## **Introduction**

Parallel computing enables a method or a system to split into sections or parts and distribute the workload into smaller tasks that can run simultaneously on different processors. There are times when the main program generates jobs, and they may need to wait their start until the parent task is completed because they need the results of that task. This will cause a queue of individual tasks to accumulate even though there are CPU’s available to perform the tasks. Understanding this computational process and the distribution of the length of time it takes to complete along with all its subtasks may help developers construct codes that efficiently run in parallel and that manages jobs on compute clusters. [1]

The authors, Nolan and Lang, detail the Aldous Krebs stochastic branching process. In this process, the initial job generates other jobs with the times between the start of jobs independent and identically distributed. These offspring jobs must all wait until their parent program completes before they can start running. However, each job can spawn new jobs of its own as soon as it is generated while it is waiting to complete or begin. They studied the time it takes for the entire system of jobs to finish running, which provided information on the combination of job creation and completion rates that led to processes completing in a fixed time-period and those that may never complete. [1]

Monte Carlo simulations are designed to model the probability of different outcomes in a process that cannot easily be predicted due to the intervention of random variables. It is a technique used to understand the impact of risk and uncertainty in prediction and forecasting models. The Monte Carlo Method repeatedly generates independent random outcomes from its probability distribution. The properties of these empirical results as approximations to their expected properties are used to provide insight to the behavior of the branching process. [1]

In this study, we simulate the Aldous and Krebs stochastic branching process and determine the condition under which the system terminates in a fixed time using the Monte Carlo method.

## **Background**

The branching process by Aldous and Krebs [3] describes the birth-and-assassination process, which is a system of evolving individuals. During its lifetime, an individual produces offspring at fixed rates *µ > 0*. In addition, each individual *u* is equipped with a random *Ku* units of times, where the positive random variables (*Ku*) are all independent and identically distributed. An individual begins when its parent dies, and dies when its complete. This branching process results in a finite collection of families and under certain conditions of *Ku*, where the critical value of *µ = µc*, that allows the process to survive indefinitely with positive probability (the queueing process). Later, Bordenave studied the birth-and-assassination process when *Ku* is an exponential random variable. He proved that the evolution stops when *µ = µc* with probability 1 and If 0 < λ < 1/4, then the process is almost surely finite [4].

The simulation study begins with a job that lasts for a random amount of time. The completion time is determined by the exponential distribution that has a rate parameter, κ, where the density is . The birth time is determined the Poisson process; which is the probability mechanism for generating random events in time, like the starting time for jobs. The count of the number of events generated in a fixed time interval is a Poisson random variable, where

The Poisson process makes it possible to determine births/start times of jobs by the inter-arrival times from the exponential distribution, that is the time between each pair of consecutive events has an exponential distribution and that the inter-arrival times are independent of one another. In the Poisson process, the initial job starts at time 0 and its 1st offspring’s birth date is the first inter-arrival time by generating as an exponential random outcome. The time between the birth of the 1st and 2nd offspring with another independent exponential distribution, and the time between the 2nd and 3rd offspring also have an independent exponential distribution of the others, and so forth.

The Poisson distribution is the number of births Poisson over a fixed time interval, t, where 𝛌 is the rate per unit of time. The rate of births is represented by the parameter kappa, **κ**, while the rate of deaths/completions is represented by the lambda, 𝛌, variable. The *familyTree()* function was developed to simulate this branching process. It encapsulates the *genKidsV()* function, which generates the offspring’s for all of the jobs, and the rates of the inter-arrival time of the offspring, **κ**, and its run time rate, 𝛌. The *familyTree()* function takes the parent’s birth and completion time to generate future generations, limits the maximum number of generations, and limits maximum number of offspring’s that generates a random birth and assassination process.

The *exptOne()* function was designed to call the *familyTree()* function and return summary statistics by taking in various combinations of the 𝛌 and **κ** parameter values. A replication function MCBA calls the *exptOne()* function in order to perform multiple replications of 𝛌 and **κ** parameter values that takes the *param* variable as a matrix of 𝛌 and **κ** value pairs and repeats multiple runs. These functions are used to simulate the branching process in R.

In this report, we carry out a simulation study to see if the re-parameterization suggested in Section 7.9.1 is appropriate. We investigate different fixed values of **κ** and run the simulation for various values of 𝛌. We compare the results to other simulations where **κ** is c ≠ 1, but the ratio of 𝛌/**κ** that matches one of the 𝛌 values from the earlier simulation when **κ** value was 1.

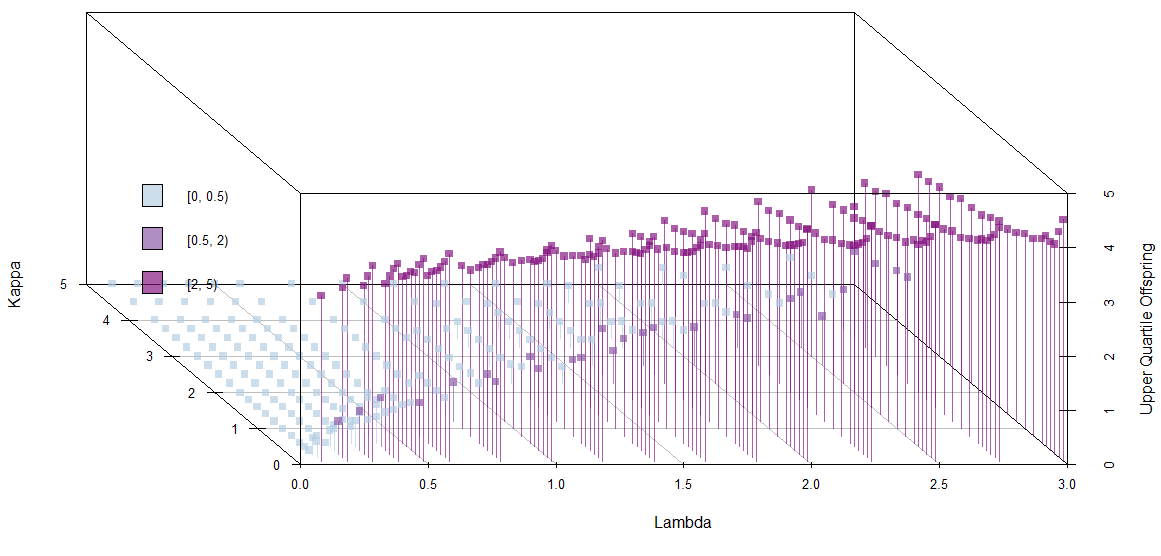
**Table 1: The** 𝛌 and **κ parameter values set in Section 7.9.1**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Lambdas | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.8 | 1.0 | 1.2 | 1.4 | 1.6 | 1.8 | 2.0 | 2.25 | 2.5 | 3.0 |
| Kappas | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.8 | 1.0 | 1.2 | 1.4 | 1.6 | 1.8 | 2.0 | 2.25 | 2.5 | 3.0 |
| 3.25 | 3.75 | 4.0 | 4.5 | 5.0 |

