Online Supplement

Probability of ruin within finite time and Cramér–Lundberg inequality for fractional risk processes

Nikolai Leonenko, Andrey Pepelyshev, Alois Pichler,
Enrica Pirozzi, Xiangyun Meng

Simulation of the compound fractional risk process

The algorithm for testing the condition $R^{\bullet}_{\alpha}(t) < 0$ for $t \in [0, T]$ in the R package can be written as follows.

```
# Input: u, c, lambda, mu, alpha, T, Delta
# Output: IsRuined=TRUE if surplus is negative before time T

# Initialize the ruin indicator to FALSE.
IsRuined = FALSE

# Initialize the total amount paid in claims to 0.
CurrentTotalPay = 0

# Initialize the current time in the inverse subordinator (Yt) to 0.
NextClaimYt = 0
CurrentYt = 0

# Initialize the original time (t) to 0.
Current.t = 0

# The main simulation loop continues as long as the original time is
# within the specified horizon T and ruin has not occurred.
while (Current.t <= T && !IsRuined) {

  # Simulate the next claim arrival time in the classical risk process (gapPP)
  # using an exponential distribution with rate lambda (lam).
  gapPP = rexp(1, lam)

  # Update the time of the next claim in the inverse subordinator's clock.
  NextClaimYt = NextClaimYt + gapPP

  # Initialize a temporary time step for the inverse subordinator.
  Delta0 = Delta
```

```
  # This inner loop simulates the inverse alpha-stable subordinator's
  # trajectory in small steps to reach the next claim time.
  while(CurrentYt < NextClaimYt) {

    # If the remaining distance to the next claim is smaller than the
    # standard time step Delta, adjust the step size to precisely
    # hit the next claim time.
    if(Delta > NextClaimYt - CurrentYt)
       Delta0 = NextClaimYt - CurrentYt

    # Generate two uniform random numbers, U and V, to be used in the
    # Chambers-Mallows-Stuck algorithm for simulating an alpha-stable
    # random variable.
    U = runif(1)
    V = runif(1)

    # Calculate the numerator of the alpha-stable random variable formula.
    numer = sin(alpha * pi * U) * (sin((1 - alpha) * pi * U))^(1/alpha - 1)

    # Calculate the denominator of the alpha-stable random variable formula.
    denom = (sin(pi * U))^(1/alpha) * (abs(log(V)))^(1/alpha - 1)

    # Calculate D1, a positive alpha-stable random variable used to scale
    # the time step in the subordinator.
    D1 = numer / denom

    # Update the original time (t) based on the current step
    # in the inverse subordinator's time.
    Current.t = Current.t + D1 * Delta0^(1/alpha)

    # Advance the time in the subordinator's clock by the step size.
    CurrentYt = CurrentYt + Delta0
  }

  # Simulate the size of the claim using an exponential distribution
  # with mean mu.
  CurrentTotalPay = CurrentTotalPay + rexp(1, 1 / mu)

  # Calculate the current surplus based on the initial capital (u),
  # premium income (c * CurrentYt), and total claims paid.
  CurrentSurplus = u + c * CurrentYt - CurrentTotalPay

  # Check if the ruin condition is met: the surplus is negative before
  # or at the end of the simulation horizon T.
  if (Current.t <= T && CurrentSurplus < 0) {
    IsRuined = TRUE
  }
}

# Return the boolean value indicating whether ruin occurred.
return(IsRuined)
```

Plotting of the cdf of the Mittag-Lefler distribution

The cdf of the Mittag-Lefler distribution in the R package can be depicted as follows.

```
library(MittagLeffleR)

# ----- Parameters you can adjust -----
alpha_values <- c(1.0, 0.9, 0.7, 0.5, 0.3) # Shape parameters a to compare
curve_colors <- c('black','red','green','blue','orange')
line_width   <- 2

# Grid on the positive support
t_min    <- 0
t_max    <- 10
n_points <- 1000L
t_grid   <- seq(t_min, t_max, length.out = n_points)

# ----- CDF of the one-parameter Mittag-Leffler distribution -----
# Uses: F(t; a) = 1 - E_{a}(-t^{a}), where E_{a} = E_{a,1}
cdf_mittag_leffler <- function(t, alpha) {
  1 - mlf(-(t^alpha), alpha)
}

# ----- Open a PDF device  -----
pdf("ruin-prob-ml.pdf", width = 7, height = 3)

# Tidy up margins and axis label spacing
par(mar = c(3, 3, 2, 1), mgp = c(1.8, 0.6, 0))

# Initialize an empty plot with proper limits for a CDF
plot(
  NA, xlim = c(t_min, t_max), ylim = c(0, 1.02),
  xlab = "t", ylab = "CDF␣␣F(t;␣a)",
  main = "Mittag-Leffler␣CDF:␣␣F(t)␣=␣1␣-␣E_a(-t^a)",
  xaxs = "i", yaxs = "i"
)

# Draw CDF curves for each a
for (i in seq_along(alpha_values)) {
  alpha_i <- alpha_values[i]
  F_vals  <- cdf_mittag_leffler(t_grid, alpha_i)
  lines(t_grid, F_vals, col = curve_colors[i], lwd = line_width)
}

# Legend
legend("bottomright",
       legend = sprintf("a␣=␣%g", alpha_values),
       col = curve_colors, lwd = line_width, bty = "n", title = "Shape")

# Close the device
dev.off()
```

