Simulation study with logit model

Benjamin Christoffersen

'2016-11-29

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
- 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 * are showing 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. 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 $d + 1$. Notice the distinction between out-sample and out-of-time. The former, out-of-time, is where we estimate the model up to time d using one set and then apply the estimates to predict the outcomes of the other set up to time d. The latter is where we observe all series to time d and then perform predicition for those who are still alive at time d for whether they die at e.g. $d + 1$

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

All method with 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 coeffecient are simulated to change. For example, we bin correctly where we simulate the coeffecients to change at time $1, 2, \ldots, d$ and we estimate the coeffecient at time $1, 2, \ldots, d$. 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
```

```
## [1] "boennecd/dynamichazard@aa871d43235a4c2d45966e749d2dcb69010bfba6"
```

```
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 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 $\vec{\beta}_0, \ldots, \vec{\beta}_d$ and given number n of series
- Series/individuals: A person/individual starting at time 0 and 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
- Miss 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 disadvantages that any given data set may require some tuning by the user. Although as will be shown, tunning by the user is not often needed with data sets like those presented here

Setup

The following values will be used in the simulation:

```r
ns <- c(200, 800, 2000) # Number of series
n_beta <- 5             # Number of covariates
T_max <- 10             # The last time we observe
n_sims <- 100           # Number of simulation in each run

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

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

```r
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 on the github site. `T_max` is the

number of bins/intervals we will 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 where we fit are second order random walk

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 = -4, # start value of intercept
  sds = c(.1, rep(1, 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
```

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,

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

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 $\eta \sim \text{Pois}(1)$ 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
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$betas, type = "l", lty = 1, ylab = "time", xlab = "beta",
          ylim = range(sims$betas, ylim), col = 1:(n_beta + 1)))()
```
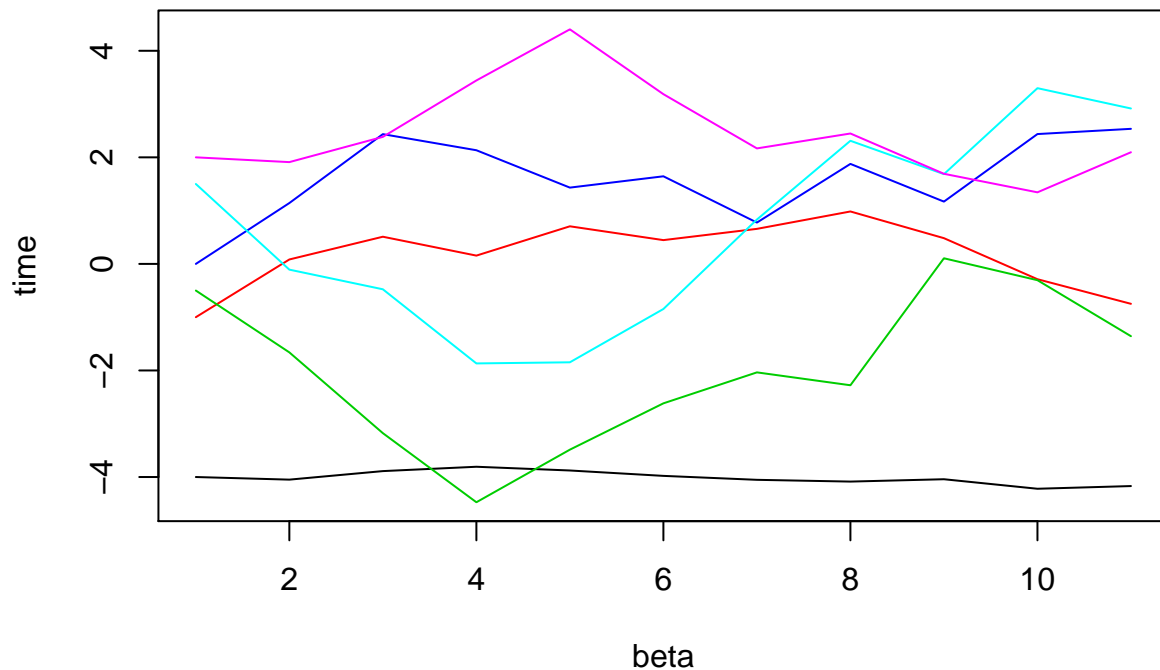
The black line is the intercept while the colered lines are the coefficients for the covariates. Next, we can look at the number of failures in each simulation:

```r
# 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
# See ?do.call
set.seed(9654731)
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]  372 1216  956 1825 1932  305 1929 1634  817  892  875 2000  923  484
## [15]  983
```

* Defintion of fit functions

We will define functions to estimate the different 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`
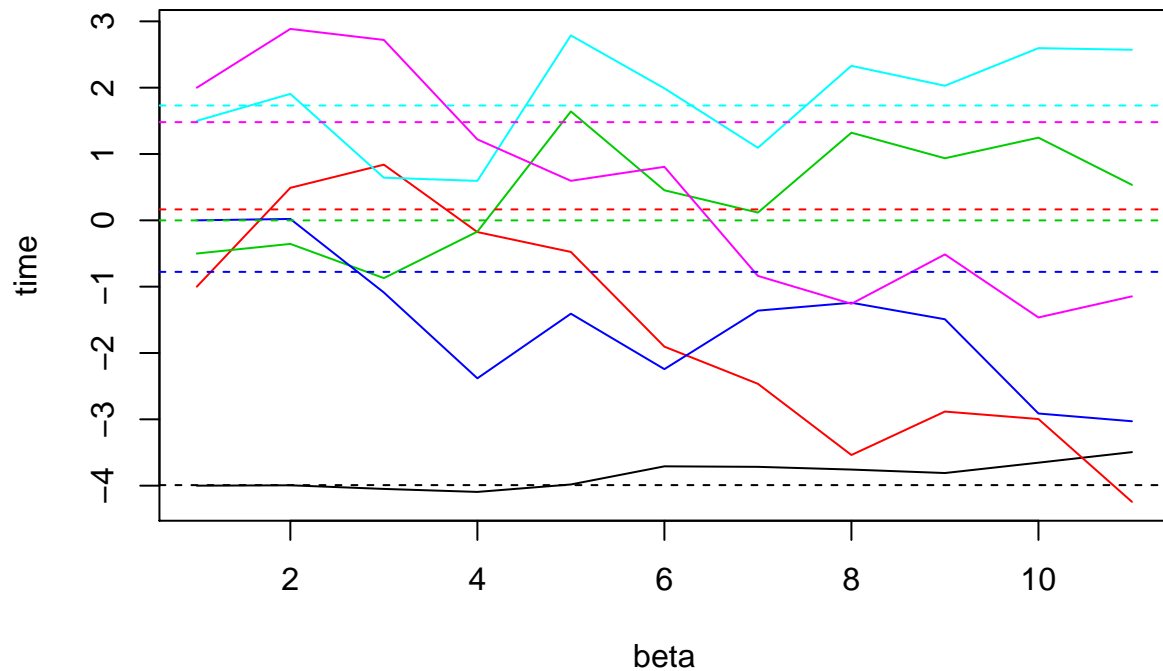
```r
library(survival); library(dynamichazard)

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

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

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

