Untitled

Miao Cai

5/28/2020

Functions for simulating PLP and JPLP data

```
# Function: simulating PLP - time truncated case
sim_plp_tau = function(tau = 30,
                       beta = 1.5,
                       theta = 10){
  # initialization
  s = 0; t = 0
  while (max(t) <= tau) {
    u <- runif(1)
    s <- s - log(u)
    t_new <- theta*s^(1/beta)
    t <- c(t, t_new)
  }
  t = t[c(-1, -length(t))]

  return(t)
}

# Function: simulate multiple NHPPs
sim_hier_plp_tau = function(N, beta = 1.5, theta){
  t_list = list()
  len_list = list()
  tau_vector = rnorm(N, 10, 1.3)

  for (i in 1:N) {
    t_list[[i]] = sim_plp_tau(tau_vector[i], beta = beta, theta = theta[i])
    len_list[[i]] = length(t_list[[i]])
  }

  event_dat = data.frame(
    shift_id = rep(1:N, unlist(len_list)),
    event_time = Reduce(c, t_list)
  )

  start_end_dat = data.frame(
    shift_id = 1:N,
    start_time = rep(0, N),
    end_time = tau_vector #difference2
  )

  return(list(event_dat = event_dat,
```

```r
              start_end_dat = start_end_dat,
              shift_length = unlist(len_list)))
}

# Function: Simulating hierarchical PLP data for D drivers
sim_hier_nhpp = function(
  beta = 1.5,             # Shape parameter for PLP
  D = 10,                 # the number of drivers
  K = 3,                  # the number of predictor variables
  group_size_lambda = 10, # the mean number of shifts for each driver
  mu0 = 0.2,              # Hyperparameters: mean
  sigma0 = 0.5,           # Hyperparameters: s.e.
  R_K = c(1, 0.3, 0.2)    # Fixed-effects parameters
)
{
  # 1. Random-effect intercepts
  r_0D = rnorm(D, mean = mu0, sd = sigma0)

  # 3. The number of shifts in the $d$-th driver: $N_{d}$
  N_K = rpois(D, group_size_lambda)
  N = sum(N_K) # the total number of obs
  id = rep(1:D, N_K)

  # 4. Generate data: x_1, x_2, .. x_K
  sim1 = function(group_sizes = N_K)
  {
    ntot = sum(group_sizes)

    int1 = rep(1, ntot)
    x1 = rnorm(ntot, 1, 1)
    x2 = rgamma(ntot, 1, 1)
    x3 = rpois(ntot, 2)

    return(data.frame(int1, x1, x2, x3))
  }
  X = sim1(N_K)

  # 5. Scale parameters of a NHPP
  # 5a. parameter matrix: P
  P = cbind(r0 = rep(r_0D, N_K),
            t(replicate(N, R_K)))
  M_logtheta = P*X

  # returned parameter for each observed shift
  theta_vec = exp(rowSums(M_logtheta))

  df = sim_hier_plp_tau(N = N, beta = beta, theta = theta_vec)

  hier_dat = list(
    N = nrow(df$event_dat),
    K = K,
    S = nrow(df$start_end_dat),
    D = max(id),
```

```r
    id = id,     # driver index at shift level
    tau = df$start_end_dat$end_time,
    event_time = df$event_dat$event_time,
    group_size = df$shift_length,  # the number of events in each shift
    X_predictors = X[,2:4]
  )

  true_params = list(
    mu0 = mu0, sigma0 = sigma0,
    r0 = r_OD, r1_rk = R_K,
    beta = beta,
    theta = theta_vec
  )

  return(list(hier_dat = hier_dat, true_params = true_params))
}
```

JPLP

```r
# Function: sample the number of stops from 1:4
get_n_stop = function() sample(1:4, 1, TRUE)

# Function: Define a inverse function for mean function Lambda
inverse = function (f, lower = 0.0001, upper = 10000) {