The authors, Nolan and Lang, run the branching process with the parameters from Table 1. They set the max number of offspring to 1000, the max number of generations to 20, and generated 400 repetitions of the process. This simulation resulted in an outcome consisting of the number of generations and the number of offspring for every lambda \* kappa pair. In this case, 17 values of lambda and 23 values of kappa equated to 391 combination of outcomes of generations and offspring.

Figure 1 shows a 3-dimensional scatter plot having the upper quartile of number of offspring in the z-axis (log base 10 of number of offspring). The process either dies out soon, or grows exponentially and generates many offspring. It appears that process ends soon when kappa is greater than lambda, and the process grows when lambda is greater than kappa.

Figure 2 on the left, shows the proportion of the 400 simulations for each (𝛌, **κ**) pair that reach 20 generations or 1000 offspring and therefore were terminated. The right image in Figure 2, shows the proportions of the 400 simulations that have at least 20 offspring.

**Figure 1. Number of Offspring (log10) by λ and κ.**

**Figure 2. Image Map of the Proportion of Replicates that Reach the Simulation Limits (Left). The Image Map of the Proportion of Simulations with at least 20 Offspring (Right).**

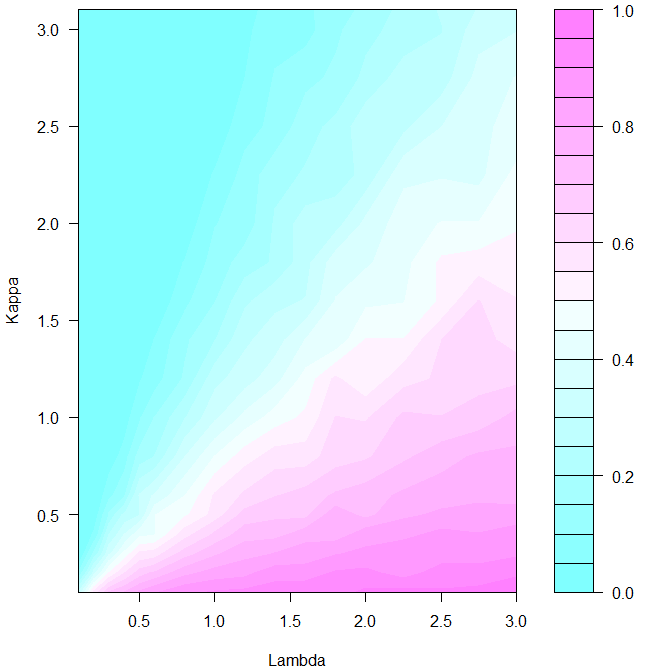
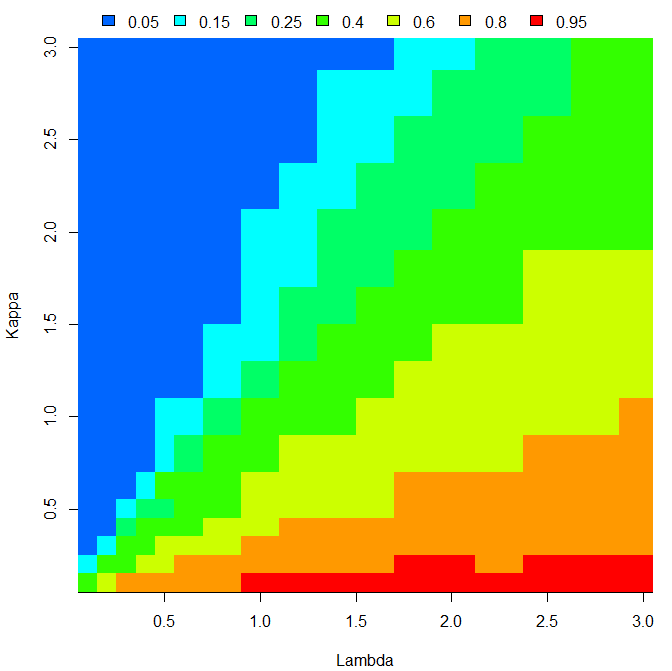
 

Figure 2 shows that nearly all of the simulated processes where kappa is small are still alive at the time the simulation stopped and nearly all of the simulated processes for small lambda values terminate before the limitations are reached. Also, it shows those pairs below the pale diagonal region are more likely than not to still be running and those above the pale diagonal region are less likely than not to be completed. Both images in Figure 2 have similar appearances. These visualizations show that the relationship between 𝛌 and **κ** are key to the longevity of the process. When **κ** > 𝛌, the process dies out with a few generations. There appears to be a region where the process at times produces many thousands of children and seems to carry on indefinitely, and other regions where lambdas are small when the process never really terminates. This suggests that the process may be reparametrized in terms of 𝛌/**κ** ratios that work with only one parameter like when **κ** = 1.

## **Method**

The methods used in this simulation branching process is from the book, Case Studies in Data Science in R, Chapter 7. We used the methods to answer Question 10 which states “Carry out a simulation study to see if the re-parameterization suggested in Section 7.9.1 is appropriate. For example, fix κ to be 1, and run the simulation for various values of λ. Compare the results to other simulations where κ is ϲ ≠ 1, but the ratio of λ / ϲ matches one of the λ values from the earlier simulation when κ was 1”.

The results from running the codes from the book showed that processes are alive (and growing exponentially) when λ is larger than κ, this is when the birth rate exceeds the completion rate. The authors suggest that there is a relationship between the input parameters λ and κ that may further optimize the outcomes in the branching process.

