Willoughby ectoparasite model

Anna Willoughby and John Drake

2025-05-22

```r
knitr::opts_chunk$set(echo = TRUE, cache=TRUE)
library('deSolve') # solving ordinary differential equations
library(dplyr) # data wrangling
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(ggplot2) # pretty plots
set.seed(1993)
```

Model

In this document we seek to use a birth-death-immigration process to model the number of ectoparasites on a host.A full description of the model is at https://github.com/DrakeLab/willoughby-grooming-model.

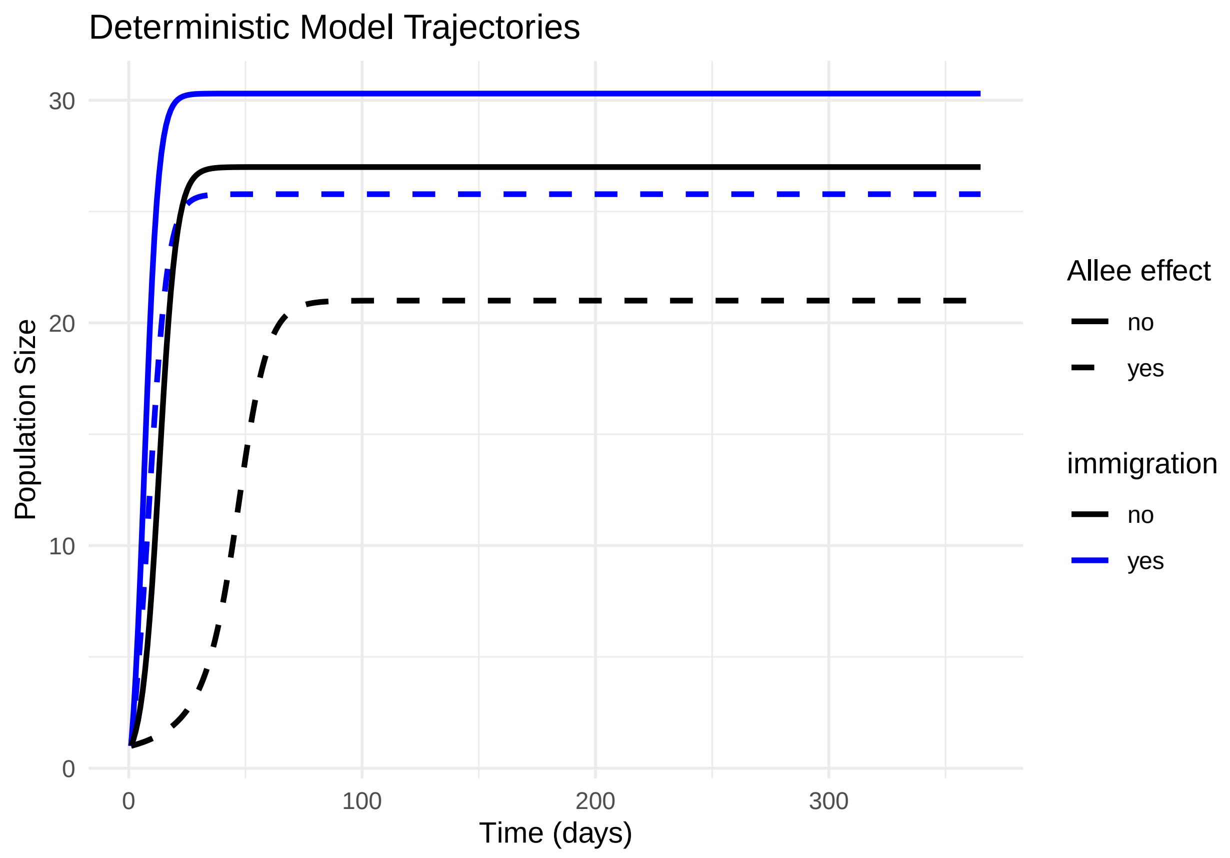
We start by writing functions for the birth rate, death rate, and immigration rate. In all cases, we let the state variable (x) be the first parameter. We also set some default parameters to make things simpler later.

```r
birthrate <- function(x, b0, theta) b0*x*(x/(theta+x)) # function for birth rate with Allee effect (the
deathrate <- function(x, d0, d1) (d0+d1*x)*x # function for death rate (quadratic right now?)
immigrationrate <- function (x, iota) iota # function for immigration rate
# set initial conditions parameters: birth rate, allee effect, small death rate, death rate at high pop
# birth rate is standard across all scenarios, right now as 100 flea births per year (0.27 per day)
# set for each:
## Basic Model (0)
parms0 <-  c(b0=0.27, theta=0, d0=0, d1=0.01, iota=0, initial_pop = 2) # no allee, no immigration
## Allee effect model (1)
parms1 <-  c(b0=0.27, theta=6, d0=0, d1=0.01, iota=0, initial_pop = 2) # no immigration
## Resources (2)
parms2 <-  c(b0=0.27, theta=0, d0=0, d1=0.01, iota=1, initial_pop = 2) # no allee
## Multiple Effects Model (3)
parms3 <-  c(b0=0.27, theta=6, d0=0, d1=0.01, iota=1, initial_pop = 2)

# make a list of the different model scenarios
model_scenarios <- list(parms0, parms1, parms2, parms3)
```

Now, we study the deterministic model numerically, as this will help to choose parameter values.

```r
# define the function for population growth rate
f <- function(t, x, parms){
  dx <- birthrate(x, parms[1], parms[2]) -
    deathrate(x, parms[3], parms[4]) +
    immigrationrate (x, parms[5])

  return(list(dx))
}

# set initial conditions: start with one individual and run through all model scenarios
## Model 0
solution0 <- as.data.frame(ode(y=1, times=seq(1:365), func=f, parms=parms0))
solution0$model <- "zero"
solution0$Allee = "no"
solution0$immigration = "no"
## Model 1
solution1 <- as.data.frame(ode(y=1, times=seq(1:365), func=f, parms=parms1))
solution1$model <- "one"
solution1$Allee = "yes"
solution1$immigration = "no"
## Model 2
solution2 <- as.data.frame(ode(y=1, times=seq(1:365), func=f, parms=parms2))
solution2$model <- "two"
solution2$Allee = "no"
solution2$immigration = "yes"
## Model 3
solution3 <-  as.data.frame(ode(y=1, times=seq(1:365), func=f, parms=parms3))
solution3$model <- "three"
solution3$Allee = "yes"
solution3$immigration = "yes"

# Combine all solutions into one dataframe
solutions <- bind_rows(solution0, solution1, solution2, solution3)

# Rename columns for clarity
colnames(solutions) <- c("time", "population", "model", "Allee", "immigration")

# Plot using ggplot
ggplot(solutions, aes(x = time, y = population, linetype = Allee, color = immigration, group = model))
  geom_line(size = 1) +  # Add lines
  theme_minimal() +  # Clean theme
  labs(title = "Deterministic Model Trajectories",
       x = "Time (days)",
       y = "Population Size",
       linetype = "Allee effect",
       color = "immigration") +
  scale_linetype_manual(values = c("solid", "dashed")) +  # Different line types
  scale_color_manual(values = c("black", "blue"))  # Custom colors
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```



solve for the carrying capacity, and max pop

```
# when does f = 0 ?