```r
# 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 simulate a first order random walk 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 loops
fit_funcs$dd <- function(s = sims$res, LR = 1, NR_eps = NULL)
  tryCatch({
    ddhazard(
      formula = Surv(tstart, tstop, event) ~ . - tstart - tstop - id - event,
      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(LR = LR, NR_eps = NR_eps, eps = 0.1))
  }, 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)
```
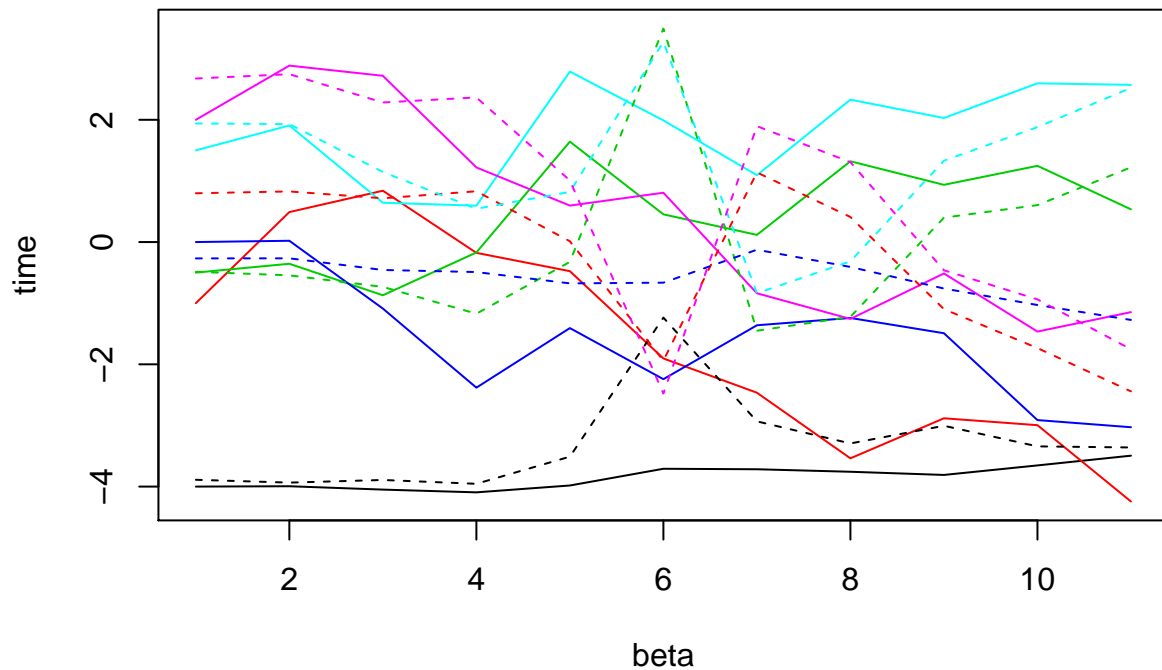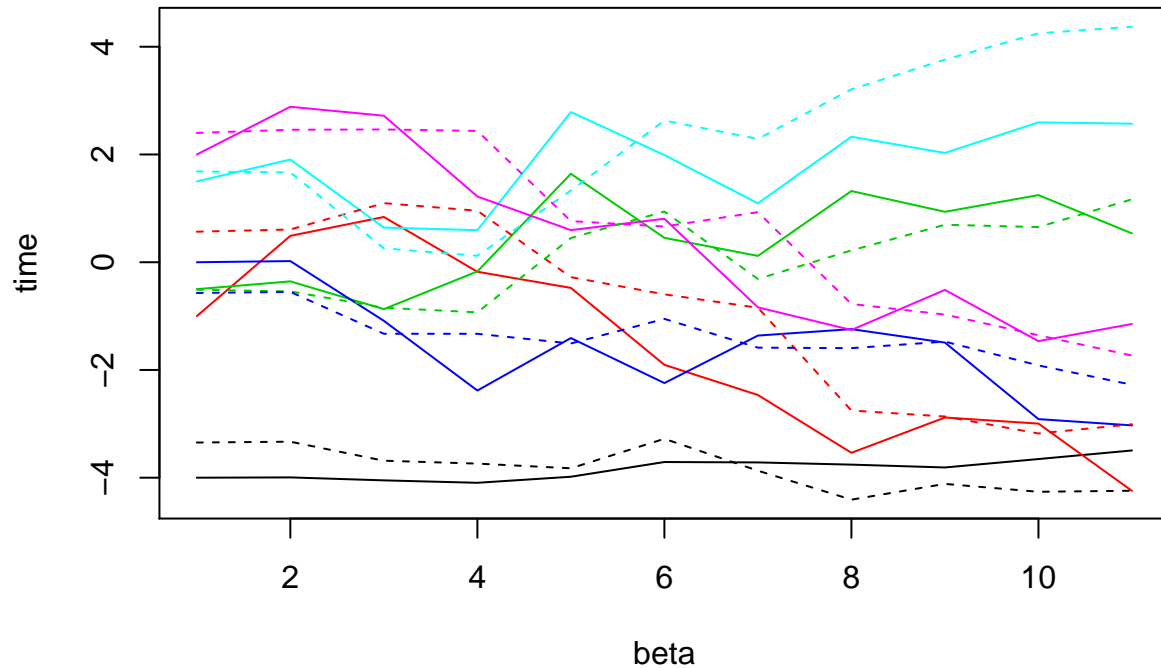