We simulate several values of the relation λ / κ that are greater than and less than 1, for three different sets of values of κ. Table 2 details the proposed configuration parameter values for the first simulation; showing the 11 values of λ for each value of κ, such that the ratio λ / κ is maintained across these three groups. The first simulation is shortened simulation with 50 repetitions and 1089 combinations (33\*33). Table 3 shows the proposed parameter values for the second simulation. The second simulation was a longer simulation with 400 repeats and took several days to complete with 6084 combinations of outcomes (78\*78). Both simulations ran the branching process using the same parameter values for max number of offspring (1000) and max number of generations (20). We expect that many repetitions of the branching process simulation may confirm similar pattern outcome for identical values of the ratio λ / κ.

**Table 2. Simulation 1 Parameter values for λ and κ.**

| λ | κ | λ / κ | λ | κ | λ / κ | λ | κ | λ / κ |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0.10 | 1.00 | 0.10 | 0.15 | 1.50 | 0.10 | 0.20 | 2.00 | 0.10 |
| 0.20 | 1.00 | 0.20 | 0.30 | 1.50 | 0.20 | 0.40 | 2.00 | 0.20 |
| 0.33 | 1.00 | 0.33 | 0.50 | 1.50 | 0.33 | 0.67 | 2.00 | 0.33 |
| 0.50 | 1.00 | 0.50 | 0.75 | 1.50 | 0.50 | 1.00 | 2.00 | 0.50 |
| 0.67 | 1.00 | 0.67 | 1.00 | 1.50 | 0.67 | 1.33 | 2.00 | 0.67 |
| 1.00 | 1.00 | 1.00 | 1.50 | 1.50 | 1.00 | 2.00 | 2.00 | 1.00 |
| 1.50 | 1.00 | 1.50 | 2.25 | 1.50 | 1.50 | 3.00 | 2.00 | 1.50 |
| 2.00 | 1.00 | 2.00 | 3.00 | 1.50 | 2.00 | 4.00 | 2.00 | 2.00 |
| 3.00 | 1.00 | 3.00 | 4.50 | 1.50 | 3.00 | 6.00 | 2.00 | 3.00 |
| 5.00 | 1.00 | 5.00 | 7.50 | 1.50 | 5.00 | 10.00 | 2.00 | 5.00 |
| 10.00 | 1.00 | 10.00 | 15.00 | 1.50 | 10.00 | 20.00 | 2.00 | 10.00 |

**Table 3. Simulation 2 Parameter values for λ and κ.**

| λ | κ | λ / κ | λ | κ | λ / κ | λ | κ | λ / κ |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0.1 | 1 | 0.1 | 0.2 | 2 | 0.1 | 0.3 | 3 | 0.1 |
| 0.2 | 1 | 0.2 | 0.4 | 2 | 0.2 | 0.6 | 3 | 0.2 |
| 0.3 | 1 | 0.3 | 0.6 | 2 | 0.3 | 0.9 | 3 | 0.3 |
| 0.4 | 1 | 0.4 | 0.8 | 2 | 0.4 | 1.2 | 3 | 0.4 |
| 0.5 | 1 | 0.5 | 1 | 2 | 0.5 | 1.5 | 3 | 0.5 |
| 0.6 | 1 | 0.6 | 1.2 | 2 | 0.6 | 1.8 | 3 | 0.6 |
| 0.8 | 1 | 0.8 | 1.6 | 2 | 0.8 | 2.4 | 3 | 0.8 |
| 1 | 1 | 1 | 2 | 2 | 1 | 3 | 3 | 1 |
| 1.2 | 1 | 1.2 | 2.4 | 2 | 1.2 | 3.6 | 3 | 1.2 |
| 1.4 | 1 | 1.4 | 2.8 | 2 | 1.4 | 4.2 | 3 | 1.4 |
| 1.6 | 1 | 1.6 | 3.2 | 2 | 1.6 | 4.8 | 3 | 1.6 |
| 1.8 | 1 | 1.8 | 3.6 | 2 | 1.8 | 5.4 | 3 | 1.8 |
| 2 | 1 | 2 | 4 | 2 | 2 | 6 | 3 | 2 |
| 2.25 | 1 | 2.25 | 4.5 | 2 | 2.25 | 6.75 | 3 | 2.25 |
| 2.5 | 1 | 2.5 | 5 | 2 | 2.5 | 7.5 | 3 | 2.5 |
| 2.75 | 1 | 2.75 | 5.5 | 2 | 2.75 | 8.25 | 3 | 2.75 |
| 3 | 1 | 3 | 6 | 2 | 3 | 9 | 3 | 3 |
| 3.5 | 1 | 3.5 | 7 | 2 | 3.5 | 10.5 | 3 | 3.5 |
| 4 | 1 | 4 | 8 | 2 | 4 | 12 | 3 | 4 |
| 4.5 | 1 | 4.5 | 9 | 2 | 4.5 | 13.5 | 3 | 4.5 |
| 5 | 1 | 5 | 10 | 2 | 5 | 15 | 3 | 5 |
| 6 | 1 | 6 | 12 | 2 | 6 | 18 | 3 | 6 |
| 7 | 1 | 7 | 14 | 2 | 7 | 21 | 3 | 7 |
| 8 | 1 | 8 | 16 | 2 | 8 | 24 | 3 | 8 |
| 9 | 1 | 9 | 18 | 2 | 9 | 27 | 3 | 9 |
| 10 | 1 | 10 | 20 | 2 | 10 | 30 | 3 | 10 |

## **Results**

Simulation 1 was ran with the input parameters from Table 2. Figure 3 shows the results of simulation 1. It shows the three sets of κ = 1, 1.5, and 2 plotted along the y axis for κ, λ along the x axis, and the number of offspring along the z axis.

From left to right, for small values of λ (less than 1), it shows mostly gray dots meaning that the number of offspring were less than 3.16 (). The few light purple points where λ is equal to κ show that the number of offspring were greater than 3 but less than 100 (). Most of the points were dark purple where λ is greater than κ that showed a larger number of offspring, between 100 and 10,000 (). This simulation was limited by the models initial parameters.

Visually, it appears that the distribution of dots is similar for κ = 1, 1.5, and 2; meaning that, disregarding the values for λ and κ, the branching process simulation is determined by the relationship λ / κ.