Plotting of the finite-time ruin probability

The finite-time ruin probability $\psi^{\alpha}(t)$ for the compound fractional risk model in the R package can be depicted as follows.

```
library(MittagLeffleR)

# ------------------------------
# Integrand for the ruin probability integral
# ------------------------------
ruin_integrand <- function(x, t, u, lambda_rate, mu_mean, premium_rate, alpha) {
  # x can be a vector; all others are scalars
  stopifnot(is.numeric(x), length(t) == 1L, length(u) == 1L,
            length(lambda_rate) == 1L, length(mu_mean) == 1L,
            length(premium_rate) == 1L, length(alpha) == 1L)

  beta <- (lambda_rate * mu_mean) / premium_rate
  u_over_mu <- u / mu_mean
  sqrt_beta <- sqrt(beta)

  # Components of the integrand
  sb_cosx <- sqrt_beta * cos(x)
  # z for the Mittag-Leffler function E_{a,1}(z)
  z <- (2 * sb_cosx - (1 + beta)) * (premium_rate / mu_mean) * (t^alpha)

  # E_{a,1}(z)
  Ea1 <- mlf(z, alpha)  # MittagLeffleR::mlf(x, alpha) = E_{a,1}(x)

  expo <- beta * exp(u_over_mu * (sb_cosx - 1))
  u_sb_sinx <- u_over_mu * sqrt_beta * sin(x)

  num <- Ea1 * expo * (cos(u_sb_sinx) - cos(u_sb_sinx + 2 * x))
  den <- (1 + beta - 2 * sb_cosx)

  # Prevent numerical blow-ups near the singular set by softening very small |den|
  # (this leaves the limit behavior well-behaved for numerical integration)
  den_safe <- ifelse(abs(den) < 1e-12, sign(den) * 1e-12, den)

  num / den_safe
}

# ------------------------------
# Single-time finite-time ruin probability
# ------------------------------
ruin_prob_single <- function(t, u, lambda_rate, mu_mean, premium_rate, alpha,
                             rel.tol = 1e-8, abs.tol = 1e-10, subdivisions = 2000L) {
  stopifnot(t >= 0, u > 0, lambda_rate > 0, mu_mean > 0, premium_rate > 0,
            alpha > 0, alpha <= 1)

  beta <- (lambda_rate * mu_mean) / premium_rate
  p_infty <- beta * exp(-(1 - beta) * (u / mu_mean))  # infinite-horizon term

  I <- integrate(
    f            = ruin_integrand,
    lower        = 0, upper = pi,
    t            = t, u = u,
    lambda_rate  = lambda_rate,
```

```
    mu_mean      = mu_mean,
    premium_rate = premium_rate,
    alpha        = alpha,
    rel.tol      = rel.tol,
    abs.tol      = abs.tol,
    subdivisions = subdivisions,
    stop.on.error = FALSE
  )

  if (!is.null(I$message) && grepl("Error", I$message, fixed = TRUE)) {
    warning(sprintf(
        "Integration warning at t=%.4g, a=%.3f: %s", t, alpha, I$message))
  }

  p_infty - (I$value / pi)
}

# Vectorized wrapper over a grid of times
ruin_prob_vec <- function(t_grid, u, lambda_rate, mu_mean, premium_rate, alpha,
                    rel.tol = 1e-8, abs.tol = 1e-10, subdivisions = 2000L) {
  vapply(