```r
# Same call with extra iterations
fit <- fit_funcs$dd(LR = .5, NR_eps = .01)

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

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


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

```r
# Fitting with UKF
fit_funcs$dd_UKF <- function(s = sims$res, alpha = 1, beta = 0){
  tryCatch({
    ddhazard(
      formula = Surv(tstart, tstop, event) ~ . - tstart - tstop - id - event,
      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)
```
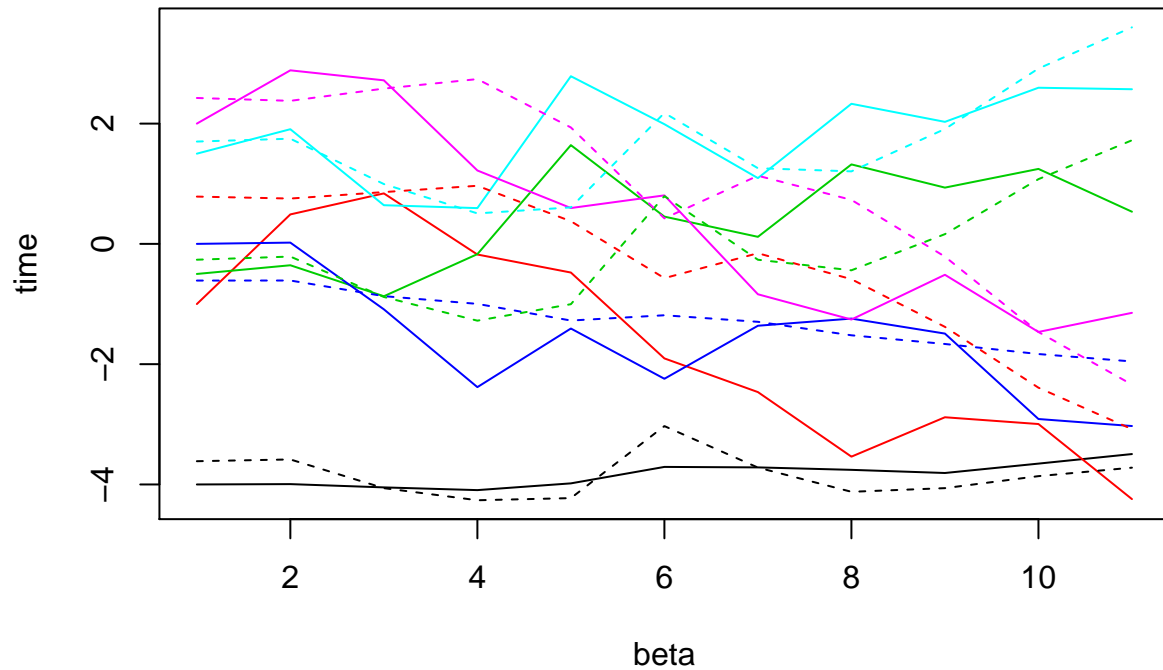
Below, we define a function to simulate a first order random walk where only one parameter (`x2`) is time varying:

```r
# Fitting with fixed effects
fit_funcs$dd_fixed <- function(s = sims$res, LR = 1, NR_eps = NULL){
  tryCatch({
    ddhazard(
      formula = Surv(tstart, tstop, event) ~
        -1 + ddFixed(rep(1, length(x1))) +     # 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))
  }, 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)
```
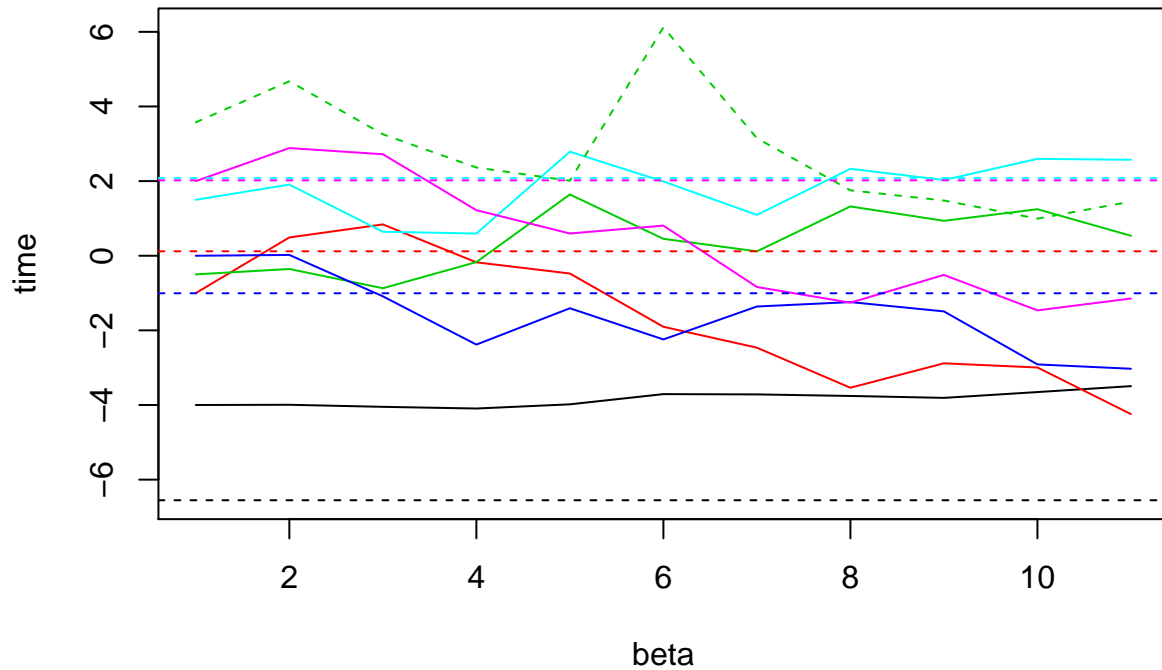
Next, we define a function to fit the model with a second order random walk:

```r
# 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) ~ . - tstart - tstop - id - event,
      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), rep(0, 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)
```
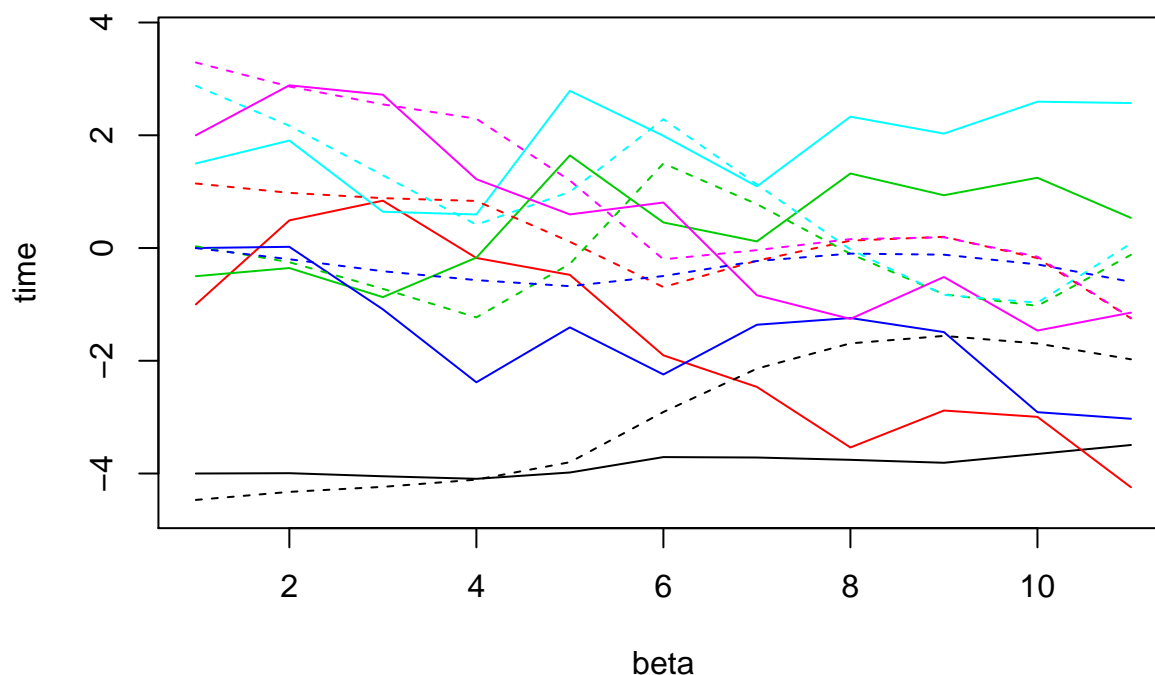
** Defintion of GAM fit function

