Simulation study with logit model

Benjamin Christoffersen

2016-12-24

Intro

This note has four objectives. The first objective is to test how the `ddhazard` fits compare with a Generalized Additive models (GAM) and a "static" logistic model with simulated data. We will look at the following models/estimation methods from `ddhazard` function in the `dynamichazard` package:

- Fits with the Extended Kalman Filters (EKF) with and without extra iterations in the scoring step
- Second order random walks with the EKF estimation method
- Mixture of fixed and time varying effects with the EKF estimation method. Fixed effects are both estimated with the E-step and M-step method described in ddhazard vignette
- Fits using the Unscented Kalman filter (UKF)

The second objective is to show how to estimate various models with the function `ddhazard`. For this reason, the note contains intermediate `R` code which is not needed to understand the simulation results. Thus, we will use * in the headers of section to distinguish the content. The headers marked with no * indicates sections with results of simulation or contains important comments. Headers with an * and ** shows increasingly less important code to understand the simulation. Consequently, you can skip to the headers with no * if you are only interested in the results

The third objective is to illustrate how the various methods performs for out-of-time prediction (forecasting). By out-of-time we mean that we only observe outcomes up to given time, d, and then predict the outcome for future observations at time $d + 1$

The final fourth objective is to show that both the EKF and UKF scales linearly with the number individuals (series)

All method use the logistic link function. We will do three runs of experiments in the following order:

1. A Model where all effects are time varying and we use the correct binning intervals
2. A model where only one parameter is time varying and we use the correct binning intervals
3. A Model where all effects are time varying but we use incorrect binning intervals

where correct or incorrect binning intervals refers to whether or not we bin at the same time where the coefficient are simulated to change. For example, we bin correctly where we simulate the coefficients to change at time $1, 2, \dots, d$ and we estimate the coefficient at time $1, 2, \dots, d$. The models will be compared in terms of Brier score, median absolute residuals and standard deviation of the absolute residuals. All metrics will be reported on out-sample data or out-of-time data. All plots will have true coefficients as continuous lines while dashed lines are estimates

You can install the version of the library used to make this vignettes from github with the `devtools` library as follows:

```
current_version # The string to pass devtools::install_github
```

```
## [1] "boennecd/dynamichazard@b76a4cde4233581cc7b53a843ec76f771440c0ec"
```

```
devtools::install_github(current_version)
```

Moreover, you will also find the source code for the vignette at the github page. The note is not meant to be self contained. It is recommend to see ddhazard vignette for an introduction to the models and methods in the `dynamichazard` package

Notions

For clarity, here is a list of notions used:

- Run: An experiment with one of the three previously specified settings where we make k simulations with n series in each
- Simulation: One simulation within a run with one set of coefficients $\vec{\beta}_0, \dots, \vec{\beta}_d$ and given number n of series
- Series/individuals: A person/individual either making it to the end of the time of the given simulation or dying at some time during the period
- Coefficients: the entries of the vectors $\vec{\beta}_t$ in a given simulation
- Covariates: vectors $\vec{x}_{it}$ for a given individual at a given time in simulation

Findings

The findings are:

- The UKF method seems to perform well for both small and larger number of series
- Taking multiple iterations in the correction step of the EKF seems to be beneficial with moderate amount or large amount of series
- Specifying a fixed effect as time varying or setting the binning number incorrectly has little effect on the results
- The UKF and/or the EKF method with extra iteration in the correction step perform close to a Generalized Additive model in terms of out-sample Brier score

You will see that the the estimation sometimes fails. It is worth stressing that it is my experience that you can always do "trail-and-error" with the initial covariance matrix in the state equation, the covariance matrix at time zero and tuning parameters in order to get a model to fit a given dataset. Of course, it is a disadvantage that any given data set may require some tuning by the user. Although as will be shown, tuning by the user is not often needed with data sets like those presented here

Setup

The following values will be used in the simulation:

```
ns <- c(200, 800, 2000) # Number of series
n_beta <- 5             # Number of covariates
T_max <- 20             # The last time we observe
n_sims <- 100           # Number of simulation in each run

gsub("(^.+)(/dynamichazard.+$)", "...\\2", getwd())
```

```
## [1] ".../dynamichazard/vignettes/Prebuild"
```

```
source("../../R/test_utils.R")
```

`ns` is the number of series (individuals) we will estimate in each of the simulation in each of the runs. Thus, we will perform simulations with a total of 200, 800 and 2000 series in each. Each simulation will have `n_beta` = 5 covariates plus an intercept. Each run will simulate `n_sims` = 100 times. Finally, we source the `test.utils.R` file to define the simulation function. You can find this script on the github site. `T_max` is the number of bins/intervals we observe. Thus, we have 1, 2, ..., `T_max` + 1 covariate vectors (+1 for the time zero coefficient vector)

Fitting true model

We will make runs for various number of individuals in this section where we estimate a model where all effects are time varying and we use the correct binning intervals. Thus, the only models that are miss specified are the model with one time varying effect (which will be `x2`) and the model were we use a second order random walk

`do.call` function

We will use `do.call` in this vignette. To my knowledge, `do.call` is not standard so this section is included to give a brief introduction to `do.call` for users who are not familiar with `do.call`. We can take an example with the `mean` function. We will make the following call where we set `na.rm` to `TRUE`:

```
mean(x = c(1, 2, NA, 6), na.rm = T)
```

```
## [1] 3
```

This call can also be made as follows `do.call`:

```
arg_ex <- list(x = c(1, 2, NA, 6), na.rm = T)
do.call(mean, arg_ex)
```

```
## [1] 3
```

Hence, `do.call` is usefull in situation where we make calls where almost all the arguments are the same. For example, in the setting where we have arguments `a1`, `a2`, `...`, up to `a1000` and we only want to change argument `a101` say. This can then be done as follows:

```
# Not runable
arg_ex <- list(a1 = x,
               a2 = y,
               ..., # enter all the other values
               a1000 = z)
do.call(some_func, arg_ex)

# change only a101 argument and keep the rest as the arguments as is
arg_ex$a101 <- some_specific_value
do.call(some_func, arg_ex)
```

Definition of simulation function

Below, we define a list of `default_args` (default arguments) to our simulation function which we can later use using `do.call`.

```
# Default arguments for simulation
default_args <- list(
  n_vars = n_beta, # Number of betas not including intercept
  beta_start = c(-1, -.5, 0, 1.5, 2), # start value of coeffecients
  intercept_start = -5, # start value of intercept
  sds = c(.1, rep(.5, n_beta)), # std. deviations in state equation
  t_max = T_max, # Largest time we observe
  x_range = 1, # range of covariates
  x_mean = .5, # mean of covariates
  tstart_sampl_func = # we randomly draw the start time of each serie
```

```
    function(...) max(runif(1, min = -10, max = 18), 0)
)
```

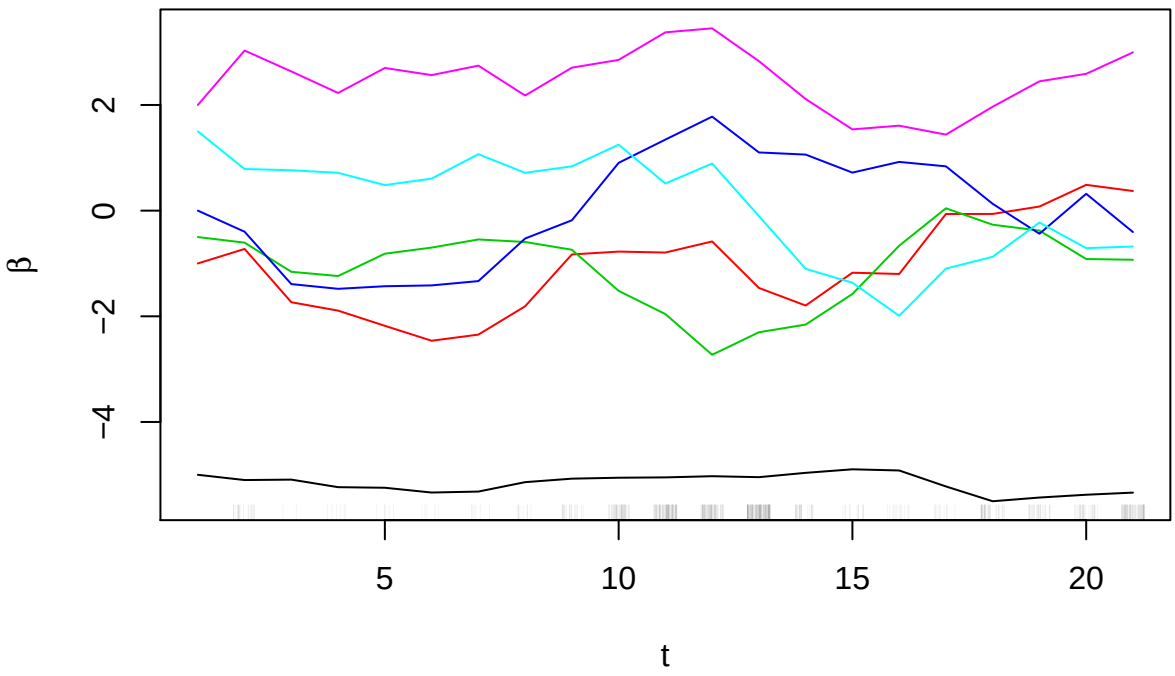
Let $\vec{\beta}_t$ denote the time varying covariates at time t. Then the `beta_start` is the time 0 values of the coefficients and `intercept_start` is the starting value of the intercept. The `sds` are the standard deviations, σ_i, in the state equation. Hence,

$$\beta_{j,t} = \beta_{j,t-1} + \epsilon_{j,t}, \qquad \epsilon_{j,t} \sim N(0, \sigma_j^2)$$

where each margin is independent of the others. The `x_mean` and `x_range` defines how the covariate values are simulated. The above setting implies that $x_{itj} = \text{Unif}(0.5 - 1/2, 0.5 + 1/2)$ where x_{itj} is the i'th individuals covariate j at time t. The covariates vector $\vec{x}_{it}$ is updated at time differences of $1 + \eta$ where $\eta \sim \text{Exp}(1)$ and ηs are drawn separately for each individual. The motivation for this behavior is that we can have different covariate update times than our binning time in a given study. For instance, say we are looking at a medical study and the covariates are laboratory values. The time of laboratory values from an individual's visit the doctor can differ from whatever binning periods we use in the state-space model. Further, the time when laboratory values are updated can differ between patients. One might see his doctor every week or so while another only sees his doctor every year