  function (y) uniroot((function (x) f(x) - y), lower = lower, upper = upper)[1]
}

# Function: Mean function Lambda for PLP
Lambda_PLP = function(t, beta = 1.5, theta = 4) return((t/theta)^beta)

# Function: Mean function Lambda for JPLP
Lambda_JPLP = function(
  t,                          # Time of the event
  tau = 12,                   # Shift end time (right-censor time)
  kappa = 0.8,                # "Jump" parameter in JPLP
  t_trip = c(3.5, 6.2, 9),    # trip stop time
  beta = 1.5,                 # Shape parameter
  theta = 4)                  # Rate parameter
{
  t_trip1 = c(0, t_trip)
  n_trip = length(t_trip1)
  comp = Lambda_PLP(t_trip, beta, theta)
  kappa_vec0 = rep(kappa, n_trip - 1)^(0:(n_trip - 2))
  kappa_vec1 = rep(kappa, n_trip - 1)^(1:(n_trip - 1))
  cum_comp0 = comp*kappa_vec0
  cum_comp1 = comp*kappa_vec1
  index_trip = max(cumsum(t > t_trip1)) - 1

  if(index_trip == 0){
    return((t/theta)^beta)
  }else{
    return(sum(cum_comp0[1:index_trip]) - sum(cum_comp1[1:index_trip]) +
             kappa^index_trip*(t/theta)^beta)
  }
```

```
}

# Function: sim_jplp:  simulate event times generated from a JPLP
sim_jplp = function(
  tau0 = 12,                  # Shift end time (right-censor time)
  kappa0 = 0.8,               # "Jump" parameter in JPLP
  t_trip0 = c(3.5, 6.2, 9),   # trip stop time
  beta0 = 1.2,                # Shape parameter
  theta0 = 0.5                # Rate parameter
)
{ s = 0; t = 0
  Lambda1 = function(t, tau1 = tau0, kappa1 = kappa0, t_trip1 = t_trip0,
                     beta1 = beta0, theta1 = theta0)
    {
    return(Lambda_JPLP(t, tau = tau1, kappa = kappa1,
                       t_trip = t_trip1, beta = beta1, theta = theta1))
    }
  inv_Lambda = inverse(Lambda1, 0.0001, 10000)

  while (max(t) <= tau0)
    {
    u <- runif(1)
    s <- s - log(u)
    t_new <- inv_Lambda(s)$root
    t <- c(t, t_new)
    }

  t = t[c(-1, -length(t))]

  return(t)
}


# Function: sim_mul_jplp: simulate event times for multiple shifts
sim_mul_jplp = function(
  kappa = 0.8,       # "Jump" parameter in JPLP
  beta = 1.2,        # Shape parameter
  theta = 2,         # Rate parameter
  n_shift = 10       # Number of shifts
)
{
  t_shift_vec = list()
  n_trip_vec = list()
  id_trip_vec = list()
  t_start_vec = list()
  t_stop_vec = list()
  n_event_shift_vec = list()
  t_event_vec = list()
  n_event_trip_vec = list()

  for (i in 1:n_shift) {
    sim_tau = rnorm(1, 10, 1.3)
    n_stop = get_n_stop()
```

```r
  sim_t_trip = round((1:n_stop)*sim_tau/(n_stop + 1) +
                       rnorm(n_stop, 0, sim_tau*0.15/n_stop), 2)
  t_events = sim_jplp(tau0 = sim_tau,
                      kappa0 = kappa,
                      t_trip0 = sim_t_trip,
                      beta0 = beta,
                      theta0 = theta)
  t_shift_vec[[i]] = sim_tau
  n_trip_vec[[i]] = n_stop + 1
  id_trip_vec[[i]] = 1:(n_stop + 1)
  t_start_vec[[i]] = c(0, sim_t_trip)
  t_stop_vec[[i]]  = c(sim_t_trip, sim_tau)
  n_event_shift_vec[[i]] = length(t_events)
  t_event_vec[[i]] = t_events

  tmp_n_event_trip = rep(NA_integer_, (n_stop + 1))