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

```r
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) ~ . - tstart - tstop - id - event,
              data = s, max_T = T_max, by = 1, id = s$id, use_weights = F)
  # fit model
  bam(
    formula = Y ~
      # cr is cubic basis with dim of k
      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")
}

# 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)
  lines(sims$betas[-1, i + 1], col = i + 1)
}
```
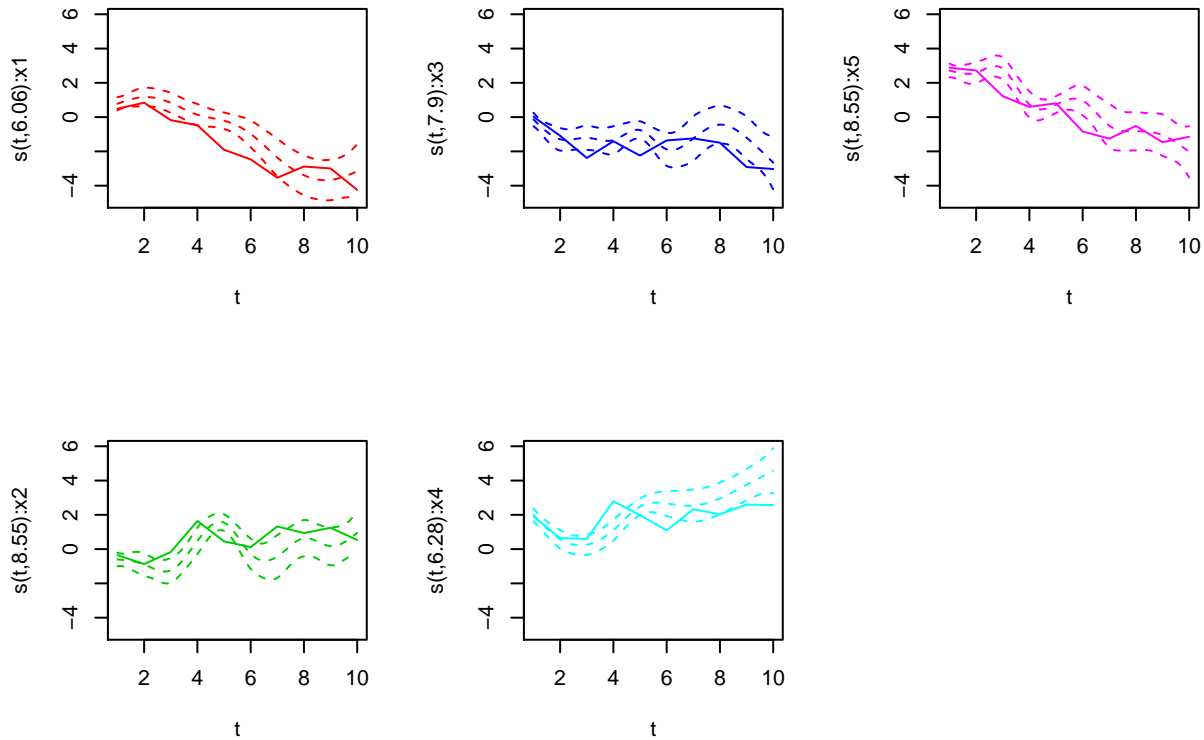

