generate data
x = sort(runif(100))
y = 1 + 2*x + rnorm(n = 100, sd = 0.1)

# Run inla
formula = y ~ 1 + x
result = inla( formula ,
  data = list(x = x , y = y) ,
  family = " gaussian ")

# Get summary
summary ( result )
```

```
## Time used:
##     Pre = 1.02, Running = 0.545, Post = 0.0567, Total = 1.62
## Fixed effects:
##              mean    sd 0.025quant 0.5quant 0.975quant  mode kld
## (Intercept) 0.997 0.018      0.962    0.997      1.032 0.997   0
## x           1.994 0.031      1.932    1.994      2.055 1.994   0
##
## Model hyperparameters:
##                                           mean    sd 0.025quant 0.5quant
## Precision for the Gaussian observations 122.44 17.32      90.92   121.63
##                                         0.975quant   mode
## Precision for the Gaussian observations     158.67 120.00
##
## Marginal log-Likelihood:  81.64
##  is computed
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

```
plot(result)

m = result$marginals.fixed[[1]]
plot(m)
plot(inla.smarginal(m))
```

Adding a random effect

```
# Generate AR (1) sequence
t = 1:100
ar = rep(0, 100)
for(i in 2:100)
  ar[i] = 0.8 *ar[i -1]+ rnorm(n = 1, sd = 0.1)

# Generate data with AR (1) component
x = runif(100)
y = 1 + 2*x + ar + rnorm(n = 100 , sd = 0.1)

# Run inla
formula = y ~ 1 + x + f(t, model="ar1")
result = inla(formula,
  data = list(x = x, y = y, t = t) ,
  family = "gaussian")

# Get summary
summary(result)
```

```
## Time used:
##     Pre = 0.467, Running = 0.558, Post = 0.184, Total = 1.21
## Fixed effects:
##              mean    sd 0.025quant 0.5quant 0.975quant  mode kld
## (Intercept) 0.985 0.064      0.859    0.984      1.119 0.984   0
## x           1.966 0.039      1.889    1.966      2.043 1.966   0
##
## Random effects:
##   Name     Model
##     t AR1 model
##
## Model hyperparameters:
##                                            mean     sd 0.025quant 0.5quant
## Precision for the Gaussian observations 165.110 55.900     82.464  156.171
## Precision for t                          40.723 14.434     18.588   38.659
## Rho for t                                 0.847  0.066      0.692    0.857
##                                         0.975quant    mode
## Precision for the Gaussian observations    299.839 139.665
## Precision for t                             74.628  34.855
## Rho for t                                    0.945   0.878
##
## Marginal log-Likelihood:  39.92
##  is computed
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

Prediction for an unobserved location

```r
# Add new location
x = c(x , 0.3)
t = c(t, 101)
y = c(y , NA)

# Re - compute
result.pred = inla(formula,
    data = list(x = x, t = t, y = y),
    family = "gaussian",
    control.predictor = list(compute = TRUE))

#m = result$marginals.linear.predictor

#m = result.pred$marginals.linear.predictor[[101]]
#round (result.pred$summary.linear.predictor[101 ,], 3)

#plot(m)
#lines(inla.smarginal(m))
```

Virker ikke helt med den marginal linear predictor greia.

Smooting binary time seires

```r
data ( Tokyo )
head ( Tokyo )
```

```
##   y n time
## 1 0 2    1
## 2 0 2    2
## 3 1 2    3
## 4 1 2    4
## 5 0 2    5
## 6 1 2    6
```

```r
head ( Tokyo ,4)
```

```
##   y n time
## 1 0 2    1
## 2 0 2    2
## 3 1 2    3
## 4 1 2    4
```

```r
# Specify linear predictor
formula = y ~ -1 + f(time , model ="rw2", cyclic = TRUE)

# Run model
result = inla(formula,
    family = "binomial",
    Ntrials = n,
    data = Tokyo)
```

```r
plot(result)