  for (j in 1:(n_stop + 1)) {
     tmp_n_event_trip[j] = sum(t_events > t_start_vec[[i]][j] &
                                 t_events <= t_stop_vec[[i]][j])
  }
  n_event_trip_vec[[i]] = tmp_n_event_trip
}

event_dt = data.frame(
  shift_id = rep(1:n_shift, unlist(n_event_shift_vec)),
  trip_id = rep(Reduce(c, id_trip_vec), Reduce(c, n_event_trip_vec)),
  event_time = Reduce(c, t_event_vec)
)

trip_dt = data.frame(
  shift_id = rep(1:n_shift, Reduce(c, n_trip_vec)),
  trip_id = Reduce(c, id_trip_vec),
  t_trip_start = Reduce(c, t_start_vec),
  t_trip_end = Reduce(c, t_stop_vec),
  N_events = Reduce(c, n_event_trip_vec)
)

shift_dt = data.frame(
  shift_id = 1:n_shift,
  start_time = rep(0, n_shift),
  end_time = Reduce(c, t_shift_vec)
)

stan_dt = list(N = nrow(event_dt),
              S = nrow(trip_dt),
              r_trip = trip_dt$trip_id,
              t_trip_start = trip_dt$t_trip_start,
              t_trip_end = trip_dt$t_trip_end,
              event_time = event_dt$event_time,
              group_size = trip_dt$N_events)

return(list(event_dt = event_dt,
            trip_dt = trip_dt,
```

```r
              shift_dt = shift_dt,
              stan_dt = stan_dt))

}

# Function: sim_hier_JPLP:  hierarchical JPLP for D different drivers
sim_hier_JPLP = function(
  beta = 1.2,               # Shape parameter for JPLP
  kappa = 0.8,              # "jump" parameter in JPLP
  D = 10,                   # the number of drivers
  K = 3,                    # the number of predictor variables
  group_size_lambda = 10,  # the mean number of shifts for each driver
  mu0 = 0.2,                # hyperparameter 1
  sigma0 = 0.5,             # hyperparameter 2
  R_K = c(1, 0.3, 0.2)     # Fixed-effects parameters
)
{
  # 1. Random-effect intercepts
  r_0D = rnorm(D, mean = mu0, sd = sigma0)

  # 3. The number of observations (shifts) in the $d$-th driver: $N_{d}$
  N_K = rpois(D, group_size_lambda)
  N = sum(N_K) # the total number of shifts for all D drivers
  # id = rep(1:D, N_K)

  # 4. Generate data: x_1, x_2, .. x_K
  simX = function(group_sizes = N_K)
  {
    ntot = sum(group_sizes)

    int1 = rep(1, ntot)
    x1 =  rnorm(ntot, 1, 1)
    x2 = rgamma(ntot, 1, 1)
    x3 =  rpois(ntot, 2)

    return(data.frame(int1, x1, x2, x3))
  }
  X = simX(N_K)

  # 5. Scale parameters of a JPLP
  # 5a. parameter matrix: P
  P = cbind(r0 = rep(r_0D, N_K), t(replicate(N, R_K)))
  M_logtheta = P*X
  theta = exp(rowSums(M_logtheta))

  # Initialization of lists
  t_shift_vec = list()
  n_trip_vec = list()
  id_trip_vec = list()
  t_start_vec = list()
  t_stop_vec = list()
  n_event_shift_vec = list()
  t_event_vec = list()
```

```r
n_event_trip_vec = list()

for (i in 1:N)
{
  sim_tau = rnorm(1, 10, 1.3)
  n_stop = get_n_stop()
  sim_t_trip = round((1:n_stop)*sim_tau/(n_stop + 1) +
                     rnorm(n_stop, 0, sim_tau*0.15/n_stop), 2)
  t_events = sim_jplp(tau0 = sim_tau,
                      kappa0 = kappa,
                      t_trip0 = sim_t_trip,
                      beta0 = beta,
                      theta0 = theta[i])
  t_shift_vec[[i]] = sim_tau                # end time for each shift