# Define the equation to solve # [can prop simplify with f above....]
equilibrium_eq <- function(x, parms) {
  with(as.list(parms), {
    birthrate <- b0*x*(x/(theta+x))
    deathrate <- (d0 + d1*x) * x
    immigrationrate <- iota
    return(birthrate - deathrate + immigrationrate)
  })
}

# Function to find equilibrium population
max_pop_solve <- function(parms) {
  root <- tryCatch(
    uniroot(function(x) equilibrium_eq(x, parms), c(0, 1000), tol = 1e-6)$root,
    error = function(e) NA  # Return NA if uniroot fails
  )
  return(root)
}
# Solve for maximum population for each scenario
max_pops <- lapply(model_scenarios, max_pop_solve)

# Print results
```

```
names(max_pops) <- c("Basic Model", "Allee Effect Model", "Resources Model", "Multiple Effects Model")
print(max_pops)
```

```
## $`Basic Model`
## [1] NA
##
## $`Allee Effect Model`
## [1] 0
##
## $`Resources Model`
## [1] NA
##
## $`Multiple Effects Model`
## [1] 25.78144
```

Stochastic Model

Now, we want to simulate a birth-death-immigration process. We start with some hyperparameters of the simulation.

```
num_simulations <- 20  # Number of times to run the simulation
# max.events <- 1000 # maximum events
max.time <- 730 # 2 year max for a squirrel
```

Here we simulate the process

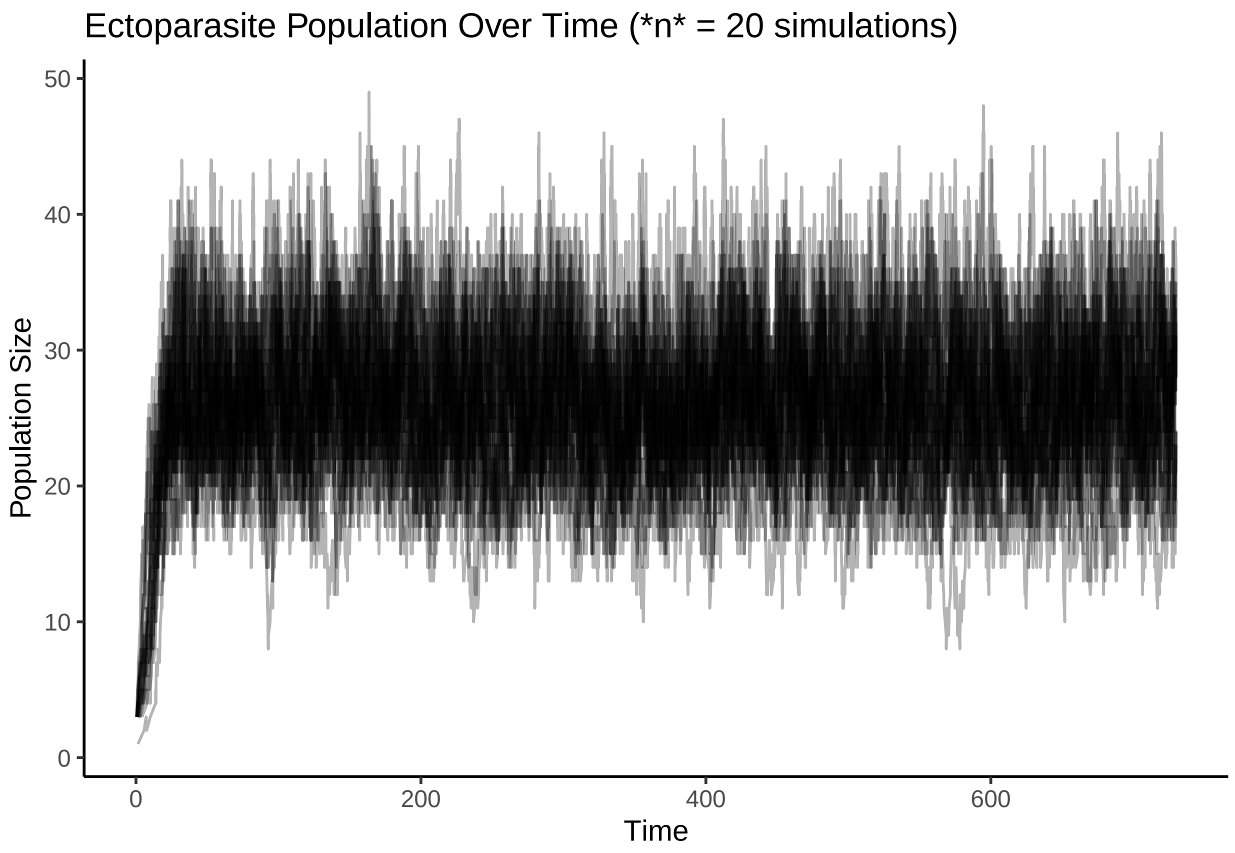
```
num_simulations <- 20  # Define number of simulations
max.time <- 730  # Define max time

# 0: Run basic model
simulation0 <- vector("list", num_simulations)  # Initialize storage

  for (sim in 1:num_simulations) {
    events <- 0  # Initialize the total number of events
    time <- 0  # Initialize current time
    x <- parms0["initial_pop"]  # Start population at initial value