**Figure 3. Simulation 1 Outcome of the Number of Offspring (log10) by λ and κ for the proposed re-parameterization.**

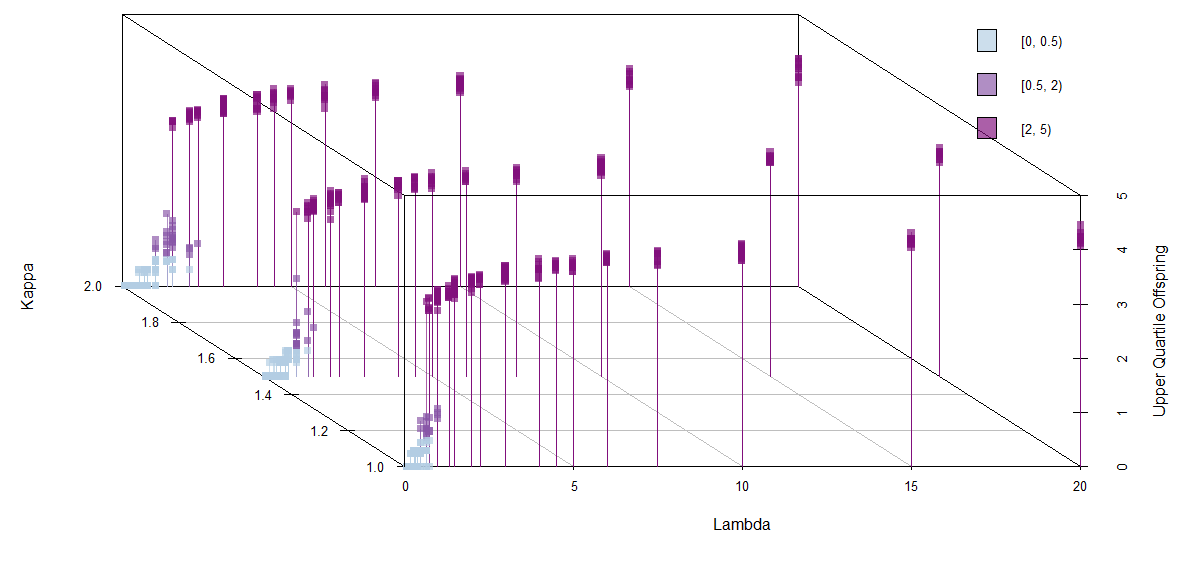
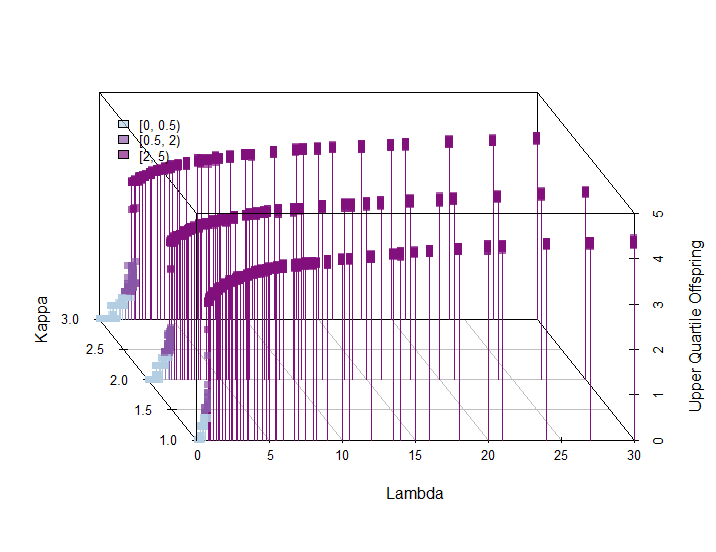


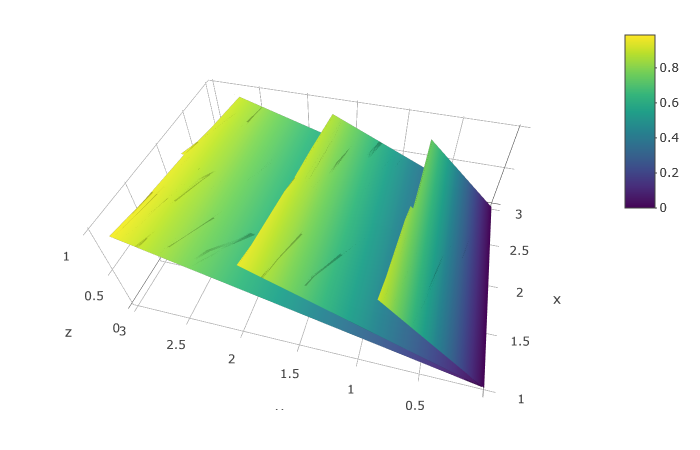
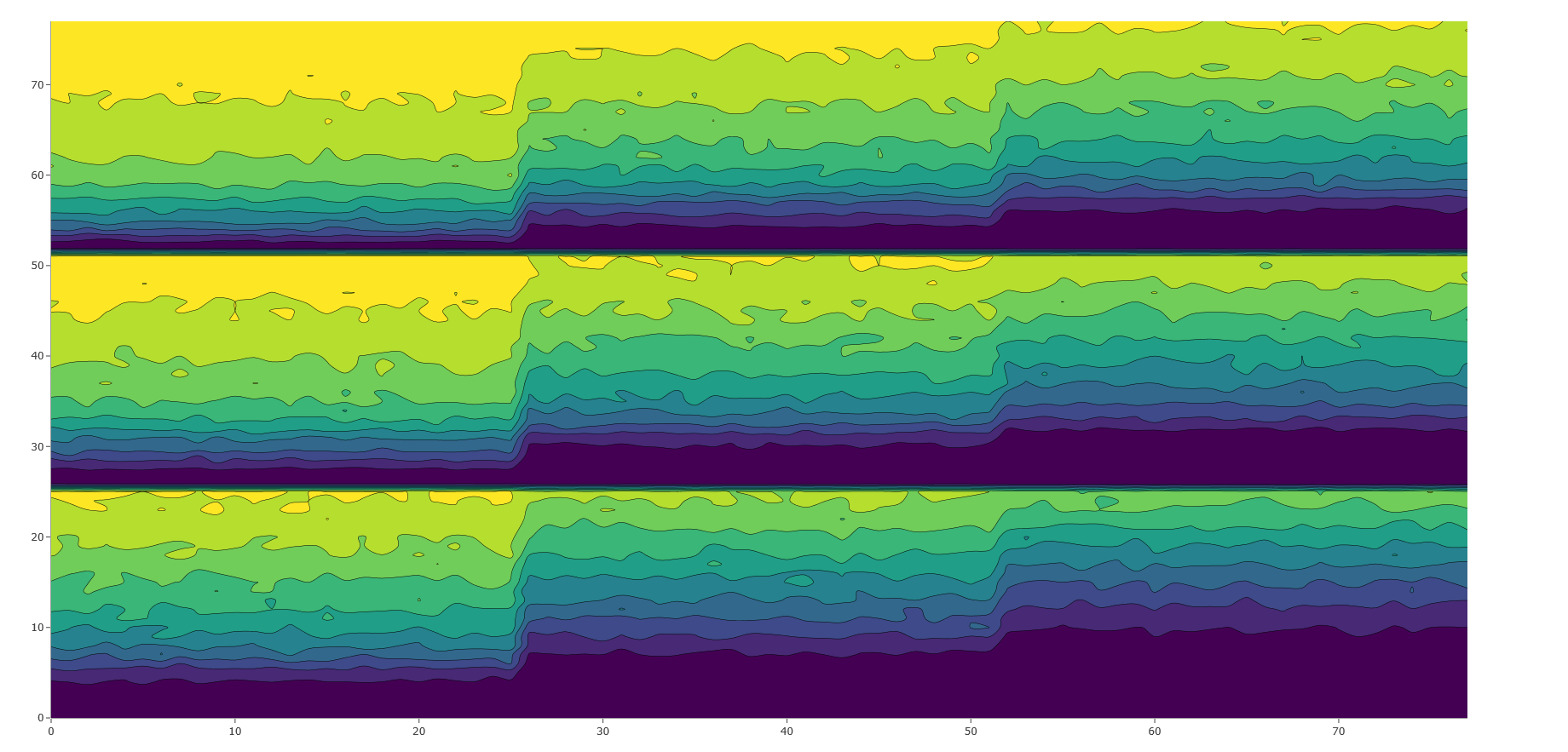
Figure 4 displays the results from the 2nd simulation in a 3D scatterplot. The second simulation evaluated kappa values at 1, 2, and 3 and lambda values from Table 3, which resulted in 6084 combinations. A similar pattern is observed between the 1st and 2nd simulations (see Figure 3 and Figure 4). Both Figures 3 and 4 show that the process either dies out or generates many offspring. Therefore, the reduced number of repeats in the 1st simulation did not have a great impact nor the reduced number of lambdas evaluated.