  n_trip_vec[[i]] = n_stop + 1              # number of trips
  id_trip_vec[[i]] = 1:(n_stop + 1)         # index for trips
  t_start_vec[[i]] = c(0, sim_t_trip)       # start time for each trip
  t_stop_vec[[i]]  = c(sim_t_trip, sim_tau) # end time for each trip
  n_event_shift_vec[[i]] = length(t_events) # number of events for each shift
  t_event_vec[[i]] = t_events               # time of SCEs

  # Create a vector of number of SCEs for each trip
  tmp_n_event_trip = rep(NA_integer_, (n_stop + 1))
  for (j in 1:(n_stop + 1)) {
    tmp_n_event_trip[j] = sum(t_events > t_start_vec[[i]][j] &
                              t_events <= t_stop_vec[[i]][j])
  }
  n_event_trip_vec[[i]] = tmp_n_event_trip
}

# shifts data
shift_dt = data.frame(
  driver_id = rep(1:D, N_K),
  shift_id = 1:N,
  start_time = rep(0, N),
  end_time = Reduce(c, t_shift_vec),
  n_trip = Reduce(c, n_trip_vec),
  n_event = Reduce(c, n_event_shift_vec)
)

# trips data set
trip_dt = data.frame(
  driver_id = rep(shift_dt$driver_id, shift_dt$n_trip),
  shift_id = rep(1:N, Reduce(c, n_trip_vec)),
  trip_id = Reduce(c, id_trip_vec),
  t_trip_start = Reduce(c, t_start_vec),
  t_trip_end = Reduce(c, t_stop_vec),
  N_events = Reduce(c, n_event_trip_vec)
)

# TEMPORARY vector: a temporary vector for events per driver
n_event_driver = shift_dt %>%
  group_by(driver_id) %>%
```

```r
    summarise(n_event = sum(n_event)) %>%
    pull(n_event)
  # TEMPORARY vector: a temporary vector for # of trips per driver
  n_trip_driver = trip_dt %>%
    group_by(driver_id) %>%
    summarise(n_trip = length(shift_id)) %>%
    pull(n_trip)

  # events data set
  event_dt = data.frame(
    driver_id = rep(1:D, n_event_driver),
    shift_id = rep(1:N, Reduce(c, n_event_shift_vec)),
    event_time = Reduce(c, t_event_vec)
  )

  stan_dt = list(
    N = nrow(event_dt),
    K = K,
    S = nrow(trip_dt),
    D = D,
    id = rep(1:D, n_trip_driver),
    #driver index, must be an array
    r_trip = trip_dt$trip_id,
    t_trip_start = trip_dt$t_trip_start,
    t_trip_end = trip_dt$t_trip_end,
    event_time = event_dt$event_time,
    group_size = trip_dt$N_events,
    X_predictors = as.matrix(X[rep(row.names(X), shift_dt$n_trip), 2:4])
  )

  stan_jplp_dt_for_plp = list(
    N = nrow(event_dt),
    K = K,
    S = nrow(shift_dt),
    D = D,
    id = rep(1:D, N_K), # driver index at shift level
    tau = shift_dt$end_time,
    event_time = event_dt$event_time,
    group_size = shift_dt$n_event, # the number of events in each shift
    X_predictors = X[,2:4]
  )

  return(list(event_time = event_dt,
              trip_time = trip_dt,
              shift_time = shift_dt,
              stan_dt = stan_dt,
              stan_jplp_dt_for_plp = stan_jplp_dt_for_plp))
}

# Function: pull_use: pull wanted estimates from the posterior distributions
pull_use = function(var = "theta", est_obj = f){
  z = est_obj %>%
    broom::tidy() %>%
```

```
    filter(grepl(var, term))
  return(z)
}
```

Stan code for estimating PLP and JPLP data

This following code chunk demonstrates the Stan code to estimate the parameters of a NHPP with PLP intensity function.