    # Dataframe to store results
    simulation_results <- data.frame(event = numeric(),
                                     time = numeric(),
                                     population = numeric(),
                                     event_type = character(),
                                     sim_id = numeric())

    while(time[length(time)] <= max.time) {
      # Track number of events
      events <- events + 1
      current.size <- x[length(x)]  # Use last recorded population size
      current.time <- time[length(time)]  # Use last recorded time

      # Calculate event rates with proper parameters
      b <- birthrate(x=current.size, b0=parms0["b0"], theta=parms0["theta"])
      d <- deathrate(x=current.size, d0=parms0["d0"], d1=parms0["d1"])
      i <- immigrationrate(x=current.size, iota=parms0["iota"])
      # Compute total rate and time to next event
      total.rate <- b + d + i
      if (total.rate == 0 || is.na(total.rate)) {
        break  # Stop simulation if no events can occur
        }
      increment.time <- -log(1 - runif(1)) / total.rate  # Exponential waiting time

      # Determine event type
      event_probs <- c(d, b, i) / total.rate
      event_outcome <- sample(c("death", "birth", "immigration"), size=1, prob=event_probs)
      change <- ifelse(event_outcome == "death", -1, ifelse(event_outcome == "birth", 1, 0))  # Immigra

      # Store results
      simulation_results <- rbind(simulation_results, data.frame(
        event = events,
        time = current.time + increment.time,
        population = max(0, current.size + change),  # Ensure population doesn't go negative
        event_type = event_outcome,
        sim_id = sim
      ))

      # Update state variables
      time <- c(time, current.time + increment.time)
      x <- c(x, max(0, current.size + change))  # Prevent negative population
    }

    # Store results of this simulation
    simulation0[[sim]] <- simulation_results
  }

# Save all results
saveRDS(simulation0, "data/simulation0.Rdata")

# 1: run Allee effect model
simulation1 <- vector("list", num_simulations)  # Initialize storage

  for (sim in 1:num_simulations) {
    events <- 0  # Initialize the total number of events
    time <- 0  # Initialize current time
    x <- parms1["initial_pop"]  # Start population at initial value

    # Dataframe to store results
    simulation_results <- data.frame(event = numeric(),
                                     time = numeric(),
                                     population = numeric(),
                                     event_type = character(),
                                     sim_id = numeric())

    while(time[length(time)] <= max.time) {
      # Track number of events
      events <- events + 1
      current.size <- x[length(x)]  # Use last recorded population size
      current.time <- time[length(time)]  # Use last recorded time
```

```r
      # Calculate event rates with proper parameters
      b <- birthrate(x=current.size, b0=parms1["b0"], theta=parms1["theta"])
      d <- deathrate(x=current.size, d0=parms1["d0"], d1=parms1["d1"])
      i <- immigrationrate(x=current.size, iota=parms1["iota"])

      # Compute total rate and time to next event
      total.rate <- b + d + i
      if (total.rate == 0 || is.na(total.rate)) {
        break  # Stop simulation if no events can occur
        }
      increment.time <- -log(1 - runif(1)) / total.rate  # Exponential waiting time

      # Determine event type
      event_probs <- c(d, b, i) / total.rate
      event_outcome <- sample(c("death", "birth", "immigration"), size=1, prob=event_probs)
      change <- ifelse(event_outcome == "death", -1, ifelse(event_outcome == "birth", 1, 0))  # Immigra

      # Store results
      simulation_results <- rbind(simulation_results, data.frame(
        event = events,
        time = current.time + increment.time,
        population = max(0, current.size + change),  # Ensure population doesn't go negative
        event_type = event_outcome,
        sim_id = sim
      ))

      # Update state variables
      time <- c(time, current.time + increment.time)
      x <- c(x, max(0, current.size + change))  # Prevent negative population
    }

    # Store results of this simulation
    simulation1[[sim]] <- simulation_results
  }

  # Save all results
  saveRDS(simulation1, "data/simulation1.Rdata")

# 2: Run resources model
simulation2 <- vector("list", num_simulations)  # Initialize storage

  for (sim in 1:num_simulations) {
    events <- 0  # Initialize the total number of events
    time <- 0  # Initialize current time
    x <- parms2["initial_pop"]  # Start population at initial value

    # Dataframe to store results
    simulation_results <- data.frame(event = numeric(),
                                     time = numeric(),
                                     population = numeric(),
                                     event_type = character(),
                                     sim_id = numeric())
    while(time[length(time)] <= max.time) {
      # Track number of events
      events <- events + 1
      current.size <- x[length(x)]  # Use last recorded population size
      current.time <- time[length(time)]  # Use last recorded time

      # Calculate event rates with proper parameters
      b <- birthrate(x=current.size, b0=parms2["b0"], theta=parms2["theta"])
      d <- deathrate(x=current.size, d0=parms2["d0"], d1=parms2["d1"])
      i <- immigrationrate(x=current.size, iota=parms2["iota"])

      # Compute total rate and time to next event
      total.rate <- b + d + i
      if (total.rate == 0 || is.na(total.rate)) {
        break  # Stop simulation if no events can occur
        }
      increment.time <- -log(1 - runif(1)) / total.rate  # Exponential waiting time

      # Determine event type
      event_probs <- c(d, b, i) / total.rate
      event_outcome <- sample(c("death", "birth", "immigration"), size=1, prob=event_probs)
      change <- ifelse(event_outcome == "death", -1, ifelse(event_outcome == "birth", 1, 0))  # Immigra

      # Store results
      simulation_results <- rbind(simulation_results, data.frame(
        event = events,
        time = current.time + increment.time,
        population = max(0, current.size + change),  # Ensure population doesn't go negative
        event_type = event_outcome,
        sim_id = sim
      ))

      # Update state variables
      time <- c(time, current.time + increment.time)
      x <- c(x, max(0, current.size + change))  # Prevent negative population
    }

    # Store results of this simulation
    simulation2[[sim]] <- simulation_results
  }

  # Save all results
  saveRDS(simulation2, "data/simulation2.Rdata")

# 3: Run complex model
simulation3 <- vector("list", num_simulations)  # Initialize storage

  for (sim in 1:num_simulations) {
    events <- 0  # Initialize the total number of events
    time <- 0  # Initialize current time
    x <- parms3["initial_pop"]  # Start population at initial value