Below we illustrate how the coefficients vectors from a simulation can look:

```
# We can simulate by
set.seed(51231)
sims <- do.call(test_sim_func_logit, c(list(n_series = max(ns)), default_args))

# This is how the state vectors look
# We define a function so we can re-use it later
plot_func <- function(ylim = c()){ # we define a function here so we can use it later
  matplot(sims$betas, type = "l", lty = 1, ylab = expression(beta), xlab = "t",
          ylim = range(sims$betas, ylim), col = 1:(n_beta + 1))

  # Add rug plot to illustrate when people die
  rug(jitter(sims$res$tstop[sims$res$event==1], amount = .25) + 1,
      col = rgb(0, 0, 0, .05))
}

plot_func()
```



The black line is the intercept while the colored lines are the coefficients for the covariates. The lines on the x-axis illustrate when we observe that individuals die. There is one line for each death. Next, we can look at the number of failures in each simulation:

```
# We get a decent amount of failures and survivers in some of the simulations
# We use do.call to avoid repeating the above argument list
set.seed(468249)
n_fails_in_sim <- rep(NA_real_, 15)
for(i in seq_along(n_fails_in_sim)){
  sims <- do.call(test_sim_func_logit, c(
    default_args, c(list(n_series = max(ns))))) # Take largest amount of series
  n_fails_in_sim[i] <- sum(sims$res$event)
}
n_fails_in_sim # number of failures in each simulation
```

```
##  [1] 1327 1868  500 1014 1249  131 1991  838 1201 1430   74 1171 1311 1180
## [15] 1316
```

* Defintion of fit functions

We will define functions to estimate the different models with a data frame as the first argument where the data frame is from a `test_sim_func_logit` call. This will reduce the amount of code later

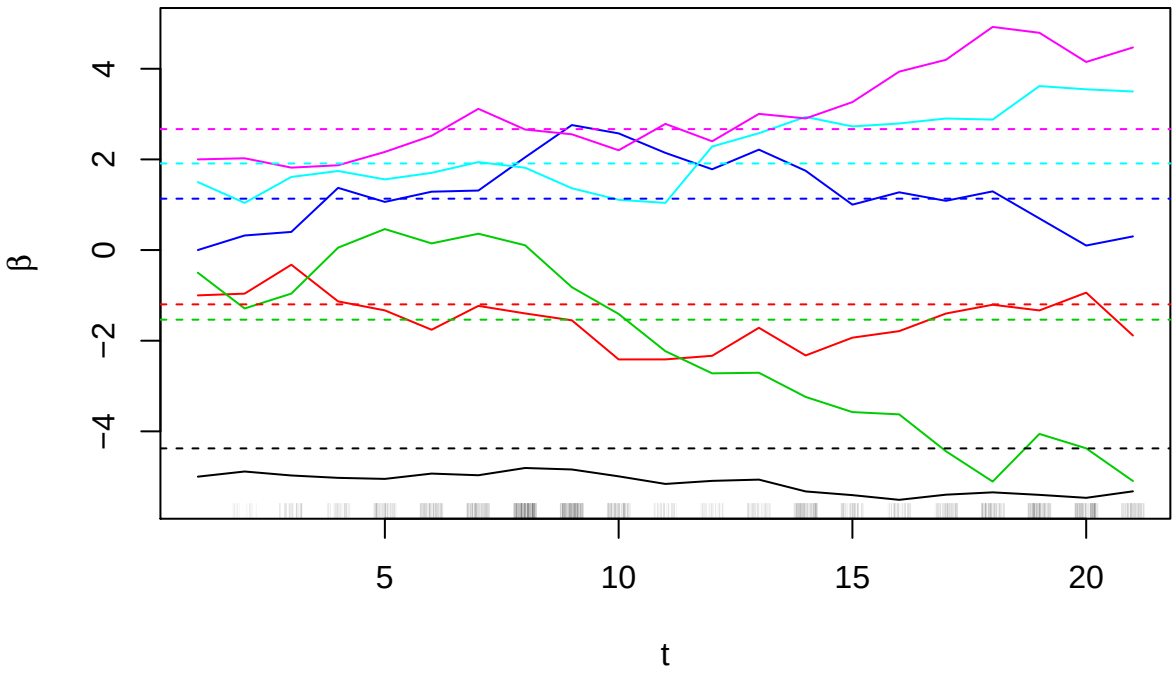
** Defintion of static fit

Below, we define function to fit a model where the coefficients are fixed ($\vec{\beta}_t = \vec{\beta}$). It is estimated using `glm`

```
library(survival); library(dynamichazard)

# Set up function for static fit
fit_funcs = list()
fit_funcs$static <- function(s = sims$res)
  static_glm(formula = Surv(tstart, tstop, event) ~ x1 + x2 + x3 + x4 + x5,
             data = s, max_T = T_max, by = 1, id = s$id)

fit <- fit_funcs$static()
class(fit) # returns a glm object
```

```
## [1] "glm" "lm"
```

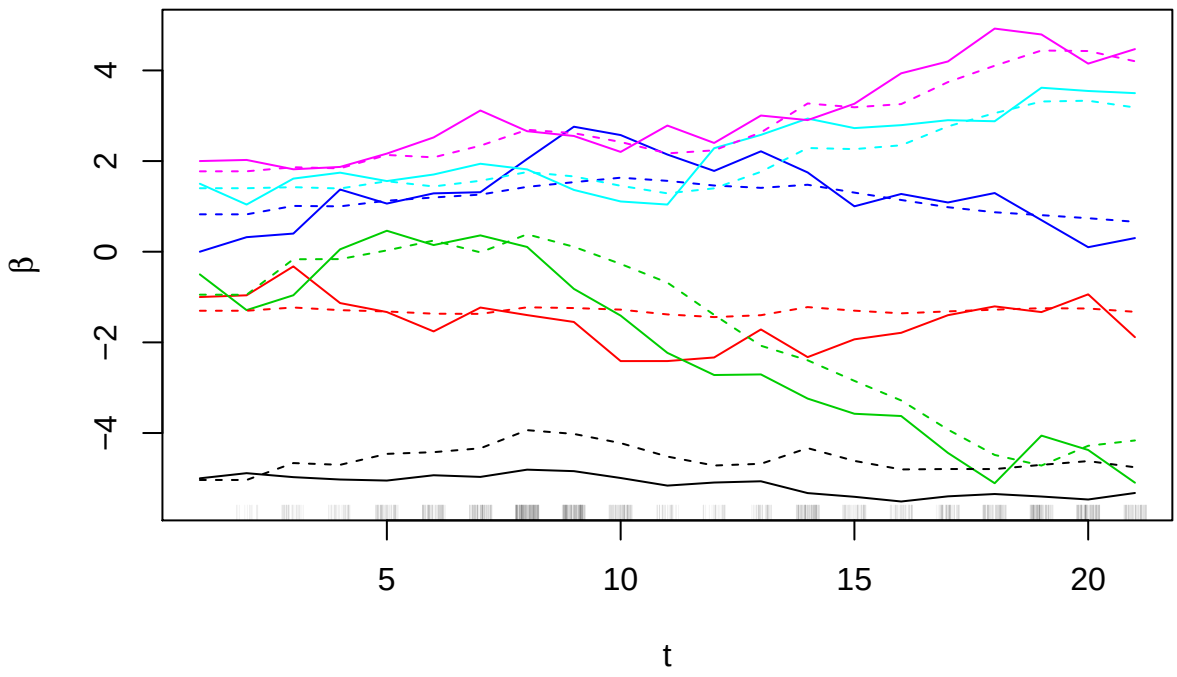
```
# Estimates seems plausible
plot_func(ylim = fit$coefficients)
abline(h = fit$coefficients, col = 1:(n_beta + 1), lty = 2)
```