```
functions{
  real nhpp_log(vector t, real beta, real theta, real tau){
    vector[num_elements(t)] loglik_part;
    real loglikelihood;
    for (i in 1:num_elements(t)){
      loglik_part[i] = log(beta) - beta*log(theta) + (beta - 1)*log(t[i]);
    }
    loglikelihood = sum(loglik_part) - (tau/theta)^beta;
    return loglikelihood;
  }
  real nhppnoevent_lp(real tau, real beta, real theta){
    real loglikelihood = - (tau/theta)^beta;
    return(loglikelihood);
  }
}
data {
  int<lower=1> N;                // total # of failures
  int<lower=1> K;                // number of predictors
  int<lower=1> S;                // total # of shifts
  int<lower=1> D;                // total # of drivers
  int<lower=1> id[S];            // driver index, must be an array
  vector<lower=0>[S] tau;        // truncated time
  vector<lower=0>[N] event_time; // failure time
  int group_size[S];             // group sizes
  matrix[S, K] X_predictors;     // predictor variable matrix
}
transformed data{
  matrix[S, K] X_centered;
  vector[K] X_means;
  for(k0 in 1:K){
    X_means[k0] = mean(X_predictors[, k0]);
    X_centered[,k0] = X_predictors[, k0] - X_means[k0];
  }
}
parameters{
  real mu0;             // hyperparameter: mean
  real<lower=0> sigma0; // hyperparameter: s.e.
  real<lower=0> beta;   // shape parameter
  vector[K] R1_K;       // fixed parameters each of K predictors
  vector[D] R0;         // random intercept for each of D drivers
}
model{
  int position = 1;
  vector[S] theta_temp;
```

```
  for (s0 in 1:S){
    theta_temp[s0] = exp(R0[id[s0]] + X_centered[s0,]*R1_K);
  }

  for (s1 in 1:S){
    if(group_size[s1] == 0) {
      target += nhppnoevent_lp(tau[s1], beta, theta_temp[s1]);
    }else{
      segment(event_time, position, group_size[s1]) ~ nhpp(beta, theta_temp[s1], tau[s1]);
      position += group_size[s1];
    }
  }
  beta ~ gamma(1, 1);
  R0 ~ normal(mu0, sigma0);
  R1_K  ~ normal(0, 10);
  mu0 ~ normal(0, 10);
  sigma0 ~ gamma(1, 1);
}
generated quantities{
  real mu0_true = mu0 - dot_product(X_means, R1_K);
  vector[D] R0_true = R0 - dot_product(X_means, R1_K);
  //real theta_correct = theta_temp - dot_product(X_centered, R1_K);
}
```

Different from NHPP with PLP intensity function, in which the likelihood function was evaluated by shifts, this JPLP likelihood function is evaluated by TRIPS, which are nested within shifts. In this way, the likelihood function can be evaluated using the `segment` function in Stan.

```
// Stan code to estimate a hierchical JPLP process
functions{
  // LogLikelihood function for shifts with events (N_{event} > 0)
  real jplp_log(vector t_event, // time of SCEs
                real trip_start,
                real trip_end,
                int r,     // trip index
                real beta,
                real theta,
                real kappa)
  {
    vector[num_elements(t_event)] loglik;
    real loglikelihood;
    for (i in 1:num_elements(t_event))
    {
      loglik[i] = (r - 1)*log(kappa) + log(beta) - beta*log(theta) +
            (beta - 1)*log(t_event[i]);
    }
    loglikelihood = sum(loglik) -
          kappa^(r - 1)*theta^(-beta)*(trip_end^beta - trip_start^beta);
    return loglikelihood;
  }
  // LogLikelihood function for shifts with no event (N_{event} = 0)
  real jplpoevent_lp(real trip_start,
                     real trip_end,
                     int r,
                     real beta,
```

```
                     real theta,
                     real kappa)
  {
    real loglikelihood = - kappa^(r - 1)*theta^(-beta)*(trip_end^beta -
                    trip_start^beta);
    return(loglikelihood);
  }
}
data {
  int<lower=0> N;                  // total # of events
  int<lower=1> D;                  // total # of drivers
  int<lower=1> K;                  // number of predictors
  int<lower=0> S;                  // total # of trips, not shifts!!