    # Dataframe to store results
```

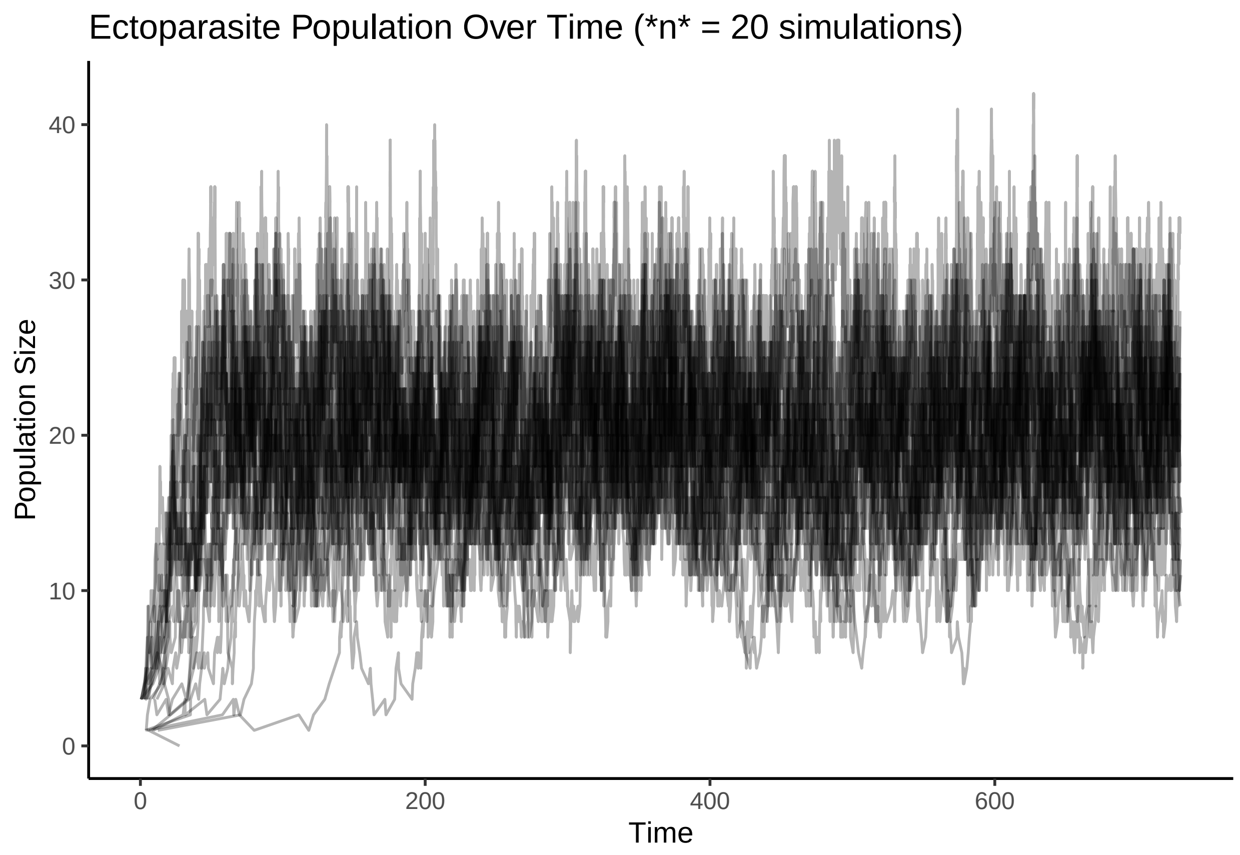
```r
    simulation_results <- data.frame(event = numeric(),
                                     time = numeric(),
                                     population = numeric(),
                                     event_type = character(),
                                     sim_id = numeric())

    while(time[length(time)] <= max.time) {
      # Track number of events
      events <- events + 1
      current.size <- x[length(x)]  # Use last recorded population size
      current.time <- time[length(time)]  # Use last recorded time

      # Calculate event rates with proper parameters
      b <- birthrate(x=current.size, b0=parms3["b0"], theta=parms3["theta"])
      d <- deathrate(x=current.size, d0=parms3["d0"], d1=parms3["d1"])
      i <- immigrationrate(x=current.size, iota=parms3["iota"])

      # Compute total rate and time to next event
      total.rate <- b + d + i
      if (total.rate == 0 || is.na(total.rate)) {
        break  # Stop simulation if no events can occur
        }
      increment.time <- -log(1 - runif(1)) / total.rate  # Exponential waiting time

      # Determine event type
      event_probs <- c(d, b, i) / total.rate
      event_outcome <- sample(c("death", "birth", "immigration"), size=1, prob=event_probs)
      change <- ifelse(event_outcome == "death", -1, ifelse(event_outcome == "birth", 1, 0))  # immigra

      # Store results
      simulation_results <- rbind(simulation_results, data.frame(
        event = events,
        time = current.time + increment.time,
        population = max(0, current.size + change),  # Ensure population doesn't go negative
        event_type = event_outcome,
        sim_id = sim
      ))

      # Update state variables
      time <- c(time, current.time + increment.time)
      x <- c(x, max(0, current.size + change))  # Prevent negative population
    }