* Defintion of `ddhazard` fit functions

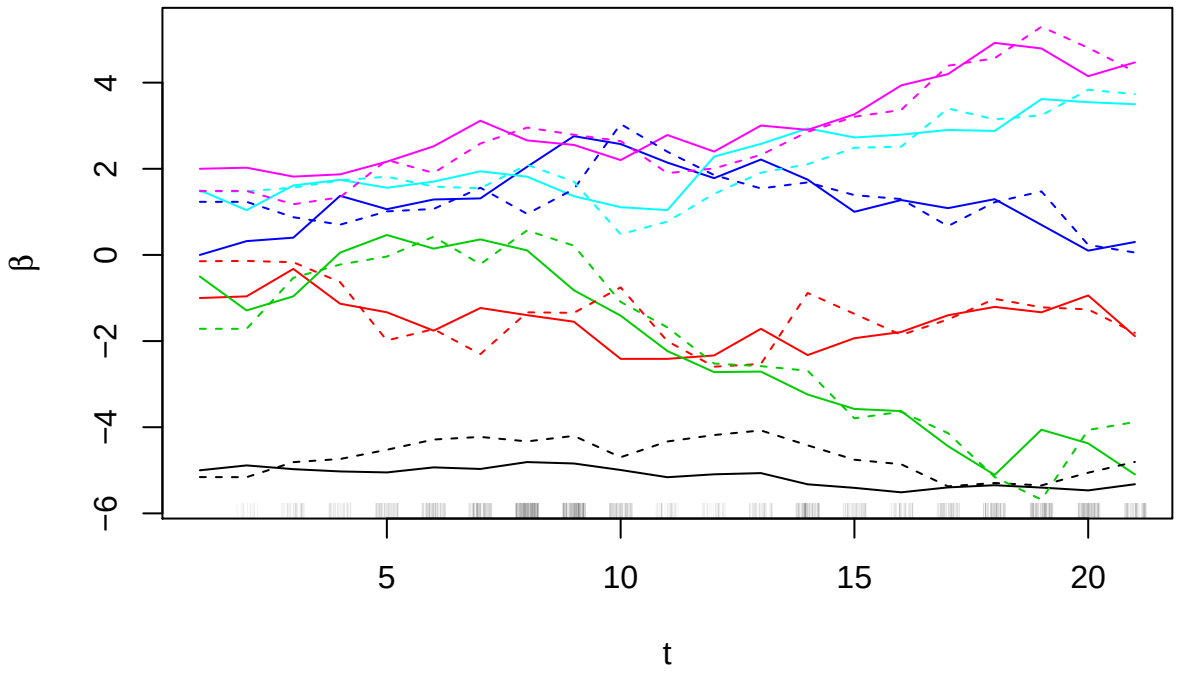
Below, we define a function to fit a first order random walk model with a given learning rate and potential extra iterations in the scoring step (see the ddhazard vignette for details):

```r
library(survival); library(dynamichazard)

# Set up function ddhazard fit function for convenience
# LR:       learning rate in correction step
# NR_eps:   tolerance in correction step. NULL yields no extra iterations
fit_funcs$dd <- function(s = sims$res, LR = 1, NR_eps = NULL)
  tryCatch({
    ddhazard(
      formula = Surv(tstart, tstop, event) ~ x1 + x2 + x3 + x4 + x5,
      data = s, max_T = T_max, by = 1, id = s$id,
      Q_0 = diag(10, n_beta + 1), Q = diag(.01, n_beta + 1),
      control = list(LR = LR, NR_eps = NR_eps, eps = 0.01))
  }, error = function(...) NA) # Return NA if fails

fit <- fit_funcs$dd()

# Plot estimates and actual coffecients
plot_func(ylim = fit$state_vecs)
matplot(fit$state_vecs, col = 1:(n_beta + 1), lty = 2,
        type = "l", add = T)
```



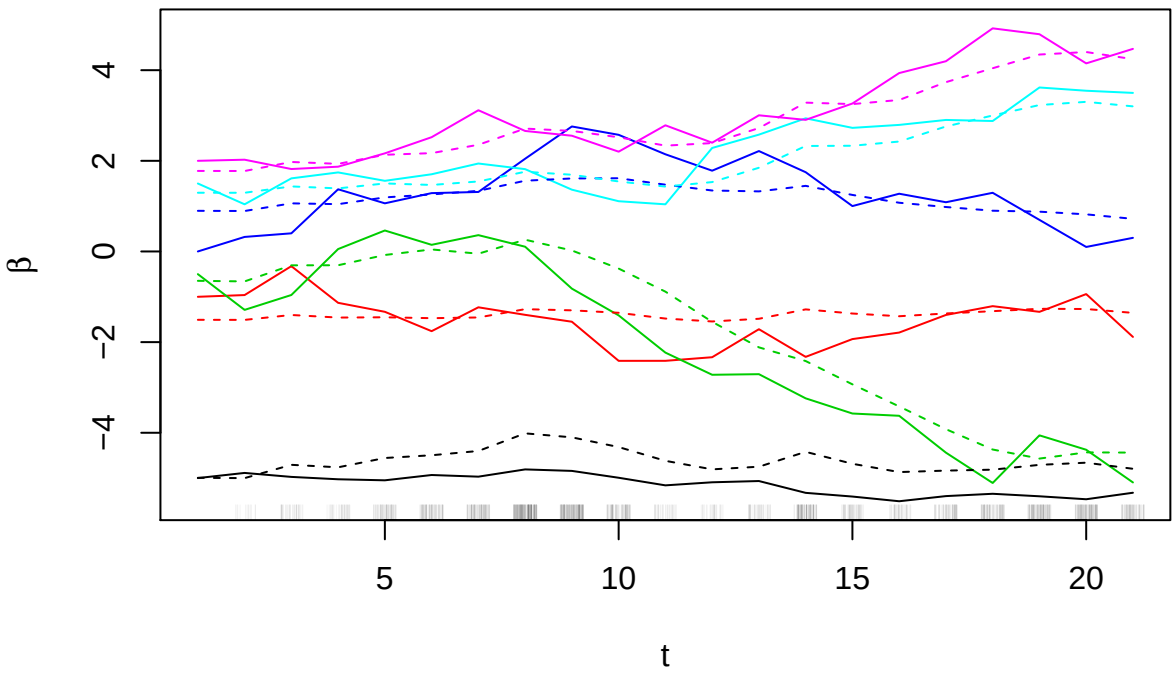

```r
# Same call with extra iterations
fit <- fit_funcs$dd(LR = .5, NR_eps = .01)

# Look at new plot
plot_func(ylim = fit$state_vecs)
```

```
matplot(fit$state_vecs, col = 1:(n_beta + 1), lty = 2,
        type = "l", add = T)
```



Below, we define a function to fit a first order random walk model with the UKF method:

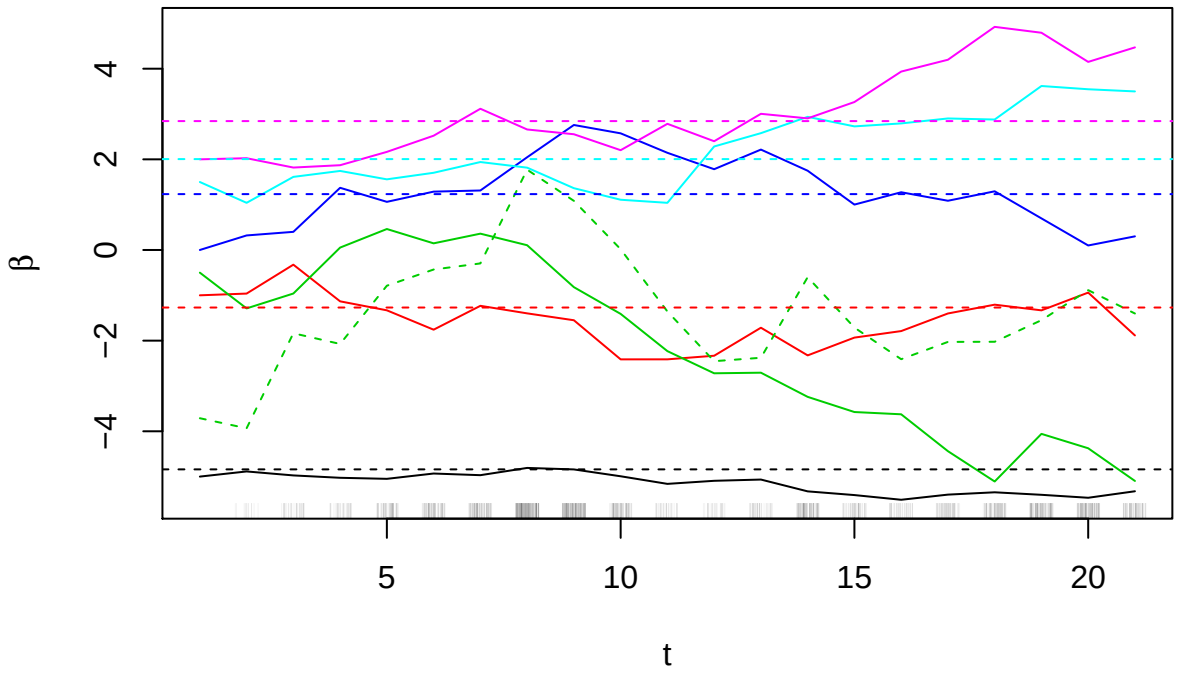
```
# Fitting with UKF
fit_funcs$dd_UKF <- function(s = sims$res, alpha = 1, beta = 0){
  tryCatch({
    ddhazard(
      formula = Surv(tstart, tstop, event) ~ x1 + x2 + x3 + x4 + x5,
      data = s, max_T = T_max, by = 1, id = s$id,
      Q_0 = diag(1, n_beta + 1), Q = diag(.01, n_beta + 1),
      control = list(
        eps = 0.1,
        alpha = alpha,       # Set tuning parameter
        beta = beta,         # Set tuning parameter
        method = "UKF"))     # Set estimation method (EKF is default)
  }, error = function(...) NA) # Return NA if fails
}

fit <- fit_funcs$dd_UKF()

# Look at new plot
plot_func(ylim = fit$state_vecs)
matplot(fit$state_vecs, col = 1:(n_beta + 1), lty = 2,
        type = "l", add = T)
```



Below, we define a function to estimate a first order random walk model where only one parameter (`x2`) is time varying:

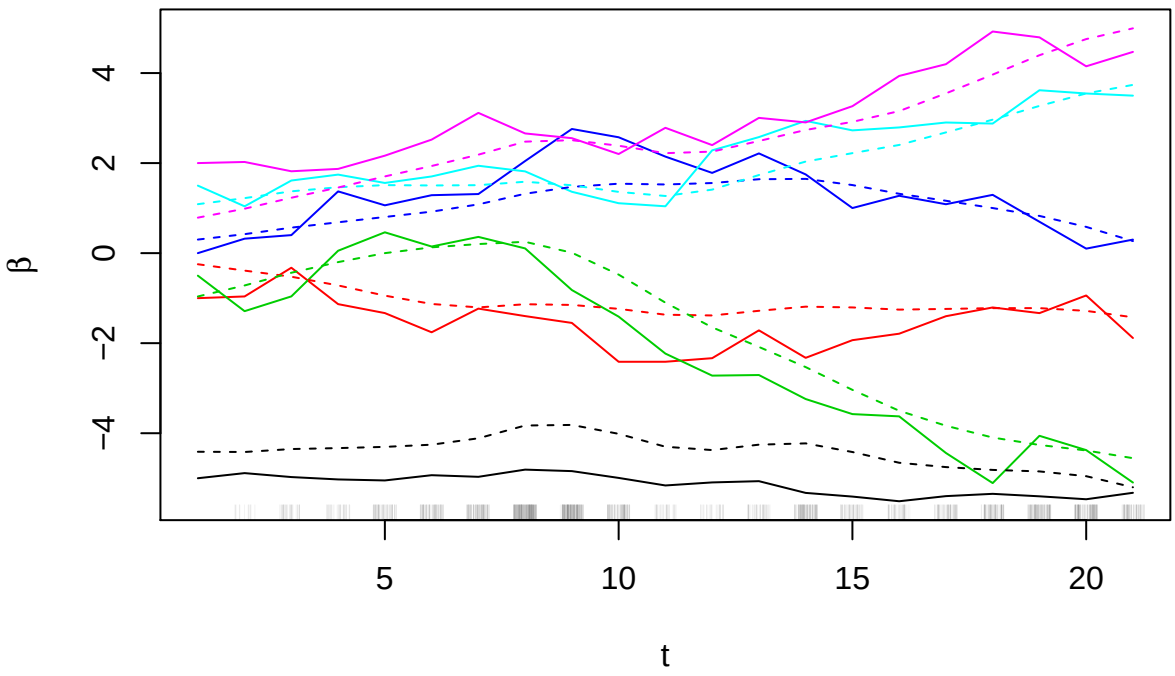
```
# Fitting with fixed effects
fit_funcs$dd_fixed <- function(
  s = sims$res, LR = 1, NR_eps = NULL,
  fixed_terms_method = "M_step"){ # The method to use to estimate the fixed
                                  # fixed effects
  tryCatch({
    ddhazard(
      formula = Surv(tstart, tstop, event) ~
        ddFixed(1) +                              # Fix intercept
        ddFixed(x1) + x2 +                        # Note x2 is time varying
        ddFixed(x3) + ddFixed(x4) + ddFixed(x5),
      data = s, max_T = T_max, by = 1, id = s$id,
      Q_0 = diag(1, 1), Q = diag(.01, 1),
      control = list(LR = LR, NR_eps = NR_eps, eps = 0.1,
                     fixed_terms_method = fixed_terms_method))
  }, error = function(...) NA) # Return NA if fails
}

fit <- fit_funcs$dd_fixed()

# Look at new plot
plot_func(ylim = range(fit$state_vecs, fit$fixed_effects))
matplot(fit$state_vecs, col = 3, lty = 2,
        type = "l", add = T)
```

```
abline(h = fit$fixed_effects, col = c(1:2, 4:6), lty = 2)
```



Next, we define a function to fit the model with a second order random walk:

```
# Fitting with second order
fit_funcs$dd_2_order <- function(s = sims$res, LR = 1, NR_eps = NULL){
  tryCatch({
    ddhazard(
      formula = Surv(tstart, tstop, event) ~ x1 + x2 + x3 + x4 + x5,
      data = s, max_T = T_max, by = 1, id = s$id,
      # Q_0 and Q needs more elements
      Q_0 = diag(c(rep(1, n_beta + 1), rep(0.5, n_beta + 1))),
      Q = diag(c(rep(.01, n_beta + 1))),
      order = 2, # specify the order
      control = list(LR = LR, NR_eps = NR_eps, eps = 0.1))
  }, error = function(...) NA) # Return NA if fails
}

fit <- fit_funcs$dd_2_order()

# Look at new plot
plot_func(ylim = fit$state_vecs)
matplot(fit$state_vecs[, 1:6], col = 1:(n_beta + 1), lty = 2,
        type = "l", add = T)
```

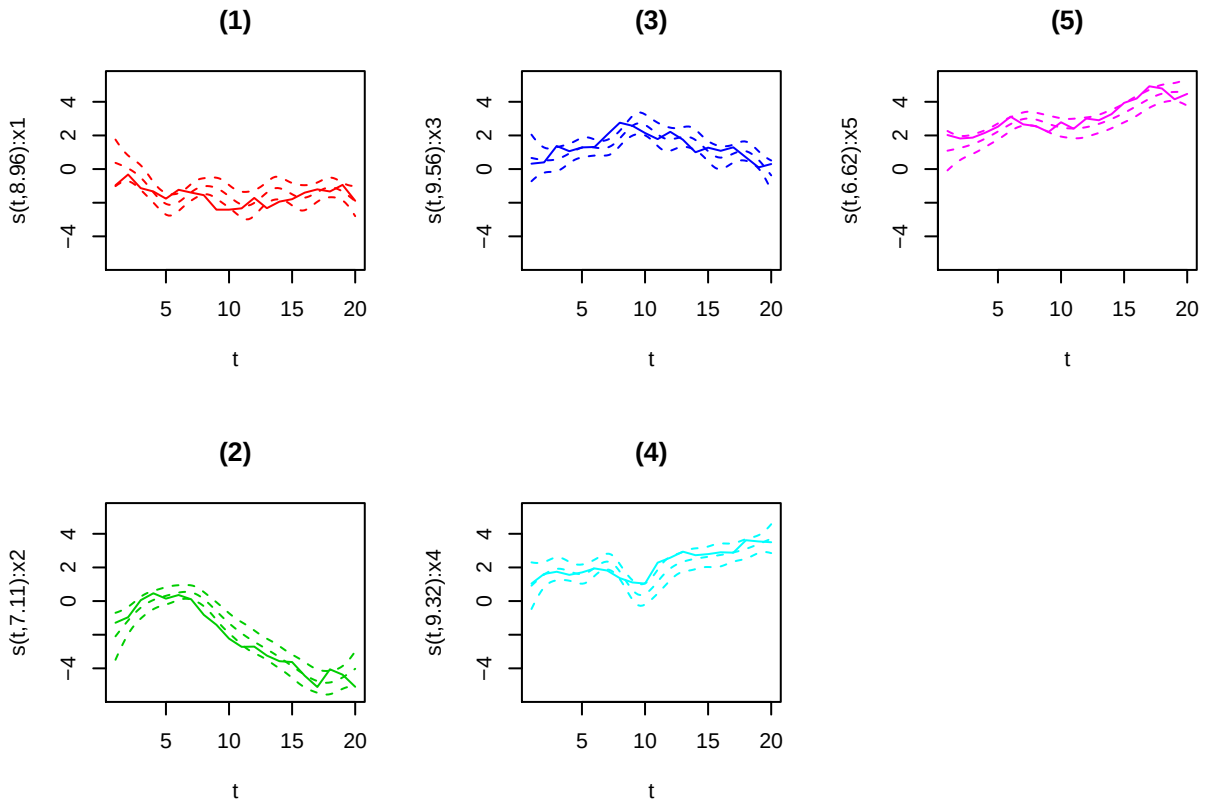


** Defintion of GAM fit function

We define the estimation method for the Generalized additive model in the next code snippet. We use `bam` function from the `mgcv` package which corresponds to `gam` but for very large datasets

```
library(mgcv)
fit_funcs$gam <- function(s = sims$res){
  # get data frame for fitting
  dat_frame <- get_survival_case_weigths_and_data(
    formula = Surv(tstart, tstop, event) ~ x1 + x2 + x3 + x4 + x5,
                data = s, max_T = T_max, by = 1, id = s$id, use_weights = F)$X
  # fit model
  bam(
    formula = Y ~
      # cr is cubic basis with k knots
      s(t, bs = "cr", k = 10, by = x1) +
      s(t, bs = "cr", k = 10, by = x2) +
      s(t, bs = "cr", k = 10, by = x3) +
      s(t, bs = "cr", k = 10, by = x4) +
      s(t, bs = "cr", k = 10, by = x5),
  family = binomial, data = dat_frame,
  method = "GCV.Cp",
  control =
    gam.control(nthreads = parallel::detectCores() - 1)) # Use parallel
}
```

```r
# fit model
fit <- fit_funcs$gam()

# Compare plot
layout(matrix(1:6, nrow = 2))
for(i in 1:n_beta){
  plot(fit, pages = 0, rug = F, col = i + 1, select = i, lty = 2,
       main = paste0("(", i, ")"))
  lines(sims$betas[-1, i + 1], col = i + 1)
}
```



** Definition of prediction functions

The following code snippets define predictions methods for each of the estimation methods. We start off by defining a split function such that we can sample individuals (series) into a test set and a training test:

```r
split_func <- function(s = sims$res){
  # Sample ids
  test_ids <- sample(
    unique(s$id), floor(length(unique(s$id)) / 2), replace = F)

  # Return seperate data frames
  return(list(test_dat = s[s$id %in% test_ids, ],
              fit_dat = s[!s$id %in% test_ids, ]))
}
```

```r
# Illustrate use
tmp <- split_func()
# No ids intersect in the two sets
length(intersect(tmp$test_dat$id, tmp$fit_dat$id))
```

```
## [1] 0
```

```r
# The union is exactly the number of ids we simulated
length(union(tmp$test_dat$id, tmp$fit_dat$id))
```

```
## [1] 2000
```