** Definition of prediction functions

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:

```
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, ]))
}

# Illustrate use
```

```r
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 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$general <- function(brier_func, fit, eval_data_frame){
  d_frame <- get_survival_case_weigths_and_data(
    formula = Surv(tstart, tstop, event) ~ . - tstart - tstop - id - event,
    data = eval_data_frame, max_T = T_max, by = 1, id = eval_data_frame$id,
    use_weights = F)

  # 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.default(resids^2),
       median_abs_res = median(abs(resids)),
       sd_res = sd(resids),
       # Info is added to debug
       info = cbind(resids = resids,
                    tstop = d_frame$tstop,
                    id = d_frame$id))
}

# 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
##     0.06507569     0.06180691     0.25511837
```

```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
##     0.06782792     0.06113168      0.26044098
```

```r
# 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
##     0.06323912     0.05258154      0.24933527
```

```r
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
##     0.06679688     0.05249414      0.25673126
```

```r
# 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
##     0.05599008     0.03507989      0.23663986
```

```r
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
##     0.05955587     0.03473694      0.24404534
```

** 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, then the new function perform `n_sims = 100` simulations for each of values `ns` (200, 800, 2000):

```r
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, 7, 3),
                 dimnames = list(
                   NULL,
                   c("static", "Extra correction", "Single correction",
                     "2 order EKF", "Fixed effect", "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 suffecient amount of deaths and
      # survivers
      repeat{
        sims <- sim_func(n)

        # We want some survivers and some deaths
        if(sum(sims$res$event) > 40 && n - sum(sims$res$event) > 40)
          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])
```

```r
  }

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

  # Fit fixed effect
  dd_fixed <- fit_funcs$dd_fixed(sim_split$fit_dat)

  # 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, dd_UKF, gam_fit))

  eval_funcs = c(brier_funcs$static,
                 replicate(length(dd_fits) + 3, brier_funcs$dd),
                 brier_funcs$gam)

  for(j in seq_along(models)){
    if(length(models[[j]]) == 1 && is.na(models[[j]]))
      next # We have to skip models 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
print(knitr::kable(t(apply(out[, , 1], 2, function(x) n_sims - sum(is.na(x)))), caption = paste(
  "Number of successful fits with", n/2, "series in test and fit data.")))
cat("\n")

n_cases_all_success <- sum(complete.cases(out[, , 1]))

if(n_cases_all_success > 0){
  print(knitr::kable(t(apply(out[complete.cases(out[, , 1]), , , drop = F],
                             3, colMeans)),
        digits = 3, caption = paste(
  "Mean of metrics with", n/2, "series in test and fit data. Only simulations that succeeds for all
  cat("\n")
}
```

```
    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 tl

    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: Number of successful fits with 100 series in test and fit data.

static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
100	79	100	100	100	100	100

Table 2: Mean of metrics with 100 series in test and fit data. Only simulations that succeeds for all setups are included. There are 79 of these simulations

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.066	0.069	0.061	0.062	0.065	0.060	0.059
Median abs res	0.056	0.029	0.056	0.065	0.049	0.047	0.030
sd res	0.252	0.257	0.242	0.242	0.249	0.240	0.239

Table 3: Mean of metrics with 100 series in test and fit data. All simulations for each setup where the fit succeeds for the given setup are included in each cell estimate

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.068	0.069	0.063	0.063	0.066	0.062	0.061
Median abs res	0.059	0.029	0.059	0.068	0.051	0.050	0.032
sd res	0.255	0.257	0.245	0.245	0.252	0.244	0.242

Table 4: Number of successful fits with 400 series in test and fit data.

static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
100	93	100	100	100	98	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 91 of these simulations

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.074	0.064	0.066	0.066	0.072	0.066	0.063
Median abs res	0.071	0.046	0.064	0.069	0.052	0.056	0.045
sd res	0.260	0.244	0.246	0.245	0.257	0.247	0.241

Table 6: Mean of metrics with 400 series in test and fit data. All simulations for each setup where the fit succeeds for the given setup are included in each cell estimate

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.073	0.066	0.065	0.065	0.071	0.065	0.062
Median abs res	0.073	0.049	0.066	0.070	0.053	0.053	0.046
sd res	0.258	0.246	0.244	0.243	0.255	0.242	0.239

Table 7: Number of successful fits with 1000 series in test and fit data.

static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
100	95	98	100	100	96	100

Table 8: Mean of metrics with 1000 series in test and fit data. Only simulations that succeeds for all setups are included. There are 90 of these simulations

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.079	0.065	0.071	0.069	0.073	0.071	0.065
Median abs res	0.084	0.053	0.069	0.071	0.057	0.048	0.052
sd res	0.266	0.243	0.252	0.250	0.258	0.251	0.243

Table 9: Mean of metrics with 1000 series in test and fit data. All simulations for each setup where the fit succeeds for the given setup are included in each cell estimate

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.077	0.065	0.069	0.067	0.072	0.069	0.063
Median abs res	0.081	0.052	0.066	0.066	0.053	0.046	0.049
sd res	0.263	0.243	0.248	0.246	0.254	0.246	0.238

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 effect`) 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 variying parameter

In this part, we will look at the performs when only a single covariate (`x2`) varies. Thus, we can see if the model where only (`x2`) is modelled as varying performs performs better. Or in other words how the miss specified models behaves

** 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_variying <- function(n){
  test_sim_func_logit(
    n_series = n,
    sds = c(sqrt(3)), # Large variance
    is_fixed = c(1:2, 4:6), # All but parem 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_variying(200)$res$event)) # print # of failures
```

```
##  [1] 179  62 158  90  90 129 198 199 200  20
```