**Figure 4. Simulation 2 Outcome of the Number of Offspring (log10) by λ and κ for the proposed re-parameterization.**



For the 2nd simulation, Figure 5 on the left, shows the proportion of the 6084 simulations for each (𝛌, **κ**) pair that reach 20 generations or 1000 offspring and therefore were terminated. The right image in Figure 5, shows the proportions of the 6084 simulations that have at least 20 offspring. The x axis in both images represent lambda values, the y axis in both images represent the kappa values, and z is the proportion of simulations that either reached simulation parameters (left image) or have at least 20 offspring (right image).

Figure 5 shows that nearly all of the simulated processes where kappa < 1 terminate before the limitations are reached and many of the kappas > 2 are still alive at the time the simulation stopped for probability of 1. Also, it shows those pairs in the green gradient regions are more likely than not to still be running and those in the blue gradient regions are less likely than not to be completed. Both images in Figure 5 have similar appearances; the left is a surface map while the right is a contour map. These visualizations confirm that the ratio between 𝛌 and **κ** are key to the longevity of the process. When **κ** < 1, the process completes more jobs with less jobs still alive and even fewer jobs reaching simulation limits. When **κ** > 2, it shows still about 20 – 25% of the jobs reaching simulation limits. The dark blue regions show the proportion of the jobs that are completed. While the yellow regions show the proportions of jobs that reached simulation limits and got hung up.

**Figure 5. Simulation 2 Image Map of the Proportion of Replicates that Reach the Simulation Limits (Left). The Image Map of the Proportion of Simulations with at least 20 Offspring (Right).**  

κ

𝛌

𝛌

κ

κ

Although the simulation in Section 7.9.1 by the authors works well, the re-parameterization they suggested for **κ** =1 ratio of 𝛌/**κ** pairs performed better. The 𝛌/**κ** pairs with low kappa values completed more jobs and had less jobs still alive. It also appears that when **κ** ≤ ¼, the process completes; this is observed in the dark regions of the images in Figure 5. Therefore, the simulation works best for kappa values less than 0.5 with a probability of 1 for all lambda values. Meaning the ratio between birth and completion rates of jobs should be equal with a probability of 1 and that it runs most efficiently when **κ** ≤ ¼.

The scatterplots of the simulation 1 outcome (in terms of number of generations and offspring) by λ, κ pairs support the finding from simulation 2. The scatterplots in Figure 7 show that when κ = λ = 1, there is a balance in the distributions between completion and birth rates which was also seen in the images in Figure 5.

Figure 6 shows scatterplots of the number of generations versus the number of offspring when κ = 1, and four values of λ less than 1. In most of the cases, the branching process died before reaching 5 generations and 10 offspring.