  int<lower=1> id[S];              // driver index, must be an array
  int r_trip[S];                   // index of trip $r$
  vector<lower=0>[S] t_trip_start; // trip start time
  vector<lower=0>[S] t_trip_end;   // trip end time
  vector<lower=0>[N] event_time;   // failure time
  int group_size[S];               // group sizes
  matrix[S, K] X_predictors;       // predictor variable matrix
}
transformed data{
  matrix[S, K] X_centered;
  vector[K] X_means;
  for(k0 in 1:K){
    X_means[k0] = mean(X_predictors[, k0]);
    X_centered[,k0] = X_predictors[, k0] - X_means[k0];
  }
}
parameters{
  real mu0;                    // hyperparameter
  real<lower=0> sigma0;        // hyperparameter
  real<lower=0> beta;          // Shape parameter
  real<lower=0, upper=1> kappa; // Jump parameter
  vector[K] R1_K;              // fixed parameters for K predictors
  vector[D] R0;                // random intercept for D drivers
}
model{
  int position = 1;
  vector[S] theta_temp;

  for (s0 in 1:S){
    theta_temp[s0] = exp(R0[id[s0]] + X_centered[s0,]*R1_K);
  }

  for (s1 in 1:S){ // Likelihood estimation for JPLP based on trips, not shifts
    if(group_size[s1] == 0){
      target += jplpoevent_lp(t_trip_start[s1], t_trip_end[s1],
            r_trip[s1], beta, theta_temp[s1], kappa);
      }else{
      segment(event_time, position, group_size[s1]) ~ jplp_log(t_trip_start[s1],
            t_trip_end[s1], r_trip[s1], beta, theta_temp[s1], kappa);
      position += group_size[s1];
    }
```

```
  }
//PRIORS
  beta ~ gamma(1, 1);
  kappa ~ uniform(0, 1);
  R0 ~ normal(mu0, sigma0);
  R1_K  ~ normal(0, 10);
  mu0 ~ normal(0, 10);
  sigma0 ~ gamma(1, 1);
}
generated quantities{
  real mu0_true = mu0 - dot_product(X_means, R1_K);
  vector[D] R0_true = R0 - dot_product(X_means, R1_K);
  //real theta_correct = theta_temp - dot_product(X_centered, R1_K);
}
```

Scale up PLP and JPLP simulation

This following chunk demonstrates how to scale up the JPLP data simulation and JPLP Stan estimation to 1000 simulations.