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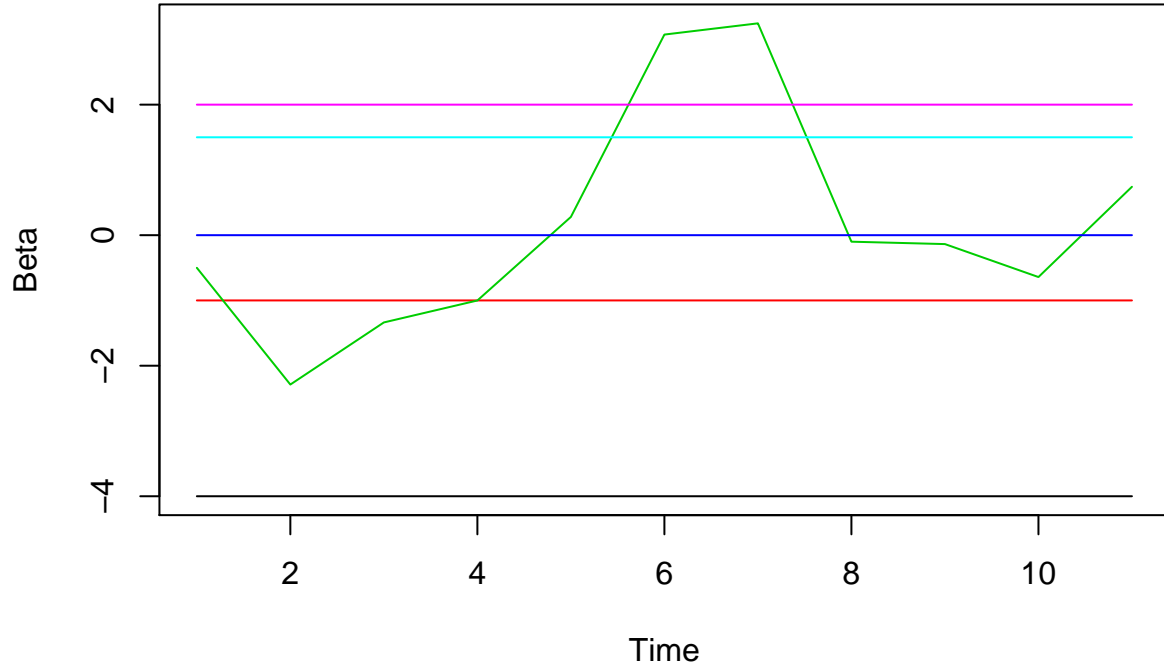
Simulating

We can simulate with the following call:

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

Table 10: Number of successful fits with 100 series in test and fit data.

static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
100	87	100	100	100	100	100

Table 11: Mean of metrics with 100 series in test and fit data. Only simulations that succeeds for all setups are included. There are 87 of these simulations

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.057	0.062	0.055	0.056	0.056	0.055	0.056
Median abs res	0.049	0.034	0.052	0.060	0.042	0.044	0.036
sd res	0.231	0.242	0.229	0.230	0.231	0.229	0.230

Table 12: Mean of metrics with 100 series in test and fit data. All simulations for each setup where the fit succeeds for the given setup are included in each cell estimate

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.058	0.062	0.056	0.057	0.057	0.056	0.056
Median abs res	0.050	0.034	0.053	0.061	0.042	0.044	0.037
sd res	0.234	0.242	0.231	0.233	0.234	0.231	0.232

Table 13: Number of successful fits with 400 series in test and fit data.

static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
100	97	100	100	100	100	100

Table 14: Mean of metrics with 400 series in test and fit data. Only simulations that succeeds for all setups are included. There are 97 of these simulations

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.068	0.063	0.064	0.065	0.065	0.064	0.062
Median abs res	0.070	0.055	0.065	0.068	0.051	0.059	0.053
sd res	0.248	0.241	0.241	0.244	0.243	0.241	0.239

Table 15: Mean of metrics with 400 series in test and fit data. All simulations for each setup where the fit succeeds for the given setup are included in each cell estimate

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.068	0.063	0.064	0.065	0.065	0.064	0.063
Median abs res	0.070	0.055	0.065	0.068	0.050	0.059	0.053
sd res	0.249	0.241	0.242	0.245	0.244	0.242	0.240

Table 16: Number of successful fits with 1000 series in test and fit data.

static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
100	97	100	99	100	100	100

Table 17: 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

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.082	0.073	0.075	0.075	0.076	0.075	0.073
Median abs res	0.090	0.069	0.079	0.082	0.061	0.071	0.067

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
sd res	0.274	0.261	0.263	0.263	0.265	0.263	0.260

Table 18: Mean of metrics with 1000 series in test and fit data. All
simulations for each setup where the fit succeeds for the given setup
are included in each cell estimate

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.082	0.073	0.076	0.075	0.077	0.076	0.073
Median abs res	0.091	0.069	0.080	0.084	0.060	0.070	0.068
sd res	0.275	0.261	0.265	0.265	0.267	0.264	0.262

Conclussion on run

The main interest here is how the correctly specified model labeled `Fixed effect` 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 miss
specifying the coefficients as time varying. This may suggest that incorrectly 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
$0, 0.1, 0.2, \ldots$ and whether and individual dies is evulated at the same times when we simulate. However, the
model will still use 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)
```

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

  res
}
```