Having defined the splitting method, we turn to the prediction functions. The idea is to define the `brier_funcs$general` function which takes in a prediction function, a fit and a data frame. Next, we then define individual prediction functions for each of the models which will be passed to `brier_funcs$general`:

```r
# Define general prediction function
brier_funcs <- list()
brier_funcs$general <- function(brier_func, fit, eval_data_frame){
  d_frame <- get_survival_case_weigths_and_data(
    formula = Surv(tstart, tstop, event) ~ x1 + x2 + x3 + x4 + x5,
    data = eval_data_frame, max_T = T_max, by = 1, id = eval_data_frame$id,
    use_weights = F)$X

  # Change start and stop times
  d_frame$tstart <- d_frame$t - 1
  d_frame$tstop <- d_frame$t

  # Compute residuals
  resids <- brier_func(fit, d_frame)

  # Return estimates
  list(brier = mean(resids^2),
       median_abs_res = median(abs(resids)),
       sd_res = sd(resids))
}

# Prediction method for static model
brier_funcs$static <- function(fit, d_frame){
  preds <- predict(fit, newdata = d_frame, type = "response")
  return(d_frame$Y - preds)
}

# Test function
fit <- fit_funcs$static(tmp$fit_dat)
unlist(
  brier_funcs$general(brier_funcs$static, fit, tmp$fit_dat)[
    c("brier", "median_abs_res", "sd_res")]) # in sample stats
```

```
##          brier median_abs_res         sd_res
##        0.06949        0.05417        0.26363
```

```r
unlist(
  brier_funcs$general(brier_funcs$static, fit, tmp$test_dat)[
    c("brier", "median_abs_res", "sd_res")]) # out sample stats
```

```
##          brier median_abs_res         sd_res
##        0.06666        0.05439        0.25814
```

```
# Define prediction function for ddhazard model
brier_funcs$dd <- function(fit, d_frame){
  preds <- predict(fit, new_data = d_frame, tstart = "tstart", tstop = "tstop")
  return(d_frame$Y - preds$fits)
}

fit <- fit_funcs$dd(tmp$fit_dat)

unlist(
  brier_funcs$general(brier_funcs$dd, fit, tmp$fit_dat)[
    c("brier", "median_abs_res", "sd_res")]) # in sample stats
```

```
##          brier median_abs_res         sd_res
##        0.06362        0.05211        0.25203
```

```
unlist(
  brier_funcs$general(brier_funcs$dd, fit, tmp$test_dat)[
    c("brier", "median_abs_res", "sd_res")]) # out sample stats
```

```
##          brier median_abs_res         sd_res
##        0.06116        0.05186        0.24674
```

```
# Define prediction function for gam model
brier_funcs$gam <- function(fit, d_frame){
  preds <- predict(fit, newdata = d_frame, type = "response")
  return(d_frame$Y - preds)
}

fit <- fit_funcs$gam(tmp$fit_dat)
unlist(
  brier_funcs$general(brier_funcs$gam, fit, tmp$fit_dat)[
    c("brier", "median_abs_res", "sd_res")]) # in sample stats
```

```
##          brier median_abs_res         sd_res
##        0.06310        0.04101        0.25122
```

```
unlist(
  brier_funcs$general(brier_funcs$gam, fit, tmp$test_dat)[
    c("brier", "median_abs_res", "sd_res")]) # out sample stats
```

```
##          brier median_abs_res         sd_res
##        0.06068        0.04148        0.24627
```

** Definition of multiple simulations function

To make things easier, we define a function that takes in a function to simulate from. Given a function to simulate with, the new function perform `n_sims` = 100 simulations for each of values `ns` (200, 800 and 2000):

```
simulate_n_print_res <- function(
  sim_func, # Function that takes one argment which is number of series
  NR_eps = c(.01, NA)) # Tolerance in scoring step
  {
  for(n in ns){
    out <- array(NA_real_, dim = c(n_sims, 8, 3),
```

```
              dimnames = list(
                NULL,
                c("static", "Extra correction", "Single correction",
                  "2 order EKF", "Fixed E-step", "Fixed M-step", "UKF", "gam"),
                c("Brier", "Median abs res", "sd res")))

n_failures_and_surviers <- array(
  NA_integer_, dim = c(2, n_sims),
  dimnames = list(c("# failures", "# survivers"), NULL))

#*******
# Progress bar for inpatient people (me)
pb <- tcltk::tkProgressBar(paste("Estimating with n =", n), "",
                           0, n_sims, 50)
#*******

for(i in 1:n_sims){
  #*******
  info <- sprintf("%.2f%% done", 100 * (i - 1) / n_sims)
  tcltk::setTkProgressBar(pb, i - 1, paste("Estimating with n =", n), info)
  #*******

  # Sample until we get an outcome have sufficient amount of deaths and
  # survivers
  repeat{
    sims <- sim_func(n)

    # We want some survivers and some deaths
    if(sum(sims$res$event) > 25 && n - sum(sims$res$event) > 25)
      break
  }

  n_failures_and_surviers["# failures", i] <- sum(sims$res$event)
  n_failures_and_surviers["# survivers", i] <- n - sum(sims$res$event)

  # Split data
  sim_split <- split_func(sims$res)

  # Fit static model
  static_fit <- fit_funcs$static(sim_split$fit_dat)

  # Fit dd model
  dd_fits <- list(rep(NA, length(NR_eps)))
  for(k in seq_along(NR_eps)){
    dd_fits[[k]] <- fit_funcs$dd(
      sim_split$fit_dat,
      NR_eps = if(is.na(NR_eps[k])) NULL else NR_eps[k])
  }

  # Fit second order
  dd_2_order <- fit_funcs$dd_2_order(sim_split$fit_dat)

  # Fit fixed effect
```

```
    dd_fixed_E_step <- fit_funcs$dd_fixed(sim_split$fit_dat,
                                          fixed_terms_method = "E_step")
    dd_fixed_M_step <- fit_funcs$dd_fixed(sim_split$fit_dat,
                                          fixed_terms_method = "M_step")

    # UKF fit
    dd_UKF <- fit_funcs$dd_UKF(sim_split$fit_dat)

    # Fit gam model
    gam_fit <- fit_funcs$gam(sim_split$fit_dat)

    # Evalute on test data
    models <- c(list(static_fit), dd_fits,
                list(dd_2_order, dd_fixed_E_step, dd_fixed_M_step,
                     dd_UKF, gam_fit))

    eval_funcs = c(brier_funcs$static,
                   replicate(length(dd_fits) + 4, brier_funcs$dd),
                   brier_funcs$gam)

    for(j in seq_along(models)){
      if(length(models[[j]]) == 1 && is.na(models[[j]]))
        next # We have to skip model fits that failed

      metrics <- brier_funcs$general(
        eval_funcs[[j]], models[[j]], sim_split$test_dat)
      out[i, j, "Brier"] <- metrics$brier
      out[i, j, "Median abs res"] <- metrics$median_abs_res
      out[i, j, "sd res"] <- metrics$sd_res
    }
  }

  #*******
  close(pb)
  #*******

  # Print results
  did_fit <- apply(out[, , 1], 2, function(x) n_sims - sum(is.na(x)))
  n_cases_all_success <- sum(complete.cases(out[, , 1]))
  metric_where_all_fit <-
    t(apply(out[complete.cases(out[, , 1]), , , drop = F], 3, colMeans))

  metric_where_all_fit <- formatC(metric_where_all_fit ,format="f", digits=3)
  n_cases_all_success <- formatC(n_cases_all_success, format="d")

  print(knitr::kable(cbind(
    t(metric_where_all_fit), "# succesful fits" = did_fit),
        caption = paste(
  "Mean of metrics with", n/2, "series in test and fit data. Only simulations that succeeds for all s
  align = "r"))
  cat("\n")

  # # Prints the metrics for all the simulation that succeeds for given setup
```

```
    # # Out commented as the metrics are comparable. Download the code and
    # # comment back if you are interested
    # print(knitr::kable(t(apply(out, 3, colMeans, na.rm = T)), digits = 3,
    #                    caption = paste(
    #   "Mean of metrics with", n/2, "series in test and fit data. All simulations for each setup where
    # cat("\n")
  }
}
```

Simulating

We are now able to simulate from the model where all effects are time varying and we use the correct binning intervals with the code below:

```
set.seed(1243)
# Use simulation function
simulate_n_print_res(
  sim_func = function(n)
    do.call(test_sim_func_logit, c(default_args, c(list(n_series = n)))))
```

Table 1: Mean of metrics with 100 series in test and fit data. Only simulations that succeeds for all setups are included. There are 82 of these simulations. The last line shows the number of successful fits for each setup

	Brier	Median abs res	sd res	# succesful fits
static	0.047	0.039	0.208	100
Extra correction	0.100	0.058	0.217	82
Single correction	0.045	0.041	0.203	100
2 order EKF	0.048	0.041	0.207	100
Fixed E-step	0.047	0.050	0.207	100
Fixed M-step	0.046	0.035	0.206	100
UKF	0.045	0.035	0.203	100
gam	0.045	0.025	0.204	100

Table 2: Mean of metrics with 400 series in test and fit data. Only simulations that succeeds for all setups are included. There are 93 of these simulations. The last line shows the number of successful fits for each setup

	Brier	Median abs res	sd res	# succesful fits
static	0.048	0.041	0.202	100
Extra correction	0.063	0.042	0.198	93
Single correction	0.043	0.035	0.193	100
2 order EKF	0.043	0.039	0.193	100
Fixed E-step	0.045	0.043	0.197	100
Fixed M-step	0.045	0.033	0.197	100
UKF	0.043	0.033	0.193	100
gam	0.043	0.027	0.193	100

Table 3: Mean of metrics with 1000 series in test and fit data. Only simulations that succeeds for all setups are included. There are 96 of these simulations. The last line shows the number of successful fits for each setup

	Brier	Median abs res	sd res	# succesful fits
static	0.059	0.056	0.224	100
Extra correction	0.051	0.034	0.211	97
Single correction	0.052	0.043	0.212	100
2 order EKF	0.051	0.047	0.210	100
Fixed E-step	0.054	0.053	0.215	100
Fixed M-step	0.054	0.042	0.216	99
UKF	0.051	0.041	0.210	100
gam	0.051	0.036	0.210	100

We should only compare across methods with mean metrics `where all succeeded to fit`. The logic being that those where the `ddhazard` method fail may be have different errors than those where all succeed to fit.