    # Store results of this simulation
    simulation3[[sim]] <- simulation_results
  }

  # Save all results
  saveRDS(simulation3, "data/simulation3.Rdata")

####  List of all simulation models
all_simulations <- list(simulation0, simulation1, simulation2, simulation3)
names(all_simulations) <- c("Model 0", "Model 1", "Model 2", "Model 3")
```

Now, I need a way to sample from these data.

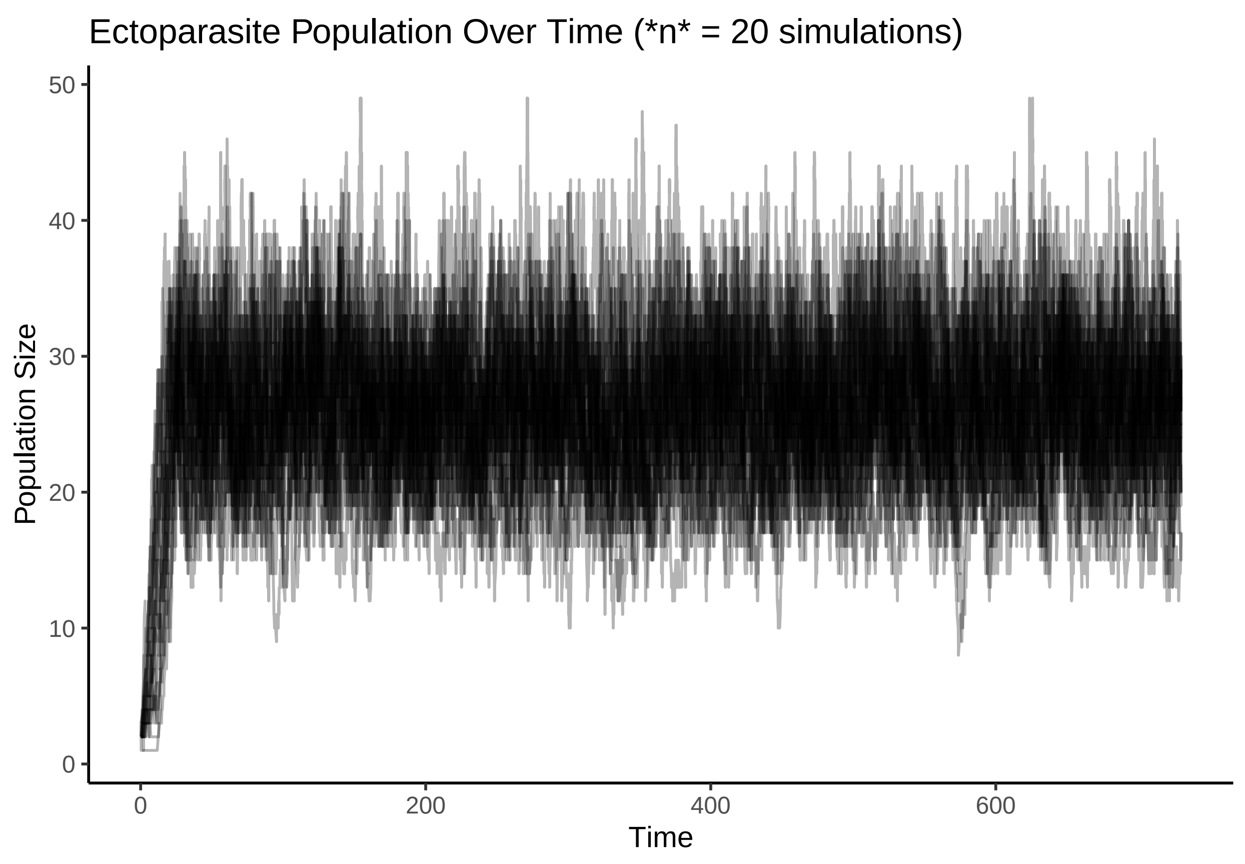
```r
# Combine simulation results into one dataframe
results0 <- do.call(rbind, simulation0)
results1 <- do.call(rbind, simulation1)
results2 <- do.call(rbind, simulation2)
results3 <- do.call(rbind, simulation3)