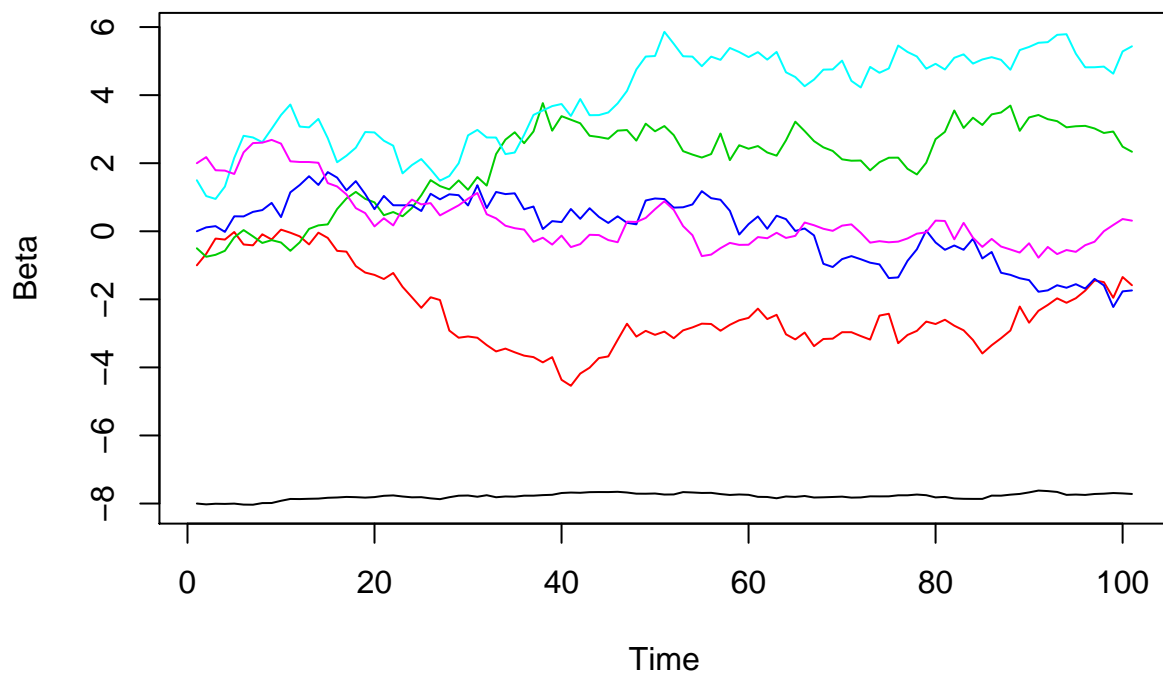## * Illustration of single simulation

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

```
##  [1] 157  90  78 200  22   2 159   9  47 195
```

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


Simulating

We are now able to simulate with the following call:

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

Table 19: Number of successful fits with 100 series in test and fit data.

static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
100	87	100	100	100	100	100

Table 20: Mean of metrics with 100 series in test and fit data. Only simulations that succeeds for all setups are included. There are 87 of these simulations

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.056	0.060	0.054	0.053	0.056	0.052	0.051
Median abs res	0.054	0.041	0.053	0.063	0.039	0.046	0.030
sd res	0.230	0.236	0.224	0.223	0.230	0.222	0.221

Table 21: Mean of metrics with 100 series in test and fit data. All simulations for each setup where the fit succeeds for the given setup are included in each cell estimate

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.058	0.060	0.055	0.055	0.058	0.054	0.053
Median abs res	0.057	0.041	0.056	0.066	0.042	0.049	0.032
sd res	0.234	0.236	0.228	0.227	0.234	0.226	0.225

Table 22: Number of successful fits with 400 series in test and fit data.

static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
100	94	100	100	100	100	100

Table 23: Mean of metrics with 400 series in test and fit data. Only simulations that succeeds for all setups are included. There are 94 of these simulations

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.050	0.043	0.046	0.045	0.049	0.046	0.043
Median abs res	0.049	0.030	0.041	0.045	0.028	0.029	0.027
sd res	0.208	0.196	0.200	0.197	0.206	0.199	0.194

Table 24: Mean of metrics with 400 series in test and fit data. All simulations for each setup where the fit succeeds for the given setup are included in each cell estimate

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.051	0.043	0.047	0.046	0.050	0.047	0.044
Median abs res	0.051	0.030	0.042	0.047	0.028	0.029	0.027
sd res	0.210	0.196	0.202	0.199	0.208	0.201	0.196

Table 25: Number of successful fits with 1000 series in test and fit data.

static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
100	94	100	100	100	98	100

Table 26: Mean of metrics with 1000 series in test and fit data. Only simulations that succeeds for all setups are included. There are 92 of these simulations

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.041	0.037	0.040	0.041	0.043	0.039	0.036
Median abs res	0.042	0.027	0.037	0.040	0.023	0.028	0.026
sd res	0.181	0.172	0.178	0.180	0.183	0.176	0.171

Table 27: Mean of metrics with 1000 series in test and fit data. All simulations for each setup where the fit succeeds for the given setup are included in each cell estimate

	static	Extra correction	Single correction	2 order EKF	Fixed effect	UKF	gam
Brier	0.043	0.038	0.042	0.043	0.044	0.040	0.037
Median abs res	0.044	0.029	0.039	0.043	0.023	0.027	0.027
sd res	0.183	0.175	0.180	0.183	0.185	0.175	0.173

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 not too far off from the `gam` fit for the model/method with the best result. In fact, the EKF model with the extra iteration has almost equal performs with the largest number of series

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 and 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

```r
# Define simulation function
out_sample_args <- default_args
out_sample_args$t_max <- default_args$t_max + 1

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 = 10){
  # 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 time period (`t_max`) by one which is the only difference in the simulation. Notice that indviduals can be in both estimation data and test data. Any failure beyond time 10 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 10. Below, we illustrate how this looks for an individual who do die beyond time 10:

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

# Illustrate for individual 185
tmp$res[tmp$res$id == 185, ]
```

```
##       id    tstart     tstop event          x1        x2         x3        x4
## 581 185 0.000000  1.353427     0 0.91531767 0.1073433 0.88193402 0.3860165
## 582 185 1.353427  5.239087     0 0.87565175 0.3303455 0.92365312 0.2077193
## 583 185 5.239087  7.633657     0 0.05045598 0.8661973 0.97055798 0.3900056
## 584 185 7.633657  8.986425     0 0.01638130 0.9365279 0.94012413 0.6748349
## 585 185 8.986425 10.986425     1 0.93038790 0.7395430 0.09007954 0.9169830
##             x5
## 581 0.47889583
## 582 0.01187893
## 583 0.69178351
## 584 0.08837330
## 585 0.67734155
```

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

# In sample data (notice event flag is changed and last tstop)
d_split$in_sample[d_split$in_sample$id == 185, ]
```

```
##        id   tstart     tstop event          x1        x2         x3        x4
## 581 185 0.000000  1.353427 FALSE 0.91531767 0.1073433 0.88193402 0.3860165
## 582 185 1.353427  5.239087 FALSE 0.87565175 0.3303455 0.92365312 0.2077193
## 583 185 5.239087  7.633657 FALSE 0.05045598 0.8661973 0.97055798 0.3900056
## 584 185 7.633657  8.986425 FALSE 0.01638130 0.9365279 0.94012413 0.6748349
## 585 185 8.986425 10.000000 FALSE 0.93038790 0.7395430 0.09007954 0.9169830
##           x5
## 581 0.47889583
## 582 0.01187893
## 583 0.69178351
## 584 0.08837330
## 585 0.67734155
```

```r
# Out sample data (notice tstart is changed)
d_split$out_sample[d_split$out_sample$id == 185, ]
```

```
##        id tstart    tstop event         x1       x2         x3       x4
## 585 185     10 10.98642     1 0.9303879 0.739543 0.09007954 0.916983
##           x5
## 585 0.6773415
```

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
##                 0.09635441                   0.09205384
## EKF with extra correction                          UKF
##                 0.07923824                   0.08594323
```