Conclussion on run

All models perform better than the static model apart from the case where the number of individuals in the training data is only 100. The miss specified models (`2 order EKF` and `Fixed ...`) tend to perform worse than the others models. Though, they still perform better than the static model labeled `static`

The UKF performs close to the gam model when the training data has less than 1000 observations while taking extra iterations in the EKF seems to be worth it in terms of out-sample Brier score when the training data has 1000 observations. This may suggest that the UKF method is better for smaller data sets while the EKF with extra iterations in the scoring step is better suited for larger data sets. In all cases, at least one of models is close to the `gam` model in terms of out-sample Brier score

Single time varying parameter

In this part, we will look at the performs when only singe coefficient (`x2`) varies. Thus, we can see if the models where only (`x2`) is modeled as varying performs performs better

** Definition of simulation function

We start by defining the simulation function. The main change here is that we only set a single standard deviation and that we set it larger than before:

```
# Use simulation function
set.seed(9999)
sim_one_varying <- function(n){
  test_sim_func_logit(
    n_series = n,
    sds = c(sqrt(3)), # Large variance
    is_fixed = c(1:2, 4:6), # All but param three is fixed

    # Same values as before
    n_vars = n_beta,
```

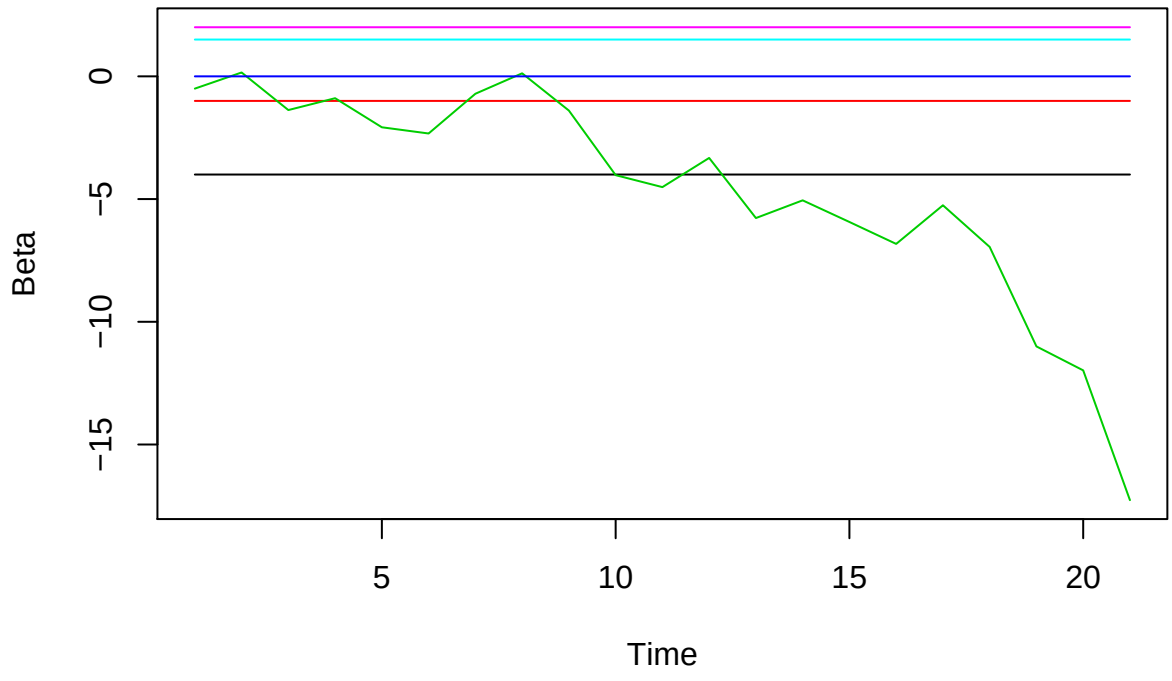
```
    beta_start = c(-1, -.5, 0, 1.5, 2),
    intercept_start = -4,
    t_max = T_max,
    x_range = 1,
    x_mean = .5)
}
```

* Illustration of single simulation

```
# We get a more variable number of failures and survivers (we simulate 200
# series)
replicate(10, sum(sim_one_varying(200)$res$event)) # print number of failures
```

```
##  [1] 199 200 123 197 156  64 200  69 200  82
```

```
# Here is an example of a series
tmp <- sim_one_varying(200)
matplot(tmp$betas, type = "l", lty = 1, ylab = "Beta", xlab = "Time")
```



Simulating

We can simulate with the following call:

```
# Use simulation function
set.seed(8080)
simulate_n_print_res(sim_func = sim_one_varying)
```

Table 4: Mean of metrics with 100 series in test and fit data. Only simulations that succeeds for all setups are included. There are 75 of these simulations. The last line shows the number of successful fits for each setup

	Brier	Median abs res	sd res	# succesful fits
static	0.039	0.030	0.190	100
Extra correction	0.074	0.050	0.210	75
Single correction	0.039	0.030	0.189	100
2 order EKF	0.039	0.038	0.190	100
Fixed E-step	0.039	0.036	0.189	100
Fixed M-step	0.038	0.027	0.187	100
UKF	0.038	0.028	0.187	100
gam	0.039	0.019	0.190	100

Table 5: Mean of metrics with 400 series in test and fit data. Only simulations that succeeds for all setups are included. There are 81 of these simulations. The last line shows the number of successful fits for each setup

	Brier	Median abs res	sd res	# succesful fits
static	0.053	0.052	0.217	100
Extra correction	0.062	0.045	0.214	81
Single correction	0.051	0.044	0.213	100
2 order EKF	0.050	0.049	0.211	100
Fixed E-step	0.050	0.048	0.212	100
Fixed M-step	0.051	0.036	0.212	100
UKF	0.050	0.043	0.211	100
gam	0.049	0.038	0.210	100

Table 6: Mean of metrics with 1000 series in test and fit data. Only simulations that succeeds for all setups are included. There are 96 of these simulations. The last line shows the number of successful fits for each setup

	Brier	Median abs res	sd res	# succesful fits
static	0.062	0.062	0.240	100
Extra correction	0.056	0.042	0.228	96
Single correction	0.059	0.051	0.233	100
2 order EKF	0.057	0.055	0.230	100
Fixed E-step	0.058	0.052	0.232	100
Fixed M-step	0.058	0.040	0.231	100
UKF	0.057	0.045	0.230	100
gam	0.056	0.044	0.228	100

Conclussion on run

The main interest here is how the models labeled `Fixed ...` roughly as good as the other fits. It seems to make a minor difference in terms of out-sample Brier score for all the settings for specify the coefficients as time varying. This may suggest that specifying an effects as time varying does not affect the result

Incorrect binning time

Now, what happens if we get the binning wrong? This is the next experiment we will perform. Specifically, we will set the binning length to `0.1` instead `1` when we simulate. Thus, coefficients are updated at time $0, 0.1, 0.2, \dots$ and whether an individual dies is evaluated at the same times when we simulate. However, the fitted model will still be based on bins of length 1

** Definition of simulation function

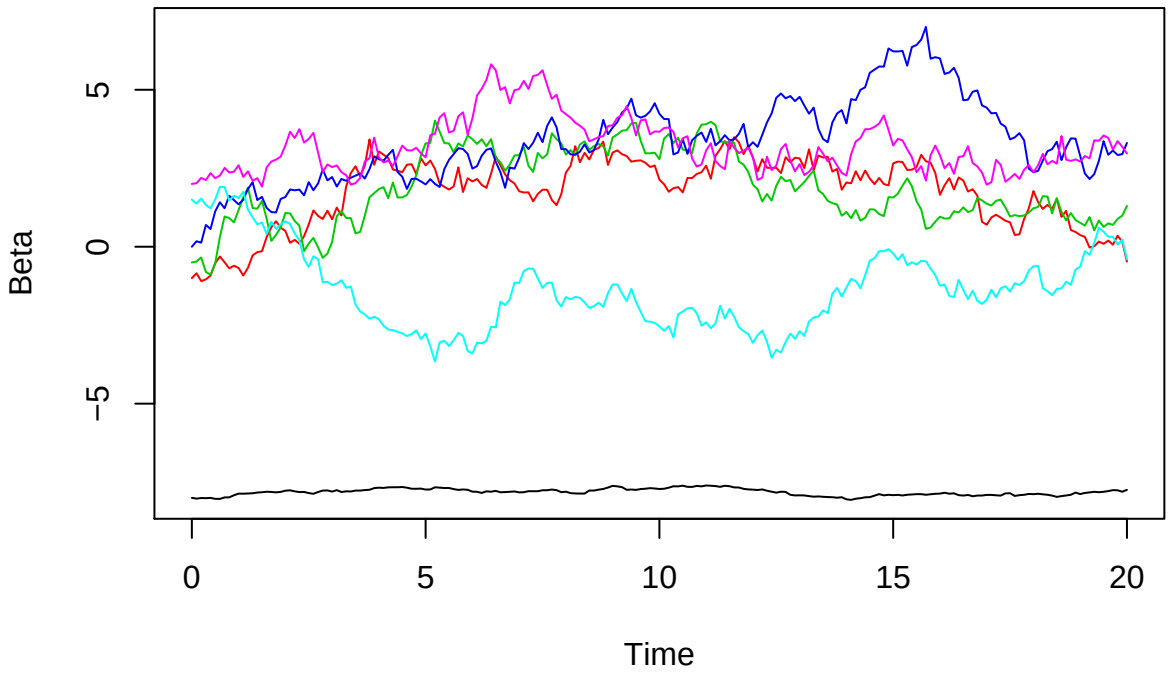
```
set.seed(9001)
sim_finer_binning <- function(n){
  time_denom = 10 # how much finer do we want to bin?

  res <- test_sim_func_logit(
    n_series = n,

    # We multiply through appropiately
    beta_start = c(-1, -.5, 0, 1.5, 2),
    intercept_start = - 8, # Note, we changed the intercept
    sds = c(.1, rep(1, n_beta)) / sqrt(time_denom),
    t_max = T_max * time_denom,
    lambda = 1 / time_denom, # note we change the time when covariates are
                             # updated (the lambda parem in the rate ~ exp(.)
                             # in the time increaments)

    n_vars = n_beta,
    x_range = 1,
    x_mean = .5)

  # Change time denominator
  res$res$tstart <- res$res$tstart / time_denom
  res$res$tstop <- res$res$tstop / time_denom

  res
}
```