```r
N_sim = 1000

set.seed(123)
#  D = 10
sim10 = list()
for (i in 1:N_sim) {
  print(paste0("D = 10, progress: ",
               round(i*100/N_sim, 2),
               "% (", i, " out of 1000)"))

  tryCatch({z = sim_hier_JPLP(beta = 1.2, D = 10)
  fit0 = stan("stan/jplp_hierarchical.stan",
              chains = 1, iter = 3000, refresh = 0,
              data = z$stan_dt, seed = 123
  )}, error=function(e){})

  sim10[[i]] = pull_use("beta|kappa|mu0_true|sigma0|R1_K", fit0)
}
data.table::fwrite(data.table::rbindlist(sim10),
                   "fit/JPLP_fit_sim_hierarchical/sim10.csv")

#  D = 25
sim25 = list()
for (i in 1:N_sim) {
  print(paste0("D = 25, progress: ",
               round(i*100/N_sim, 2),
               "% (", i, " out of 1000)"))

  tryCatch({z = sim_hier_JPLP(beta = 1.2, kappa = 0.8,
                              mu0 = 0.2, sigma0 = 0.5,
                              R_K = c(1, 0.3, 0.2), D = 25)},
           error=function(e){})

  tryCatch({fit0 = stan("stan/jplp_hierarchical.stan",
                        chains = 1, iter = 4000, refresh = 0,
```

```r
                        data = z$stan_dt, seed = 123)},
           error=function(e){})

  sim25[[i]] = pull_use("beta|kappa|mu0_true|sigma0|R1_K", fit0)
}
data.table::fwrite(data.table::rbindlist(sim25),
                   "fit/JPLP_fit_sim_hierarchical/sim25.csv")

#  D = 50
sim50 = list()
for (i in 1:N_sim) {
  print(paste0("D = 50, progress: ",
               round(i*100/500, 2),
               "% (", i, " out of 500)"))

  tryCatch({z = sim_hier_JPLP(beta = 1.2, kappa = 0.8,
                              mu0 = 0.2, sigma0 = 0.5,
                              R_K = c(1, 0.3, 0.2), D = 50)},
           error=function(e){})

  tryCatch({fit0 = stan("stan/jplp_hierarchical.stan",
                        chains = 1, iter = 4000, refresh = 0,
                        data = z$stan_dt, seed = 123)},
           error=function(e){})

  sim50[[i]] = pull_use("beta|kappa|mu0_true|sigma0|R1_K", fit0)
}
data.table::fwrite(data.table::rbindlist(sim50),
                   "fit/JPLP_fit_sim_hierarchical/sim50.csv")

#  D = 75
sim75 = list()
for (i in 1:N_sim) {
  print(paste0("D = 75, progress: ",
               round(i*100/N_sim, 2),
               "% (", i, " out of 1000)"))

  tryCatch({z = sim_hier_JPLP(beta = 1.2, kappa = 0.8,
                              mu0 = 0.2, sigma0 = 0.5,
                              R_K = c(1, 0.3, 0.2), D = 75)},
           error=function(e){})

  tryCatch({fit0 = stan("stan/jplp_hierarchical.stan",
                        chains = 1, iter = 4000, refresh = 0,
                        data = z$stan_dt, seed = 123)},
           error=function(e){})

  sim75[[i]] = pull_use("beta|kappa|mu0_true|sigma0|R1_K", fit0)
}
data.table::fwrite(data.table::rbindlist(sim75),
                   "fit/JPLP_fit_sim_hierarchical/sim75.csv")

#  D = 100
```

```r
sim100 = list()
for (i in 1:N_sim) {
  print(paste0("D = 100, progress: ", round(i*100/N_sim, 2),
               "% (", i, " out of 1000)"))

  tryCatch({z = sim_hier_JPLP(beta = 1.2, kappa = 0.8, mu0 = 0.2,
                              sigma0 = 0.5, R_K = c(1, 0.3, 0.2), D = 100)},
           error=function(e){})

  tryCatch({fit0 = stan("stan/jplp_hierarchical.stan",
                        chains = 1, iter = 4000, refresh = 0,
                        data = z$stan_dt, seed = 123)},
           error=function(e){})

  sim100[[i]] = pull_use("beta|kappa|mu0_true|sigma0|R1_K", fit0)
}
data.table::fwrite(data.table::rbindlist(sim100),