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

```
## [1] 94
```

Above, we do 100 simulations with 1000 series in each simulation. The EKF with the extra iterations 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")
```

Static	EKF	EKF with extra correction	UKF

Table 28: Number of times each set got a given rank in terms of Brier Score

Static	EKF	EKF with extra correction	UKF
15	26	36	17
8	36	19	31
10	27	23	34
61	5	16	12

The main take away is that the EKF method with the extra iteration in the corretion step 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):

```r
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 the linear time complexity

```r
# 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) ~ . - tstart - tstop - id - event,
      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,
```
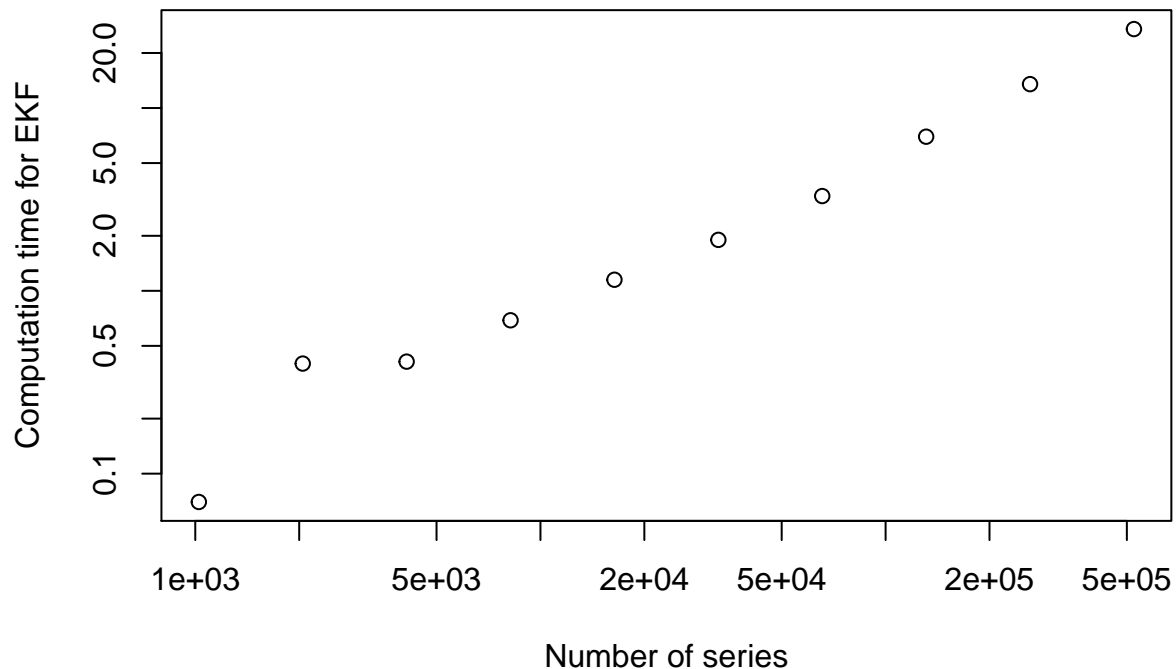
```
        method = "UKF")))

  # 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")
```
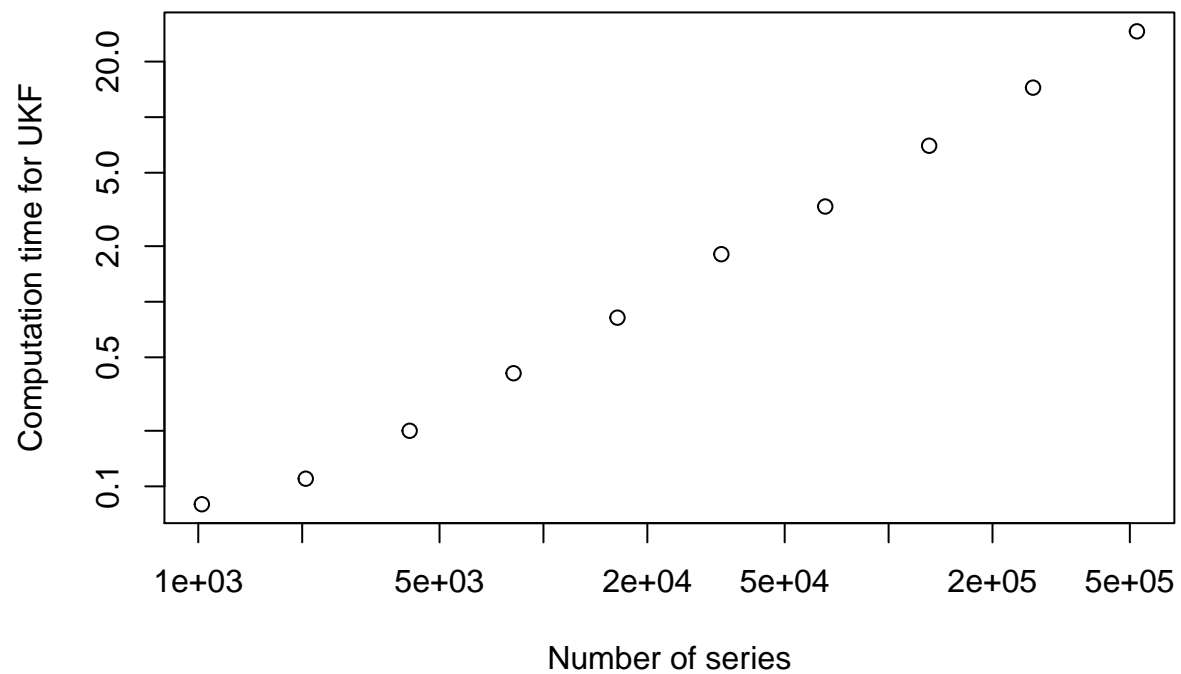


```
coef(lm(log(ekf_time) ~ log(n_for_test))) # log-log slope is roughly one

##     (Intercept) log(n_for_test)
##      -8.0796556       0.8534374
```

```
# 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")
```

29

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

```
##   (Intercept) log(n_for_test)
##    -9.6120755       0.9796281
```