* Illustration of single simulation

```
# We get more variable outcomes (we simulate 200 series)
replicate(10, sum(sim_finer_binning(200)$res$event)) # Number of failures
```

```
##  [1] 189 142   8 200 164 103 193  34 105 104
```

```r
# Here is an example of the series
tmp <- sim_finer_binning(200)
matplot((1:nrow(tmp$betas) - 1) / 10,
        tmp$betas, type = "l", lty = 1, ylab = "Beta", xlab = "Time")
```



Simulating

We are now able to simulate with the following call:

```r
# Use simulation function
set.seed(747)
simulate_n_print_res(sim_func = sim_finer_binning)
```

Table 7: Mean of metrics with 100 series in test and fit data. Only simulations that succeeds for all setups are included. There are 86 of these simulations. The last line shows the number of successful fits for each setup

	Brier	Median abs res	sd res	# succesful fits
static	0.035	0.031	0.179	100
Extra correction	0.087	0.065	0.194	86
Single correction	0.038	0.024	0.180	100
2 order EKF	0.033	0.026	0.174	100
Fixed E-step	0.035	0.038	0.177	100
Fixed M-step	0.035	0.020	0.178	100

	Brier	Median abs res	sd res	# succesful fits
UKF	0.033	0.023	0.172	100
gam	0.033	0.012	0.173	100

Table 8: Mean of metrics with 400 series in test and fit data. Only simulations that succeeds for all setups are included. There are 76 of these simulations. The last line shows the number of successful fits for each setup

	Brier	Median abs res	sd res	# succesful fits
static	0.034	0.034	0.163	100
Extra correction	0.048	0.025	0.164	85
Single correction	0.037	0.025	0.161	98
2 order EKF	0.031	0.025	0.156	100
Fixed E-step	0.033	0.033	0.159	93
Fixed M-step	0.034	0.016	0.163	98
UKF	0.032	0.019	0.157	100
gam	0.030	0.017	0.154	100

Table 9: Mean of metrics with 1000 series in test and fit data. Only simulations that succeeds for all setups are included. There are 89 of these simulations. The last line shows the number of successful fits for each setup

	Brier	Median abs res	sd res	# succesful fits
static	0.041	0.042	0.185	100
Extra correction	0.036	0.021	0.173	91
Single correction	0.039	0.031	0.181	100
2 order EKF	0.040	0.035	0.180	100
Fixed E-step	0.040	0.039	0.181	99
Fixed M-step	0.041	0.023	0.186	100
UKF	0.038	0.025	0.177	99
gam	0.036	0.023	0.173	100

Conclussion on run

The UKF seems to perform well in all settings. Moreover, the extra iteration seems to be worth it when there are a moderate amount of observations. The miss specified `Fixed effect` seems to perform worse than the other fit/estimates. Finally, the mean Brier score is again similar to the `gam` fit for the model/method with the best result

Out-of-time prediction

We will investigate how the different estimation method performs when the following period have to be predicted in the following paragraphs. Thus, we cannot use the GAM model because it uses in-sample splines. Though, we can still use the state-space models as we can predict the following state vector given the previous. Further, we can use the static model to compare with

** Define simulation and data splitting function

We start by defining a simulation function and a function to split the data into the first time period which we will use for estimation and the later time period which we will use for the test

```
# Define simulation function
out_sample_args <- default_args
out_sample_args$t_max <- 21

sim_func <- function(n_series = 200)
  do.call(test_sim_func_logit, c(list(n_series = n_series), out_sample_args))

# Define split function
split_data_func <- function(d_frame, split_time = 20){
  # Find data before split_time and set event flag and stop time
  in_sample <- d_frame[d_frame$tstart < split_time, ]
  in_sample$event <- in_sample$event & in_sample$tstop <= split_time
  in_sample$tstop <- pmin(in_sample$tstop, split_time)

  # Find data that ends after split_time and set start time
  out_sample <- d_frame[split_time < d_frame$tstop, ]
  out_sample$tstart <- pmax(out_sample$tstart, split_time)

  # Return
  list(in_sample = in_sample, out_sample = out_sample)
}
```

We extend the period (`t_max`) by one which is the only difference in the simulation. Notice that individuals can be in both estimation data and test data. Any failure beyond time 20 will only count as a failure in the test data. Thus, we need to change the event flag for these in the `in_sample` data if the stop time is beyond time 20. Below, we illustrate how this looks for an individual who do die beyond time 20:

```
# Illustrate with example
set.seed(1117)
tmp <- sim_func()

# Illustrate for individual 146
tmp$res[tmp$res$id == 146, ]
```

```
##       id tstart tstop event     x1      x2     x3     x4      x5
## 784 146   0.00  1.81     0 0.9495 0.58882 0.2244 0.6772 0.23857
## 785 146   1.81  3.27     0 0.5870 0.01161 0.2709 0.6255 0.24397
## 786 146   3.27  5.45     0 0.9181 0.15308 0.4282 0.3712 0.87772
## 787 146   5.45  8.20     0 0.6698 0.98267 0.2053 0.2813 0.03428
## 788 146   8.20  9.26     0 0.6440 0.62564 0.9090 0.5274 0.17542
## 789 146   9.26 11.80     0 0.2434 0.60591 0.2036 0.2170 0.07848
## 790 146  11.80 13.70     0 0.7454 0.29490 0.9624 0.2426 0.33665
## 791 146  13.70 14.94     0 0.6580 0.40228 0.3234 0.7857 0.34378
## 792 146  14.94 16.28     0 0.2397 0.31621 0.1368 0.7452 0.13836
## 793 146  16.28 18.52     0 0.4724 0.10613 0.7525 0.7799 0.88757
## 794 146  18.52 21.00     1 0.2503 0.20445 0.8681 0.1589 0.49792
```

```
# Split data
d_split <- split_data_func(tmp$res)

# In sample data (notice event flag is changed and last tstop)
```

```r
d_split$in_sample[d_split$in_sample$id == 146, ]
```

```
##      id tstart tstop event     x1      x2     x3     x4      x5
## 784 146   0.00  1.81 FALSE 0.9495 0.58882 0.2244 0.6772 0.23857
## 785 146   1.81  3.27 FALSE 0.5870 0.01161 0.2709 0.6255 0.24397
## 786 146   3.27  5.45 FALSE 0.9181 0.15308 0.4282 0.3712 0.87772
## 787 146   5.45  8.20 FALSE 0.6698 0.98267 0.2053 0.2813 0.03428
## 788 146   8.20  9.26 FALSE 0.6440 0.62564 0.9090 0.5274 0.17542
## 789 146   9.26 11.80 FALSE 0.2434 0.60591 0.2036 0.2170 0.07848
## 790 146  11.80 13.70 FALSE 0.7454 0.29490 0.9624 0.2426 0.33665
## 791 146  13.70 14.94 FALSE 0.6580 0.40228 0.3234 0.7857 0.34378
## 792 146  14.94 16.28 FALSE 0.2397 0.31621 0.1368 0.7452 0.13836
## 793 146  16.28 18.52 FALSE 0.4724 0.10613 0.7525 0.7799 0.88757
## 794 146  18.52 20.00 FALSE 0.2503 0.20445 0.8681 0.1589 0.49792
```

```r
# Out sample data (notice tstart is changed)
d_split$out_sample[d_split$out_sample$id == 146, ]
```

```
##      id tstart tstop event     x1     x2     x3     x4     x5
## 794 146     20    21     1 0.2503 0.2044 0.8681 0.1589 0.4979
```

Simulation

We can now run the simulation with the following code. We end the code by printing the mean Brier score for the test data:

```r
# Setup
N <- 100                                    # number of simulations
n <- 1000                                   # number of series
out <- matrix(NA_real_, nrow = N, ncol = 4) # matrix for output

# Run simulation
set.seed(42)
for(i in 1:N){
  # Simulate data and split
  repeat{
    sims <- sim_func(n)

    # We want some survivers and some deaths
    if(sum(sims$res$event) > 50 && n - sum(sims$res$event) > 50)
      break
  }
  d_split <- split_data_func(sims$res)

  # Estimate models
  static_fit <- fit_funcs$static(d_split$in_sample)
  ekf_fit <- fit_funcs$dd(d_split$in_sample)
  ekf_extra_fit <- fit_funcs$dd(d_split$in_sample, NR_eps = .01)
  ukf_fit <- fit_funcs$dd_UKF(d_split$in_sample)

  # Predict outcome
  error <- list(
     static =
       predict(static_fit, d_split$out_sample, type = "response"),
```

```
    ekf = if(is.na(ekf_fit)) NA else
      predict(ekf_fit, new_data = d_split$out_sample,
              tstart = "tstart", tstop = "tstop")$fits,

    ekf_extra = if(is.na(ekf_extra_fit)) NA else
      predict(ekf_extra_fit, new_data = d_split$out_sample,
              tstart = "tstart", tstop = "tstop")$fits,

    ukf = if(is.na(ukf_fit)) NA else
      predict(ukf_fit, new_data = d_split$out_sample,
              tstart = "tstart", tstop = "tstop")$fits)

  # Compute Brier score
  error <- unlist(lapply(
    error, function(x) if(is.na(x)) NA else
     mean.default((x - d_split$out_sample$event)^2)))

  # Save results
  out[i, ] <- error
}

# Print mean for cases where all could fit
colnames(out) <- c("Static", "EKF", "EKF with extra correction", "UKF")
colMeans(out[complete.cases(out), ])
```

```
##                   Static                      EKF
##                  0.09634                  0.07533
## EKF with extra correction                      UKF
##                  0.07842                  0.07713
```

```
# Print median
apply(out[complete.cases(out), ], 2, median)
```

```
##                   Static                      EKF
##                  0.05372                  0.05301
## EKF with extra correction                      UKF
##                  0.05422                  0.05358
```

```
# Print number of cases where all methods succeed to estimate
sum(complete.cases(out))
```

```
## [1] 95
```