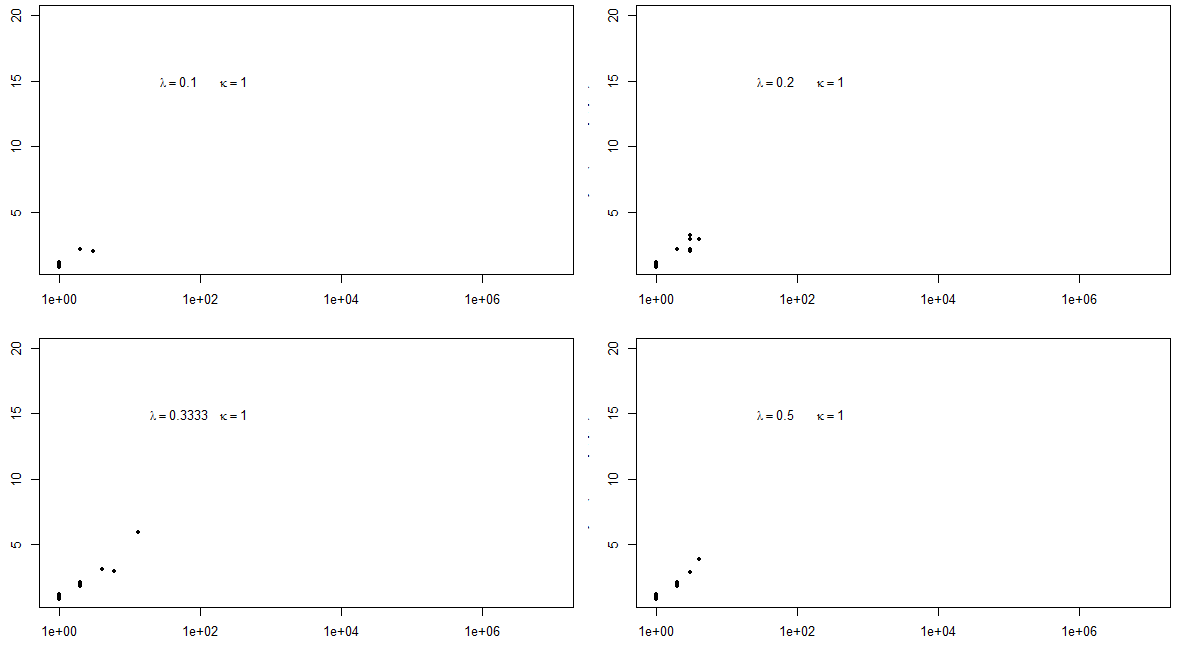
**Figure 6. Scatterplots of the number of generations versus the number of offspring, κ = 1, λ < 1.** 

Figure 7 shows scatterplots when κ = 1, and λ greater or equal than 1. When κ = λ = 1, there is a balance in the distributions: about half of the simulation processes died promptly, and the other half grew to more than 10 generations and thousands of offspring. When λ > κ, the bigger the difference, the faster growth in generations and offspring.

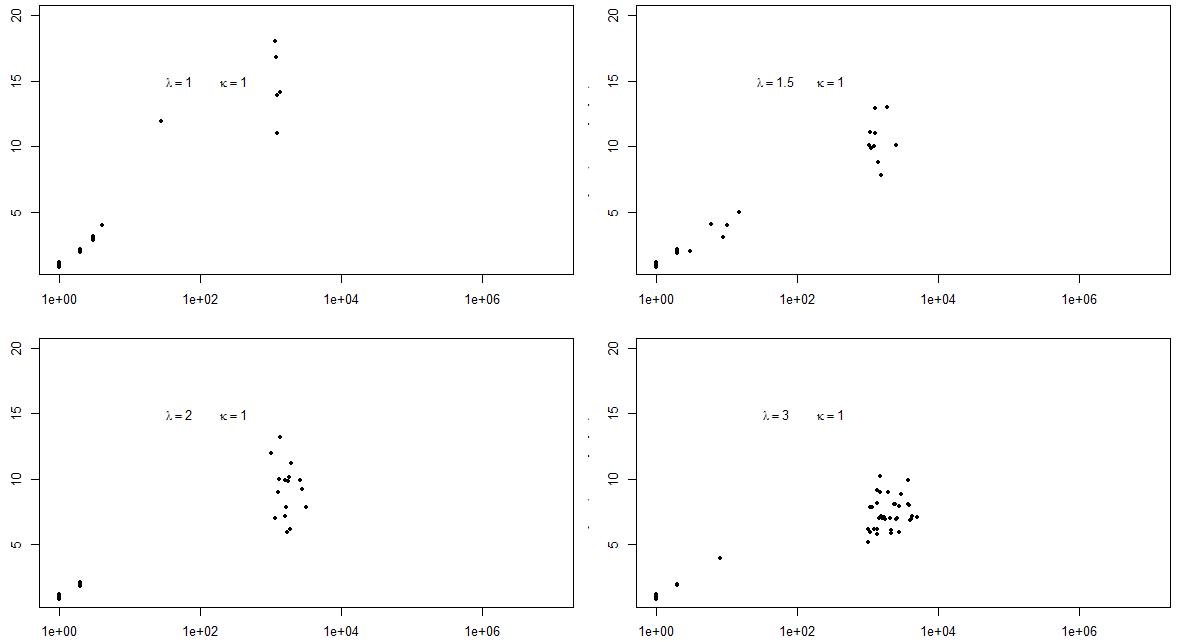
**Figure 7. Scatterplots of the number of generations versus the number of offspring, κ = 1, λ >= 1.** 

Figure 8 shows scatterplots when κ = 2, and λ is less than 2. In most of the cases, the branching process died soon, just as the scatterplot when the complete rate κ = 1.