# plot(time,x, type = 'b', xlab='Time', ylab='Ectoparasite population size') # this is for single sim

library(ggplot2)
# make a plot with all the simulations
ggplot(results0, aes(x = time, y = population, group = sim_id)) +
  geom_line(alpha = 0.3) +  # Add transparency to avoid clutter
  labs(title = "Ectoparasite Population Over Time (*n* = 20 simulations)",
       x = "Time", y = "Population Size", color = "Simulation ID") +
  theme_classic() +
  theme(legend.position = "none")  # hide legend cause too many lines
```



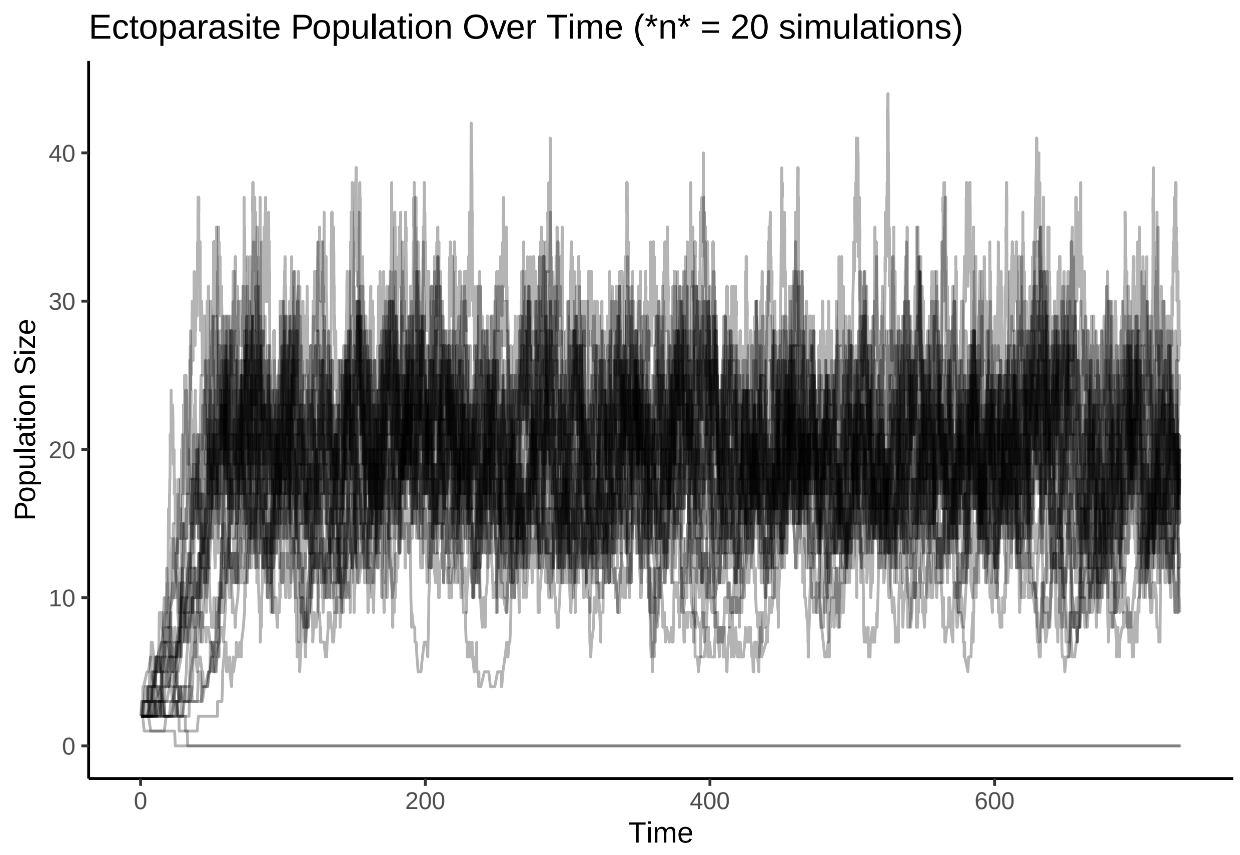
```r
ggplot(results1, aes(x = time, y = population, group = sim_id)) +
  geom_line(alpha = 0.3) +  # Add transparency to avoid clutter
  labs(title = "Ectoparasite Population Over Time (*n* = 20 simulations)",
       x = "Time", y = "Population Size", color = "Simulation ID") +
  theme_classic() +
  theme(legend.position = "none")  # hide legend cause too many lines
```



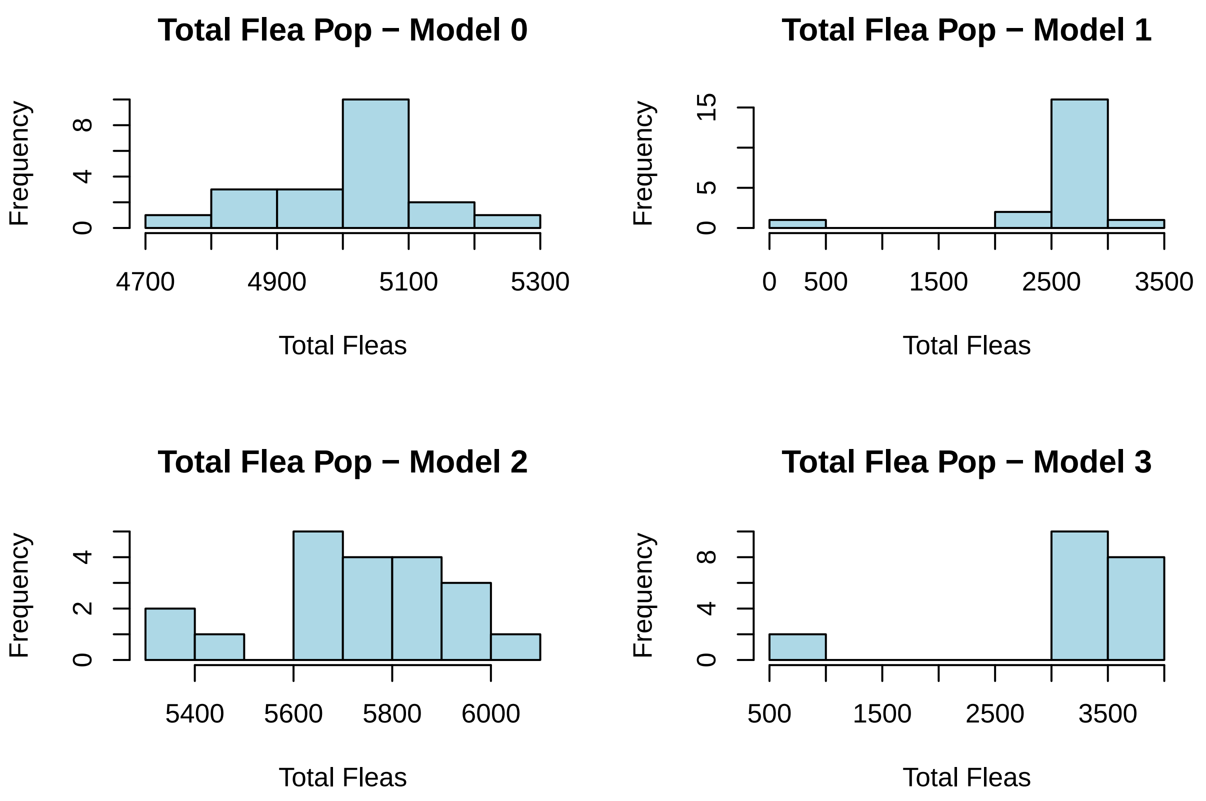
```r
ggplot(results2, aes(x = time, y = population, group = sim_id)) +
  geom_line(alpha = 0.3) +  # Add transparency to avoid clutter
  labs(title = "Ectoparasite Population Over Time (*n* = 20 simulations)",
       x = "Time", y = "Population Size", color = "Simulation ID") +
  theme_classic() +
  theme(legend.position = "none")  # hide legend cause too many lines
```



```r
ggplot(results3, aes(x = time, y = population, group = sim_id)) +
  geom_line(alpha = 0.3) +  # Add transparency to avoid clutter
  labs(title = "Ectoparasite Population Over Time (*n* = 20 simulations)",
       x = "Time", y = "Population Size", color = "Simulation ID") +
  theme_classic() +
  theme(legend.position = "none")  # hide legend cause too many lines
```

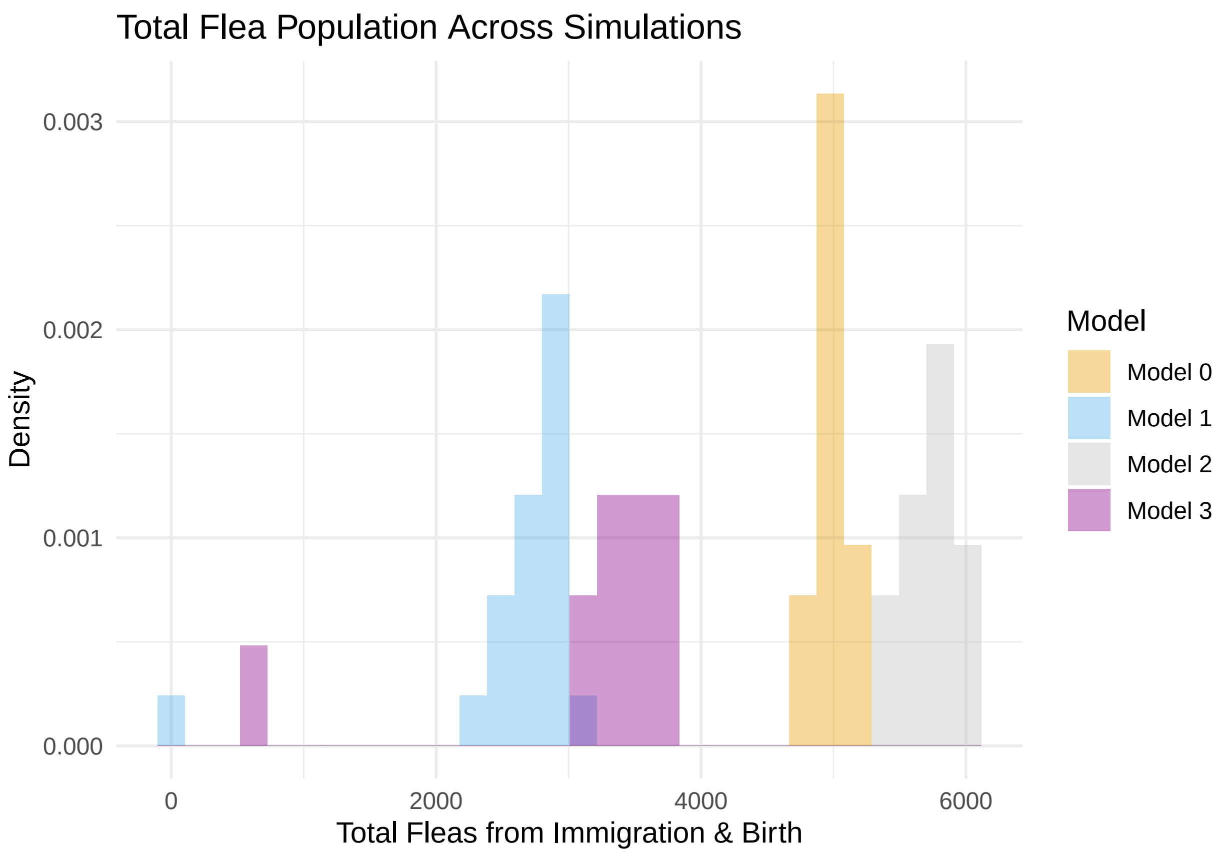