                   "fit/JPLP_fit_sim_hierarchical/sim100.csv")
```

Bayesian hierarchical reliability models: real data estimation

```r
# ******************************************************
# *********       NHPP real data estimation     **********

# *********  Run Stan with some simulated data  **********
source('Functions/NHPP_functions.R')
df = sim_hier_nhpp(D = 5, beta = 1.2)
fit0 = stan("Stan/nhpp_plp_hierarchical.stan",
            chains = 1, iter = 1000, data = df$hier_dat, refresh = 1)
broom::tidy(fit0)

# ******************   Read in data    *******************
dnhpp = as.data.table(read_fst('Data/dnhpp.fst'))
djplp = as.data.table(read_fst('Data/djplp.fst'))
sce = read_fst('Data/sce.fst') %>%
  dplyr::select(driver_id, shift_id_num, trip_id_num, t_trip_start,
                t_trip_end, T2SCE_trip, event_type) %>%
  left_join(djplp[,.(driver_id, shift_id_num, trip_id_num, tau)],
            by = c('driver_id', 'shift_id_num', 'trip_id_num')) %>%
  mutate(T2SCE = T2SCE_trip + t_trip_start) %>%
  mutate(T2SCE = fifelse(T2SCE == 0, 0.1, T2SCE)) %>%
  as.data.table()

# ************  Create a list data for stan  *************
dt_nhpp = list(
  N = sce[,.N],
  K = 9,
  S = dnhpp[,.N],
  D = dnhpp[,.N,driver_id][,.N],
  id = dnhpp[,driver_id],
  tau = dnhpp[,tau/60],
  event_time = sce[,T2SCE/60],
  group_size = dnhpp[,N_SCE],
```

```
  X_predictors = dnhpp[,.(age, Black, Other_Race, Female, speed_mean,
                          speed_sd, prep_inten,  prep_prob, wind_speed)]
)

# *********  Run Stan with real data  **********
start_time = Sys.time()
fit_NHPP = stan("Stan/nhpp_plp_hierarchical.stan", data = dt_nhpp, seed = 123,
                chains = 4, cores = 4, iter = 5000, warmup = 1000, refresh = 1)
(Time_diff = Sys.time() - start_time)
broom::tidy(fit_NHPP)
saveRDS(fit_NHPP, 'Fit/fit_NHPP.rds')
```

```
# *****************************************************
# *********       JPLP real data estimation     **********
# *********  Run Stan with some simulated data  **********
source('Functions/JPLP_functions.R')
sim_df = sim_hier_JPLP(D = 10, beta = 1.2)
fit0 = stan("Stan/jplp_hierarchical.stan",
            chains = 1, iter = 1000, data = sim_df$stan_dt, refresh = 1)
broom::tidy(fit0)

# ******************   Read in data    ******************
djplp = as.data.table(read_fst('Data/djplp.fst'))
sce = read_fst('Data/sce.fst') %>%
  dplyr::select(driver_id, shift_id_num, trip_id_num, t_trip_start,
                t_trip_end, T2SCE_trip, event_type) %>%
  mutate(T2SCE = t_trip_start + T2SCE_trip) %>%
  mutate(T2SCE = fifelse(T2SCE == 0, 0.1, T2SCE)) %>%
  as.data.table()

# ************  Create a list data for stan  *************
dt_JPLP = list(
  N = sce[,.N],
  K = 9,
  S = djplp[,.N],
  D = djplp[,.N,driver_id][,.N],
  id = djplp[,driver_id],
  r_trip = djplp[,trip_id_num],
  t_trip_start = djplp[,t_trip_start/60],
  t_trip_end = djplp[,t_trip_end/60],
  event_time = sce[,T2SCE/60],
  group_size = djplp[,N_SCE],
  X_predictors = djplp[,.(age, Black, Other_Race, Female, speed_mean,
                          speed_sd, prep_inten,  prep_prob, wind_speed)]
)

# *********  Run Stan with real data  **********
nchain = 4
n_iter = 5000

start_time = Sys.time()
fit_JPLP = stan("Stan/jplp_hierarchical.stan", data = dt_JPLP, seed = 123,
                chains = nchain, cores = nchain, iter = n_iter,
                warmup = 1000, refresh = 1)
```

```r
(Time_diff = Sys.time() - start_time)
broom::tidy(fit_JPLP)
saveRDS(fit_JPLP, 'Fit/fit_JPLP.rds')
```