**Figure 8. Scatterplots of the number of generations versus the number of offspring, κ = 2, λ < 2.**

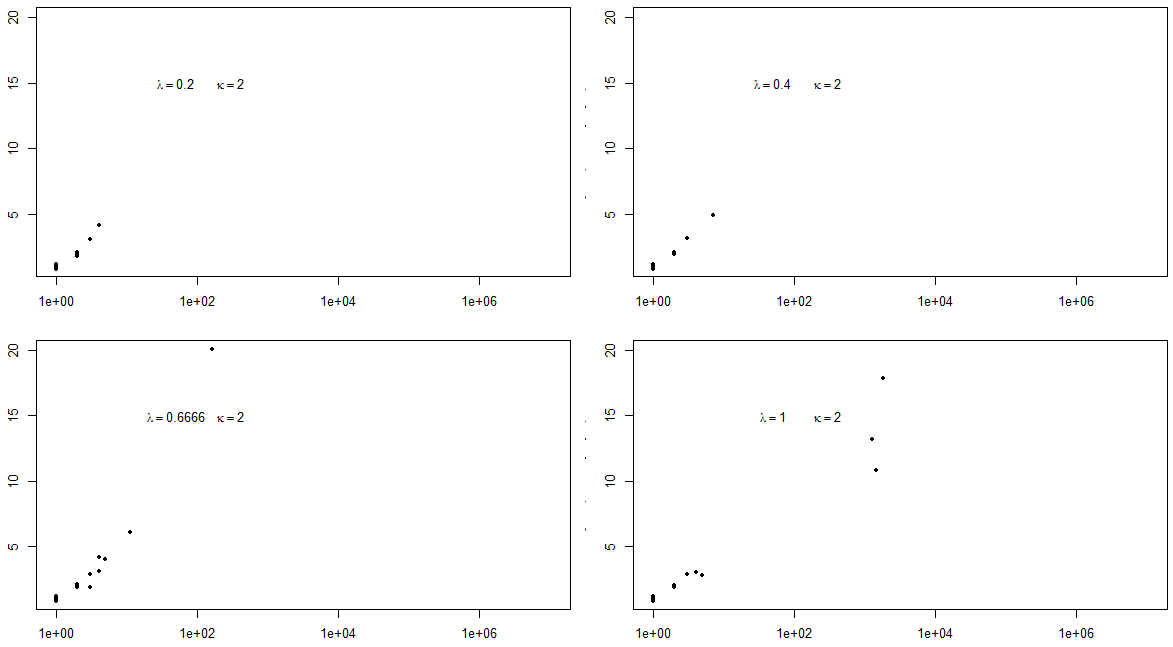
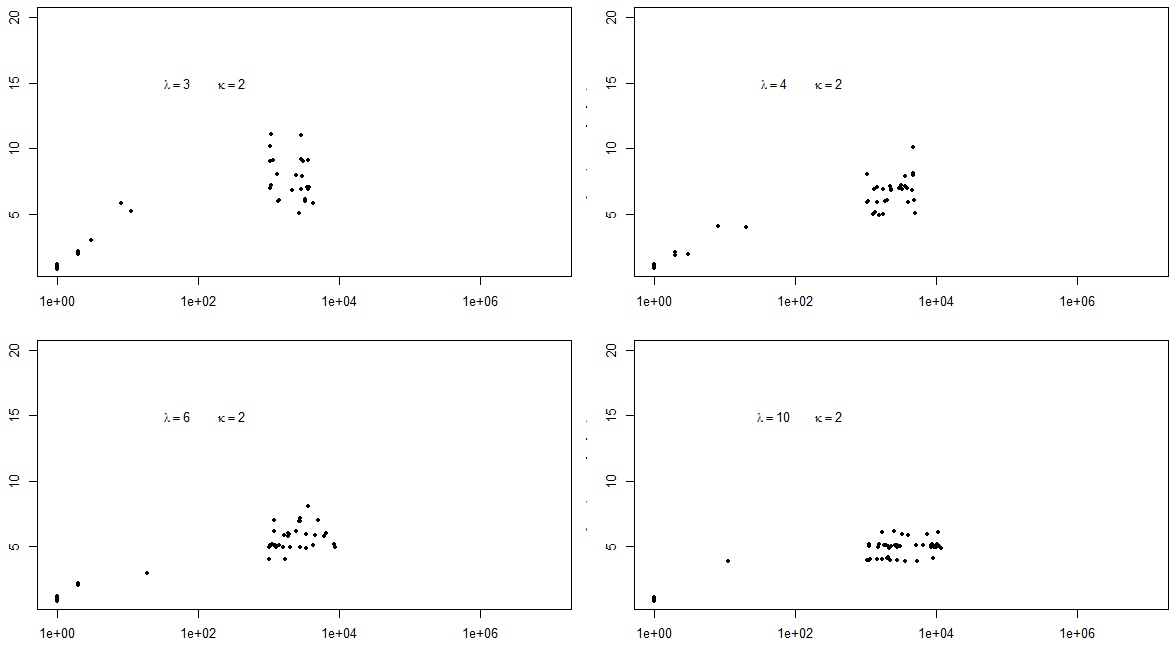


Figure 9 shows scatterplots when κ = 2, and λ greater than 2. Just as the previous scatterplots, when λ > κ, there are more chances that the simulation process continues running and growing, than the simulation ends up dying soon.

**Figure 9. Scatterplots of the number of generations versus the number of offspring, κ = 2, λ >2.**



## **Conclusions**

Although the simulation in Section 7.9.1 worked well, the suggested re-parameterization when **κ** = 1 of its 𝛌/**κ** ratios performed better. The 𝛌/**κ** ratios of **κ** = 1 showed less jobs never completing and nearly all the jobs completed when **κ** ≤ ½. Furthermore, it appears that all jobs were completed when **κ** ≤ ¼. The results confirmed that the relationship between 𝛌 and **κ** is the key to the longevity process and that it worked most efficiently with ratios of **κ** = 1;meaning the birth rate equals the completion rate with a probability of 1. Also, these results confirmed the findings from the literature reviews.

Further investigation in the branching process for births and assassination process should limit the ratios of **κ** = 1 combinations to kappa values < 1 with lambdas values < 5. These parameters should provide the best lambda and kappa values of probability 1 to be able to complete a Poisson branching processes.

In social and industry applications the possibilities are endless, e.g. when considering how this could be applied to networks or wireless systems. To study ad-hoc wireless networks, interacting space-time models were investigated by evaluating the spatial birth-death process that modeled spectrum sharing of an infinite number of queues to effectively model a large wireless system [5]. They explored the potential to boost the service performance of wireless networks without incurring additional infrastructure cost by capitalizing on several networks operating on orthogonal bandwidths or multiple wireless technologies pooled by mobile virtual network operators. This heuristic association scheme may be an alternative to the standard rules applied in metropolitan cities. Further investigation is warranted in understanding and validating the alternative models for spatial-time network processes.

Also, branching process can provide important information on the study of social networks by being able to measure various virilities like the probability of virility, expected number of shares, and the rate of growth of expected shares; which may help predict the emergence of global macro properties from the laws and parameters that determine local interactions. Local interactions depend greatly upon structure timelines like holding content or number of friends of users of the network.

These continuous-time branching process evolving on a fixed network can help determine the relative importance of model parameters to the growth of the population on the network; which could then be applied to modeling the likelihood of early stages of an flu like epidemic spreading across cities and its total size.