#Transform to probability
result = inla ( formula,
    family = "binomial",
    Ntrials = n,
    data = Tokyo,
    control.predictor = list(compute = TRUE))
plot(result)
```

Add weights to random components $formula = y \sim \mu + ... + f(idx, weight, model = ..., ...)$ makes the random effect term have a weight, from $\eta_i = ... + f_{idx_i}$ to $\eta_i = ... + weight_{idx_i} f_{idx_i}$. So, it adds a weight parameter.

Changing the prior

```r
# Old way
formula = y ~ f(idx, model = "iid", prior = "loggamma",
    param = c(1, 0.1), initial = 4, fixed = FALSE)

# New way
hyper = list(prec = list(prior = "loggamma",
    param = c(1, 0.1),
    initial = 4,
    fixed = FALSE))

formula = y ~ f(idx,model = "iid", hyper = hyper) + ...

inla.models()$latent$iid$hyper
```

```
## $theta
## $theta$hyperid
## [1] 1001
## attr(,"inla.read.only")
## [1] FALSE
##
## $theta$name
## [1] "log precision"
## attr(,"inla.read.only")
## [1] FALSE
##
## $theta$short.name
## [1] "prec"
## attr(,"inla.read.only")
## [1] FALSE
##
## $theta$prior
## [1] "loggamma"
## attr(,"inla.read.only")
## [1] FALSE
##
## $theta$param
## [1] 1e+00 5e-05
## attr(,"inla.read.only")
## [1] FALSE
##
## $theta$initial
## [1] 4
## attr(,"inla.read.only")
## [1] FALSE
##
## $theta$fixed
## [1] FALSE
## attr(,"inla.read.only")
## [1] FALSE
##
## $theta$to.theta
## function (x)
## log(x)
## <bytecode: 0x00000171fef39c48>
## <environment: 0x00000171fef4d5b0>
## attr(,"inla.read.only")
## [1] TRUE
##
## $theta$from.theta
## function (x)
## exp(x)
## <bytecode: 0x00000171fef39b30>
## <environment: 0x00000171fef4d5b0>
## attr(,"inla.read.only")
## [1] TRUE
```

Assign your own prior

```r
# use suitable support points x
lprec = seq( -10, 10, len =100)

# link the x and corresponding y values into a string which begins with " table :""
#With some prior function
prior.table = INLA:::inla.paste(c("table:",cbind(lprec, prior.function(lprec))))

hyper = list(prec = list(prior = prior.table))
```

Repeated poisson count:

$$\log(\mu_{jk}) = \alpha_0 + \alpha_1 \log(Base_j/4) + \alpha_2 TRT_j + \alpha_3 TRT_j \log(Base_j/4) + \alpha_4 Age_j + \alpha_5 V4 + Ind_j + \beta_{jk}$$

α_i follows a $N(0, \tau_\alpha)$ where tau_α is known. Ind_j and β_{jk} are the same with gamma hyperpriors on their respective precisions.

```r
data(Epil)
head(Epil,n =3)

formula = y ~  Base*Trt + Age + V4 +
          f(Ind, model ="iid",
          hyper = list(prec = list(prior = "loggamma",
          param = c(1, 0.01)))) +
          f(rand, model ="iid",
          hyper = list(prec = list(prior = "loggamma",
          param = c(1, 0.01))))

result = inla(formula, family ="poisson", data = Epil,
  control.fixed = list(prec.intercept = 0.001 , prec = 0.001))
```

In the control.fixed above prec.intercept is just for the intercept precision while prec is for all other fixed effects. control.predictor can compute posterior marginals for linear predictors with compute, and apply linear transformations with A. control.compute can compute measures of fit, like dic, mlik and cpo. Also many other control statements. This is done below.

```r
result = inla(formula, #some formula
          data = data, #some data
          control.compute = list(mlik = TRUE)) #Computes the marginal likelihood

# See result
result$mlik
result$dic$dic
result$cpo$cpo
result$cpo$pit
```

Can exchange mlik, marginal likelihood, with dic, Deviance information criteria, cpo, Conditional predictive ordinate or cpo for probability integral transform. DIC measures the trade-off between goodness of fit while trying to keep the model simple, ie. few parameters. CPO measures the fit with Bayesian hold on out cross validation for all data points. PIT:

$$Prob(Y_i \leq y_i^{obs} | \mathbf{y}_{-i})$$

There are several tools to extend these standard models like replicate and group, multiple likelihoods, copy, linear transformation of η (A matrix), linear combinations, values and remote computing. Replicate simulates the random effects the requested number of times as f(. . . , replicate = r [, nrep = nr]).