```r
# FLEA POPULATION TRAITS - calculate relevant traits

## values for one simulation
### Total Flea Population: the count of fleas during the entire simulation run

#### Function to calculate total flea population from immigration & birth events
calculate_flea_population <- function(simulation_list) {
  sapply(simulation_list, function(sim_df) {
    sum(sim_df$event_type == "immigration") + sum(sim_df$event_type == "birth")
  })
}

#### Apply function to all models
flea_total_pop_lists <- lapply(all_simulations, calculate_flea_population)

#### Plot histograms for each model as facet
par(mfrow = c(2, 2))  # Arrange plots in a 2x2 grid
for (i in seq_along(flea_total_pop_lists)) {
  hist(flea_total_pop_lists[[i]],
       main = paste("Total Flea Pop -", names(flea_total_pop_lists)[i]),
       xlab = "Total Fleas",
       col = "lightblue",
       border = "black")
}
```



```r
par(mfrow = c(1,1))  # Reset plot layout

#### Convert list to long-format dataframe for ggplot
flea_data <- bind_rows(
  lapply(names(flea_total_pop_lists), function(model) {
    data.frame(Total_Fleas = flea_total_pop_lists[[model]], Model = model)
  })
)

#### Plot histogram using ggplot2 for each model as overlapping on same grid
ggplot(flea_data, aes(x = Total_Fleas, fill = Model)) +
  geom_histogram(aes(y = ..density..), position = "identity", alpha = 0.4, bins = 30) +
  scale_fill_manual(values = c("#E69F00", "#56B4E9","grey", "darkmagenta")) +  # Custom colors
  labs(title = "Total Flea Population Across Simulations",
       x = "Total Fleas from Immigration & Birth",
       y = "Density") +
  theme_minimal()
```

```
## Warning: The dot-dot notation (`..density..`) was deprecated in ggplot2 3.4.0.
## i Please use `after_stat(density)` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