    t_grid,
    function(tt) ruin_prob_single(
      t = tt, u = u, lambda_rate = lambda_rate, mu_mean = mu_mean,
      premium_rate = premium_rate, alpha = alpha,
      rel.tol = rel.tol, abs.tol = abs.tol, subdivisions = subdivisions
    ),
    numeric(1L)
  )
}

# ------------------------------
# Plotting function
# ------------------------------
make_ruin_plot <- function(
    file_name,
    t_grid,
    u,
    lambda_rate,
    mu_mean,
    premium_rate,
    alpha_values = c(1.0, 0.9, 0.7, 0.5, 0.3),
    y_lower_scale = 0.8,    # lower y bound fraction of the max across curves
    curve_colors  = c("black", "red", "green", "blue", "orange"),
    line_width    = 2
) {
  stopifnot(length(alpha_values) <= length(curve_colors))

  # Precompute curves so we can set a consistent y-range
  curves <- lapply(alpha_values, function(a)
    ruin_prob_vec(t_grid, u, lambda_rate, mu_mean, premium_rate, alpha = a)
  )
  y_max <- max(vapply(curves, max, numeric(1), na.rm = TRUE))
  ylim  <- c(y_lower_scale * y_max, 1.00 * y_max)

  # Open device
  pdf(file_name, width = 7, height = 3)
  on.exit(dev.off(), add = TRUE)
```

```
  par(mar = c(3, 3, 2.5, 6), xpd = TRUE, mgp = c(1.8, 0.6, 0))

  # Draw first curve
  plot(
    t_grid, curves[[1]],
    type = "l", lwd = line_width, col = curve_colors[1],
    xlab = "t", ylab = "Ruin probability",
    ylim = ylim, xaxs = "i", yaxs = "i",
    main = "CFR-model finite-time ruin probability"
  )

  # Add remaining curves
  if (length(alpha_values) > 1) {
    for (i in 2:length(alpha_values)) {
      lines(t_grid, curves[[i]], col = curve_colors[i], lwd = line_width)
    }
  }

  # Legend on the right
  legend(
    "right",
    legend = sprintf("a = %g", alpha_values),
    inset = -0.2, lwd = line_width, col = curve_colors,
    title = "alpha", bty = "n"
  )

  # Parameter annotation
  beta <- (lambda_rate * mu_mean) / premium_rate
  mtext(
    sprintf("u = %g    lambda = %g    mu = %g    c = %g    beta = %.4f",
            u, lambda_rate, mu_mean, premium_rate, beta),
    side = 3, line = -0.05, cex = 0.9
  )
}

# Time grid (same structure as your original)
t_grid <- sort(c(seq(1, 2, by = 0.1), seq(2.5, 10, by = 0.5)))

# Common parameters
lambda_rate <- 45
mu_mean     <- 50

# 1) Baseline
u            <- 250
premium_rate <- 3000
make_ruin_plot("ruin-prob-ml-ft.pdf",  t_grid,
       u, lambda_rate, mu_mean, premium_rate)

# 2) Different initial reserve u
u <- 100
make_ruin_plot("ruin-prob-ml-ft2.pdf", t_grid,
      u, lambda_rate, mu_mean, premium_rate, y_lower_scale = 0.89)

# 3) Different premium rate
u            <- 250
premium_rate <- 2500
make_ruin_plot("ruin-prob-ml-ft3.pdf", t_grid,
       u, lambda_rate, mu_mean, premium_rate, y_lower_scale = 0.60)
```