Above, we do 100 simulations with 1000 series in each simulation. The EKF does best. Another question is how often the various method got a given rank within a simulation in terms of their Brier score. We answer this question below (the rank are given as the first printed value such that one implies being the lowest Brier score in a given simulation):

```
# Look at number of cases where each method got each rank
knitr::kable(apply(t(apply(out[complete.cases(out), ], 1, rank)),
                   2, function(x) xtabs(~x)),
             caption = "Number of times each set got a given rank in terms of Brier Score",
             row.names = T)
```

Table 10: Number of times each set got a given rank in terms of Brier Score

	Static	EKF	EKF with extra correction	UKF
1	18	30	23	24
2	5	49	13	28
3	17	15	27	36
4	55	1	32	7

The main take away is that the EKF method does better with these specification in terms of getting the lowest mean out-sample Brier score and getting the lowest Brier score in most of the simulation

Linear Time complexity

We will illustrate that the EKF and UKF have linear time complexity in the number of observation. This is particularly easy because the simulation function start of by simulating the coefficients as shown below (hence, variation will not be due to different coefficients vectors and only the number of series):

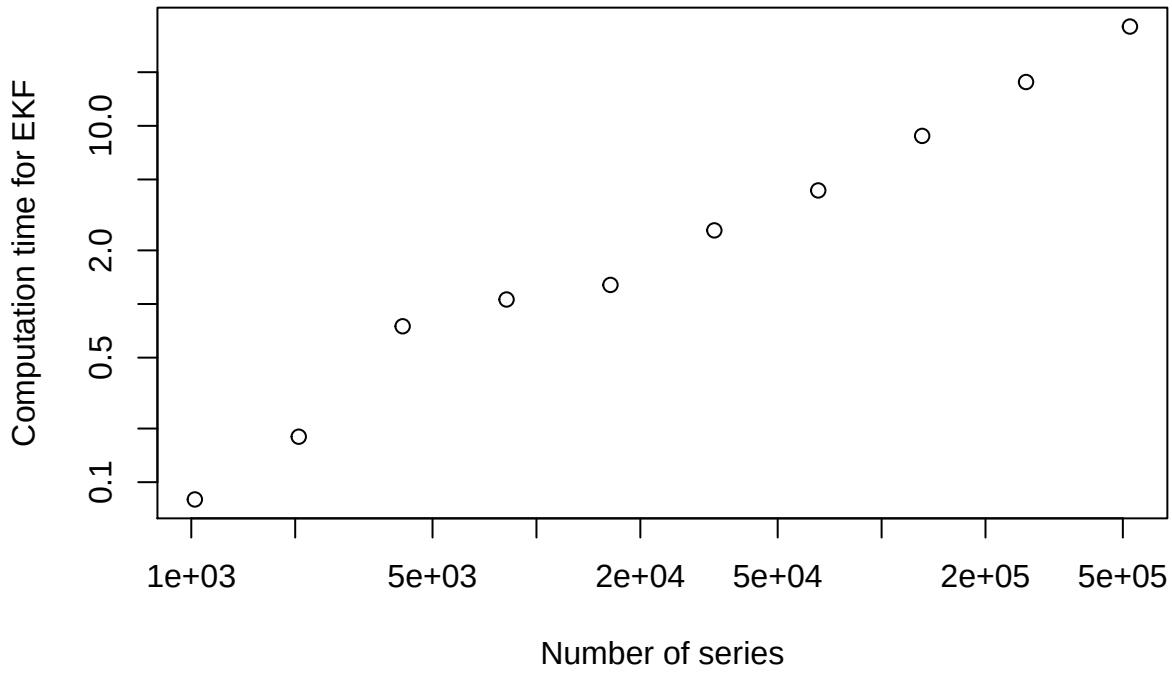
```
some_seed <- 69284
set.seed(some_seed)
res_1 <- test_sim_func_logit(100)

set.seed(some_seed)
res_2 <- test_sim_func_logit(1000) # different number of series

all.equal(res_1$betas, res_2$betas) # Coeffecients are equal
```

```
## [1] TRUE
```

Next, we plot the computation time versus the number of simulation for the EKF and UKF method. Further, we print the linear regression slope for the log-log regression. The slope is close to one implying that the linear time complexity is linear in the number of observations

```
# Define function to record run time for a given number of series
run_time_func <- function(n, sim_args = default_args){
  set.seed(7851348) # Use the same seed
  sim_args$n_series <- n
  sims <- do.call(test_sim_func_logit, sim_args)

  time_EKF <- system.time(fit_EKF <- fit_funcs$dd(sims$res))
  time_UKF <- system.time(
    fit_UKF <- ddhazard(
      formula = Surv(tstart, tstop, event) ~ x1 + x2 + x3 + x4 + x5,
      data = sims$res, max_T = T_max, by = 1, id = sims$res$id,
      Q_0 = diag(.1, n_beta + 1), Q = diag(.1, n_beta + 1),
      control = list(
        eps = 0.1,
        alpha = 1,
        beta = 0,
        method = "UKF")))

  # Check that both succed to fit
  if(is.na(fit_EKF) || is.na(fit_UKF))
```

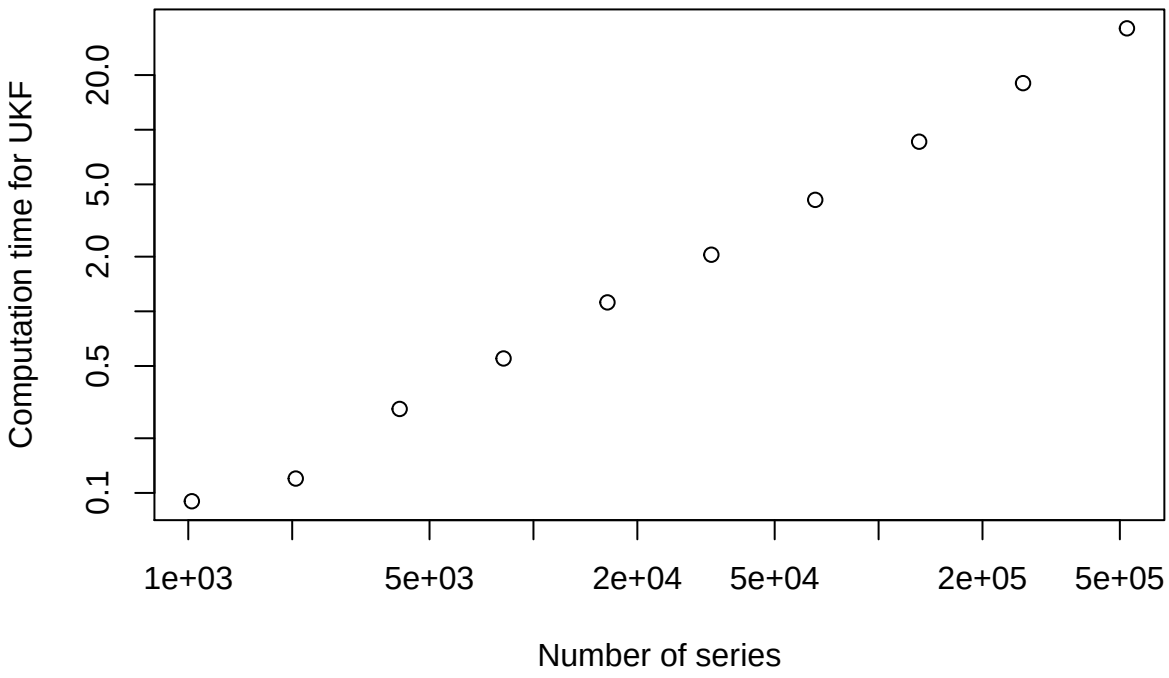
```
    stop()

  list(time_EKF = time_EKF, time_UKF = time_UKF)
}

n_for_test <- 2^(10:19)
run_time <- sapply(n_for_test, run_time_func)

# Plot EKF and print log-log regression slope
ekf_time <- sapply(run_time["time_EKF", ], function(x) x[["user.self"]])
plot(n_for_test, ekf_time, type = "p", log = "xy",
     xlab = "Number of series", ylab = "Computation time for EKF")
```




```
coef(lm(log(ekf_time) ~ log(n_for_test))) # log-log slope is roughly one
```

```
##     (Intercept) log(n_for_test)
##         -8.4855          0.9122
```

```
# Plot UKF and print log-log regression slope
ukf_time <- sapply(run_time["time_UKF", ], function(x) x[["user.self"]])
plot(n_for_test, ukf_time, type = "p", log = "xy",
     xlab = "Number of series", ylab = "Computation time for UKF")
```




```
coef(lm(log(ukf_time) ~ log(n_for_test))) # log-log slope is roughly one
```

```
##     (Intercept) log(n_for_test)
##         -9.4472          0.9849
```