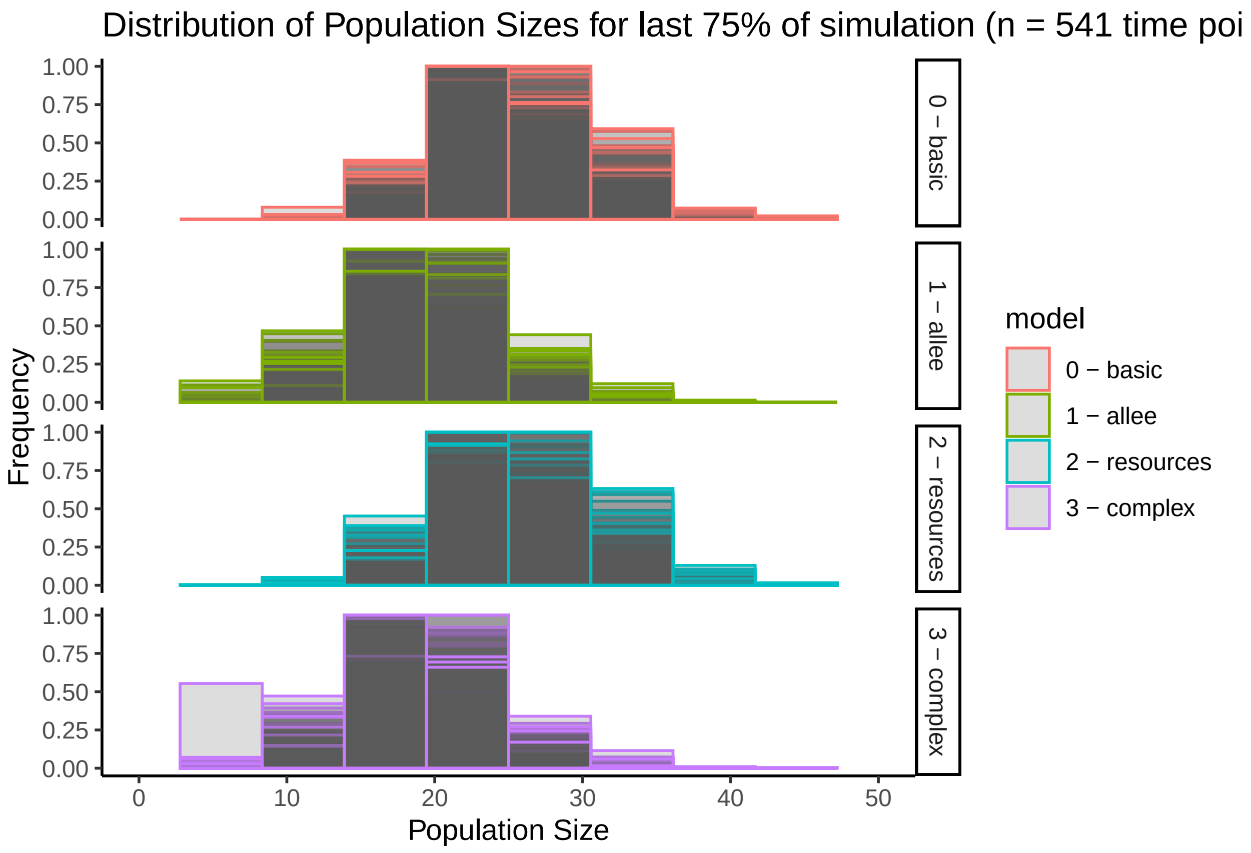


```
### Snapshot/Sampling Event Flea Population: the count of fleas at one snapshot (e.g., end of day)

#### create function round time to the nearest integer, keep only the last observation per time unit
end_of_day_flea_pop <-function(sim){
  sim %>%  # Apply operations to the dataset
  mutate(time_unit = floor(time)) %>%  # Convert time to integer units
  group_by(time_unit) %>%
  slice_tail(n = 1) %>%  # Get the last observation per unit
  ungroup() %>%
  select(time_unit, population)  # Keep relevant columns
}

# apply the end of day calculation across simulations
sim0_end_of_day_pops <- lapply(simulation0, end_of_day_flea_pop)
sim1_end_of_day_pops <- lapply(simulation1, end_of_day_flea_pop)
sim2_end_of_day_pops <- lapply(simulation2, end_of_day_flea_pop)
sim3_end_of_day_pops <- lapply(simulation3, end_of_day_flea_pop)

# create a function to sample only a portion of the days (e.g. last 25%)
subset_simulation <- function(sim_df, range_percent) {
  # Ensure range_percent is a valid numeric vector of length 2 (e.g., c(0, 10) or c(75, 100))
  if (length(range_percent) != 2 || any(range_percent < 0) || any(range_percent > 100)) {
    stop("range_percent must be a numeric vector of length 2 between 0 and 100.")
  }

  # Get the start and end time from the simulation
  min_time <- min(sim_df$time_unit)
```

```r
  max_time <- max(sim_df$time_unit)

  # Compute the time range corresponding to the percentage range
  time_start <- min_time + (range_percent[1] / 100) * (max_time - min_time)
  time_end <- min_time + (range_percent[2] / 100) * (max_time - min_time)

  # Subset the dataframe based on the calculated time range
  sim_df %>%
    filter(time_unit >= time_start & time_unit <= time_end)
}

# limit population snapshots to latter 75% (eliminating burn-in period)
sim0_eod_pop_last75 <- lapply(sim0_end_of_day_pops, subset_simulation, range_percent = c(26, 100))
sim0_eod_pop_last75_combined <- bind_rows(
  lapply(seq_along(sim0_eod_pop_last75), function(i) {
    sim0_eod_pop_last75[[i]] %>% mutate(sim_id = as.factor(i))  # Assign simulation ID
  })
)
sim0_eod_pop_last75_combined$model <- "0 - basic"
sim0_eod_pop_last75_combined$sim_id <- paste("0-", sim0_eod_pop_last75_combined$sim_id, sep = "")
sim1_eod_pop_last75 <- lapply(sim1_end_of_day_pops, subset_simulation, range_percent = c(26, 100))
sim1_eod_pop_last75_combined <- bind_rows(
  lapply(seq_along(sim1_eod_pop_last75), function(i) {
    sim1_eod_pop_last75[[i]] %>% mutate(sim_id = as.factor(i))  # Assign simulation ID
  })
)
sim1_eod_pop_last75_combined$model <- "1 - allee"
sim1_eod_pop_last75_combined$sim_id <- paste("1-", sim1_eod_pop_last75_combined$sim_id, sep = "")

sim2_eod_pop_last75 <- lapply(sim2_end_of_day_pops, subset_simulation, range_percent = c(26, 100))
sim2_eod_pop_last75_combined <- bind_rows(
  lapply(seq_along(sim2_eod_pop_last75), function(i) {
    sim2_eod_pop_last75[[i]] %>% mutate(sim_id = as.factor(i))  # Assign simulation ID
  })
)
sim2_eod_pop_last75_combined$model = "2 - resources"
sim2_eod_pop_last75_combined$sim_id <- paste("2-", sim2_eod_pop_last75_combined$sim_id, sep = "")

sim3_eod_pop_last75 <- lapply(sim3_end_of_day_pops, subset_simulation, range_percent = c(26, 100))
sim3_eod_pop_last75_combined <- bind_rows(
  lapply(seq_along(sim3_eod_pop_last75), function(i) {
    sim3_eod_pop_last75[[i]] %>% mutate(sim_id = as.factor(i))  # Assign simulation ID
  })
)
sim3_eod_pop_last75_combined$model = "3 - complex"
sim3_eod_pop_last75_combined$sim_id <- paste("3-", sim3_eod_pop_last75_combined$sim_id, sep = "")

eod_pop_last75 <- rbind(sim0_eod_pop_last75_combined,
                        sim1_eod_pop_last75_combined,
                        sim2_eod_pop_last75_combined,
                        sim3_eod_pop_last75_combined)
# plot the histogram
# hist(sim0_eod_pop_last75[[1]]$population)
```

```
ggplot(eod_pop_last75, aes(x = population, group = sim_id, color = model)) +
 # geom_freqpoly(binwidth = 10, size = 1, alpha = 0.5) +  # stepbins like a histogram
  geom_histogram(alpha = 0.2, aes(y = ..ncount..), position = 'identity', bins = 10) +
 # geom_bar(pos="dodge") +
  scale_x_continuous(limits = c(0, 50)) +
 # scale_y_continuous(limits = c(0, 0.15)) +
  labs(title = "Distribution of Population Sizes for last 75% of simulation (n = 541 time points) ",
       x = "Population Size",
       y = "Frequency") +
  theme_classic() +
 facet_grid(model ~ .)
```

```
## Warning: Removed 160 rows containing missing values or values outside the scale range
## (`geom_bar()`).
```


Distribution of Population Sizes for last 75% of simulation (n = 541 time poi
Frequency
0 – basic
1 – allee
2 – resources
3 – complex
Population Size
model
0 – basic
1 – allee
2 – resources
3 – complex