```r
#need to define i and r manually
formula = y ~ f(i , model ="ar1", replicate =r) + intercept -1
result = inla(formula, family = "poisson",
    data = data.frame(y, i, r, intercept))
```

Similar with groups, but these are for structured random effects, like RW1, used by f(. . . , group = g [, ngroup = ng]). Looks quite strange, but for a RW2, it connects a points not only to the 2 next and 2 previous points, but also the five closest points on every other RW2. When our data follows multiple likelihoods, we can structure the response y as a matrix or list, and apply a specific likelihood to each part of the list. Probably different hyperparameteres for each likelihood.

```r
# With Y as a matrix with NA when the row is in another group, so only one
# non NA per row, groups as columns
result = inla ( formula = Y ~ 1 + x ,
          family = c("gaussian","gaussian") ,
          data = list(Y = Y , x = x))
```

Copy allows us to use different elements of the same random effect, for example x_i and x_{i-1} if it doesent fit like an AR1 or RW2 or something predefined. The formula would then be: formula = y ~ f(i, model = "iid") + f(i.plus, copy="i") + . . . when $\eta_i = u_i + u_{i-1}$. I assume the .plus shifts the sequence one to the right or something. Can also scale the copy, like f(i.plus, copy="i", hyper = list(beta=list(fixed=FALSE))).

Can also apply a linear transformation A to η:

$$\eta^* = A\eta \text{ with } y_i \sim \pi(y_i|\eta_i^*, \theta)$$

```r
result = inla(formula, ... , control.predictor = list(A = A))
```

Can sometimes simplify the model and can be interchangeable with the copy feature sometimes. Can also do it as below.

```r
# Alternative construction
A = cbind(rep(1 , n), x)
x1 = c(1, 0) ; x2 = c(0, 1)

# Run model
result = inla ( formula = y ~ x1 + x2 - 1,
          data = list(y = y, x1 = x1, x2 = x2),
          control.predictor = list(A = A))
```

We might improve the model by using a linear combination of the latent field, like $v = Bx$ for some matrix B. Ideally we then use the vector $\hat{x} = [x, c]$, but this gives a much denser (less sparse) precision matrix. Can instead approximate.

```r
n = 100
x = rnorm(n)
z = rnorm(n)
idx = 1:n
eta = 5 + x + z + rnorm(n)
formula = y ~ 1 + x + z + f(idx , model ="iid")
y = rpois(n, lambda = exp(eta))

# Define linear combinations
# Get alpha _x - alpha _z
lc1 = inla.make.lincomb(x =1, z = -1)
names(lc1) = "lc1"

# Get an average over all random effects
lc2 = inla.make.lincomb(idx = rep(1/n , n))
names(lc2)= "lc2"

# Run inla
r = inla(formula , "poisson", data = data.frame (y , x , z , idx ) ,
      lincomb = c( lc1 , lc2 ),
      control.inla = list(lincomb.derived.correlation.matrix = TRUE ))
r$summary.lincomb.derived
```

RW2 with unobserved points in the middle

```r
# Generate data set
y = cumsum(cumsum(rnorm(100, sd =0.01)))+ rnorm (100 , sd = 0.1)
y = y[c(1:30 , 70:100)]

# Make time vector * and * value vector
t = c (1:30 , 70:100)
```

```r
v = 1:100

# Run inla
#values is the time points
result = inla(formula = y ~ f(t, model ="rw2", values =v,
          constr = FALSE ) -1 , data = list (y = y, t = t, v = v))

plot(result)
```

Can also make INLA run on a remote server with some setup, use inla.call = “remote”. Something about submit and retrieve, some sort of saving of the model?

Useful tricks, set verbose = TRUE, will show size, optimizer steps and where it failed. Can also check result$logfile.