Plotting of the density of the inverse subordinator

The density of $Y_\alpha(1)$ in the R package can be depicted as follows.

```
library(MWright)

# ------------------------------
# 1) Series kernel g_a(z) for a single z
# ------------------------------
g_alpha_single <- function(z, alpha, n_terms = 120L) {
  # z: positive scalar
  # alpha: in (0,1]
  # n_terms: number of series terms

  if (!is.numeric(z) || length(z) != 1L || z <= 0) {
    stop("g_alpha_single() expects a single positive z.")
  }
  if (!(alpha > 0 && alpha <= 1)) {
    stop("alpha must be in (0,1].")
  }

  k <- seq_len(n_terms)

  # Use lgamma to prevent overflow in gamma(alpha*k+1)/gamma(k+1)
  coeff <- exp(lgamma(alpha * k + 1) - lgamma(k + 1))
  signs <- (-1)^(k + 1)
  terms <- signs * coeff * sin(pi * alpha * k) / (z^(alpha * k + 1))

  sum(terms) / pi
}

# ------------------------------
# 2) Inverse a-stable density via the g_a kernel
# ------------------------------
dinv_alpha_stable <- function(x, alpha, n_terms = 120L) {
  # Vectorized in x.
  # For x <= 0, the density is 0 on this support.
  if (!(alpha > 0 && alpha <= 1)) {
    stop("alpha must be in (0,1].")
  }

  inv_alpha <- 1 / alpha
  out <- numeric(length(x))

  positive_idx <- which(x > 0)
  if (length(positive_idx) > 0) {
    x_pos <- x[positive_idx]
    # z = 1 / x^{1/alpha}
    z_vals <- x_pos^(-inv_alpha)
    g_vals <- sapply(z_vals, g_alpha_single, alpha = alpha, n_terms = n_terms)
    out[positive_idx] <- inv_alpha * g_vals / (x_pos^(1 + inv_alpha))
  }

  out
}

# ------------------------------
```

```r
# Common plotting parameters
# ------------------------------
alpha_values <- c(0.95, 0.9, 0.7, 0.5, 0.3)
curve_colors <- c("black", "red", "green", "blue", "orange")
line_width   <- 2

# x-grid used in both figures
x_min    <- 1 / 200              # = 0.005 (matches your original grid start)
x_max    <- 5
n_points <- 1000L
x_grid   <- seq(x_min, x_max, length.out = n_points)

# ------------------------------
# FIGURE 1: Inverse a-stable density via series
# ------------------------------
# Precompute all curves to set a consistent ylim
ias_curves <- lapply(alpha_values,
        function(a) dinv_alpha_stable(x_grid, a, n_terms = 120L))
ias_ymax   <- max(vapply(ias_curves, max, numeric(1), na.rm = TRUE))
ylim_ias   <- c(0, max(3.2, 1.05 * ias_ymax))

pdf("dens-ias.pdf", width = 7, height = 3)
par(mar = c(3, 3, 2, 1), mgp = c(1.8, 0.6, 0))  # margins and axis spacing

# Draw the first curve
plot(
  x_grid, ias_curves[[1]],
  type = "l", lwd = line_width, col = curve_colors[1],
  xlab = "x", ylab = "Density",
  main = "Inverse␣a-stable␣density␣(series␣approximation)",
  ylim = ylim_ias, xaxs = "i", yaxs = "i"
)

# Add the rest
for (i in 2:length(alpha_values)) {
  lines(x_grid, ias_curves[[i]], col = curve_colors[i], lwd = line_width)
}

legend("topright",
       legend = sprintf("a␣=␣%g", alpha_values),
       col = curve_colors, lwd = line_width, bty = "n", title = "Shape")
dev.off()

# ------------------------------
# FIGURE 2: Inverse a-stable density via MWright::dmwright1
# ------------------------------
# dmwright1(alpha, scale, n_points, x_max) returns a 2-col matrix [x, f(x)]
mw_curves <- lapply(alpha_values, function(a) {
  xy <- dmwright1(a, 1, n_points, x_max)
  # Ensure it's a 2-column object: x in col 1, density in col 2
  as.matrix(xy)
})

mw_ymax <- max(vapply(mw_curves,
                function(m) max(m[, 2], na.rm = TRUE), numeric(1)))
ylim_mw <- c(0, max(3.2, 1.05 * mw_ymax))

pdf("dens-fmwf.pdf", width = 7, height = 3)
par(mar = c(3, 3, 2, 1), mgp = c(1.8, 0.6, 0))
```

```
# First curve
plot(
  mw_curves[[1]][, 1], mw_curves[[1]][, 2],
  type = "l", lwd = line_width, col = curve_colors[1],
  xlab = "x", ylab = "Density",
  main = "Inverse a-stable density via (MWright::dmwright1)",
  ylim = ylim_mw, xaxs = "i", yaxs = "i"
)

# Remaining curves
for (i in 2:length(alpha_values)) {
  lines(mw_curves[[i]][, 1], mw_curves[[i]][, 2],
        col = curve_colors[i], lwd = line_width)
}

legend("topright",
       legend = sprintf("a = %g", alpha_values),
       col = curve_colors, lwd = line_width, bty = "n", title = "Shape")
dev.off()
```

Plotting trajectories of the inverse subordinator

The random trajectories of $Y_\alpha(t)$ in the R package can be depicted as follows.

```
r_pos_stable <- function(n, alpha, u_eps = 1e-12) {
  stopifnot(is.numeric(n), n >= 1, is.finite(n),
            is.numeric(alpha), length(alpha) == 1L, alpha > 0, alpha < 1)

  inv_alpha <- 1 / alpha
  # Avoid U exactly at {0, pi} to prevent sin() under/overflow
  U <- runif(n, min = u_eps, max = pi - u_eps)  # U in (0, pi)
  E <- rexp(n)                                  # E ~ Exp(1)

  num  <- sin(alpha * U) * (sin((1 - alpha) * U))^(inv_alpha - 1)
  den  <- (sin(U))^inv_alpha * (E)^(inv_alpha - 1)

  x <- num / den
  # Theoretically X > 0; clamp any tiny negatives due to floating error
  pmax(x, 0)
}

# ------------------------------
# Simulate one a-stable subordinator path D(s)
# ------------------------------
# Discrete grid s_k = k * dt, k = 0, 1, ..., n_steps-1 (length n_steps).
# D(0) = 0, and increments are scaled as dt^{1/a}.
simulate_subordinator_path <- function(dt, n_steps, alpha) {
  stopifnot(is.numeric(dt), dt > 0, n_steps >= 2)
  # n_steps points => (n_steps - 1) increments
  inc <- dt^(1 / alpha) * r_pos_stable(n_steps - 1, alpha)
  c(0, cumsum(inc))
}

# ------------------------------
# Make multi-page PDF: one page per a, with multiple sample paths
# ------------------------------
make_inv_subordinator_plots <- function(
    file_name         = "inverse-stable-subordinator.pdf",
    alpha_values      = c(0.99, 0.95, 0.90, 0.70),
    dt                = 0.01,      # time step for the subordinator's clock s
    s_horizon         = 2.0,       # max "operational time" s
    n_paths_per_alpha = 50,        # how many paths per a (plus a base one)
    seed_per_alpha    = 137,       # set.seed() before each a (reproducible pages)
    xlim_fixed        = NULL,      # e.g. c(0, 6); if NULL, determined per page
    base_path_lwd     = 2,         # thicker line for the first path
    other_paths_lwd   = 1,         # thinner lines for the remaining paths
    col_palette_fun   = function(n) grDevices::hcl.colors(n, "Dark␣3") # colors
) {
  stopifnot(is.character(file_name), length(file_name) == 1L)
  stopifnot(all(alpha_values > 0 & alpha_values < 1))
  stopifnot(dt > 0, s_horizon > 0, n_paths_per_alpha >= 1)

  n_steps <- as.integer(round(s_horizon / dt)) + 1L
  s_grid  <- dt * (0:(n_steps - 1))   # "physical" time for the inverse (y-axis)

  # Open PDF device
  pdf(file_name, width = 7, height = 3)
```

```
  on.exit(dev.off(), add = TRUE)
  par(mar = c(3, 3, 2.2, 1), mgp = c(1.8, 0.6, 0))

  for (alpha in alpha_values) {
    # Reproducibility per page (same RNG state across a runs)
    if (!is.null(seed_per_alpha)) set.seed(seed_per_alpha)

    # Colors for paths (first path highlighted)
    cols <- col_palette_fun(n_paths_per_alpha + 1L)
    cols[1] <- "black"  # first path in black

    # Simulate first (base) path
    D_base <- simulate_subordinator_path(dt, n_steps, alpha)

    # Optionally determine x-limits per page, based on a small pilot set
    if (is.null(xlim_fixed)) {
      D_max <- max(D_base)
      # simulate a few more to get a robust x-maximum (without drawing yet)
      pilot <- 10L
      for (i in seq_len(min(pilot, n_paths_per_alpha))) {
        D_tmp <- simulate_subordinator_path(dt, n_steps, alpha)
        D_max <- max(D_max, D_tmp)
      }
      xlim_use <- c(0, D_max * 1.05)
    } else {
      xlim_use <- xlim_fixed
    }

    # --- Draw base path (as inverse: x = D(s), y = s) ---
    plot(
      x = D_base, y = s_grid, type = "s",
      xlim = xlim_use, ylim = c(0, s_horizon),
      xlab = expression(D(s)), ylab = "s",
      main = bquote("Inverse␣alpha-stable␣subordinator"),
      col = cols[1], lwd = base_path_lwd, xaxs = "i", yaxs = "i"
    )

    # Draw additional paths
    for (i in seq_len(n_paths_per_alpha)) {
      D_i <- simulate_subordinator_path(dt, n_steps, alpha)
      lines(D_i, s_grid, type = "s",
            col = cols[i + 1L], lwd = other_paths_lwd)
    }

    # Annotate a on the page
    mtext(bquote(alpha == .(alpha)), side = 3, line = 0.1, adj = 0)
  }
}

make_inv_subordinator_plots(
  file_name         = "invsubord.pdf",
  alpha_values      = c(0.99, 0.95, 0.90, 0.70),
  dt                = 0.01,
  s_horizon         = 2.0,
  n_paths_per_alpha = 50,
  seed_per_alpha    = 137,
  xlim_fixed        = c(0, 6)  # set to NULL for auto-scaling per a
)
```