## **References**

1. Nolan, D. and Lang, D. T. “Data Science in R.” CRC Press, 2015 (Chapter 7).
2. <https://www.investopedia.com/terms/m/montecarlosimulation.asp#ixzz5MDUf2yo8>
3. <https://www.stat.berkeley.edu/~aldous/Papers/me44.pdf>
4. <https://projecteuclid.org/download/pdf_1/euclid.ejp/1464819139>
5. <http://abishek90.github.io/QualReport.pdf>

## **Appendix - R Code**

### Case Study

# Make parameters in genKidsU as vectors

genKidsV = function(bTimes, cTimes, lambda = 0.5, kappa = 0.3) {

# bTimes & cTimes - vector of birth and completion times

# Determine how many children each job has

parentAge = cTimes - bTimes

numKids = rpois(n = length(parentAge),

lambda = lambda\*parentAge)

# Determine the birth and completion times of the children

mapply(function(n, min, max) {

births = sort(runif(n, min, max))

runtimes = rexp(n, rate = kappa)

completes = rep(max, n) + runtimes

data.frame(births, completes) },

n = numKids , min = bTimes, max = cTimes,

SIMPLIFY = FALSE)

}

# return a single dataframe

genKidsV2 = function(bTimes, cTimes, lambda = 0.5, kappa = 0.3, parentID)

{

if (is.null(bTimes)) return(NULL)

# Call current function

kidStats = genKidsV (bTimes, cTimes, lambda, kappa)

if (is.null(kidStats)) return(NULL)

# get numkids from return value

numKids = sapply(kidStats, nrow)

if (is.null(numKids)) return(NULL)

SumNumKids = sum(numKids)

if (is.null(SumNumKids)) return(NULL)

if (SumNumKids == 0) return(NULL)

# calculate new vars

data.frame(parentID = rep(parentID, numKids),

kidID = 1:sum(numKids),

births = unlist(sapply(kidStats, "[[", "births")),

completes = unlist(sapply(kidStats,"[[", "completes")))

}

# terminates the process once a specified threshold

# for the number of offspring is exceeded

familyTree = function(lambda = 0.5, kappa = 0.3,

maxGen = 10, maxOffspring = 1000) {

# Return value - a list with 1 data frame per generation.

allGens = vector(mode = "list", length = maxGen)

# Generate root of the tree

allGens[[1]] = data.frame(parentID = NA, kidID = 1,

births = 0,

completes = rexp(1, rate = kappa))

currentNumOffspring = 0

# Generate future generations, one at a time.

for (i in 2:maxGen) {

nextGen = genKidsV2(bTimes = allGens[[ (i - 1) ]]$births,

cTimes = allGens[[ (i - 1) ]]$completes,

parentID = allGens[[ (i - 1) ]]$kidID,

lambda = lambda, kappa = kappa)

if (is.null(nextGen)) return(allGens[ 1:(i - 1) ])

allGens[[ i ]] = nextGen

currentNumOffspring = currentNumOffspring + nrow(nextGen)

if (currentNumOffspring > maxOffspring)

return(allGens[1:i])

}

allGens

}

# automate the call to familyTree

exptOne = function(l, k, mG, mO){

# Helper function to call familyTree

# Returns - summary statistics for analysis,

aTree = familyTree(lambda = l, kappa = k, maxGen = mG,

maxOffspring = mO)

numGen = length(aTree)

numJobs = sum(sapply(aTree, nrow))

c(numGen, numJobs)

}

# create function to replicate several calls

MCBA = function(params, repeats = 5, mG = 10, mO = 1000){

# params: matrix columns of lambda and kappa values

# For each lambda and kappa pair, run "repeats" times

n = nrow(params)

mcResults = vector("list", length = n)

for (i in 1:n) {

cat("param set is ", i, "\n")

mcResults[[i]] = replicate(repeats,

exptOne(l = params[i, 1],

k = params[i, 2],

mG = mG, mO = mO))

}

mcResults

}

# set initial values

seed2 = 12212013

set.seed(seed2)

lambdas\_base = c(0.1, 0.2, 0.3333, 0.5, 0.6666, 1.0, 1.5, 2.0, 3.0, 5.0, 10.0)

lambdas = c(lambdas\_base, 1.5\*lambdas\_base, 2\*lambdas\_base)

kappas = c(rep(1,11), rep(1.5,11), rep(2,11))

paramGrid = as.matrix(expand.grid(lambdas, kappas))

# test with num of offsprings limited to 1000

p\_repeats = 50

p\_mG = 20

p\_mO = 1000

mcGrid = MCBA(params = paramGrid, repeats = p\_repeats, mG = p\_mG,

mO = p\_mO)

# plot generations vs offspring by lambda\*kappa

oldPar = par(mfrow = c(2, 2), mar = c(3,3,1,1))

mapply(function(oneSet, lambda, kappa) {

plot(x = oneSet[2,], y = jitter(oneSet[1, ], 1), log = "x",

ylim = c(1,20), xlim = c(1, 10^7), pch = 19, cex = 0.6)

text(x = 50, y = 15, bquote(paste(lambda == .(lambda))) )

text(x = 300, y = 15, bquote(paste(kappa == .(kappa))) )

},

mcGrid, lambda = lambdas, kappa = kappas)

par(oldPar)

# 3dplot of 75% percentile

# compute the log upper quartile of the number of children

logUQkids = sapply(mcGrid, function(x)

log(quantile(x[2, ], probs = 0.75), base = 10))

# distinguish between trees that die out and those that

# surpass the 1,000 offspring limit

UQCut = cut(logUQkids, breaks = c(-0.1, 0.5, 2, max(logUQkids)) )

color3 = c("#b3cde3aa", "#8856a7aa", "#810f7caa")

colors = color3[UQCut]

library(scatterplot3d)

sdp = scatterplot3d(x = paramGrid[ , 1], y = paramGrid[ , 2],

z = logUQkids, pch = 15, color = colors,

xlab = "Lambda", ylab = "Kappa",

zlab = "Upper Quartile Offspring",

angle = 120, type="h")

legend("topright", inset = .0, bty = "n", cex = 0.8,

legend = c("[0, 0.5)", "[0.5, 2)", "[2, 5)"),

fill = color3)

#set initial values for simulation 2

seed2 = 12212013

set.seed(seed2)

lambdas\_base = c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.25, 2.5,

2.75, 3.0, 3.5, 4.0, 4.5, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0)

lambdas = c(lambdas\_base, 2\*lambdas\_base, 3\*lambdas\_base)

kappas = c(rep(1,26), rep(2,26), rep(3,26))

paramGrid = as.matrix(expand.grid(lambdas, kappas))

mcGrid = MCBA(params = paramGrid, repeats = 400, mG = 20,

mO = 1000)

save(mcGrid, file = "mcGridOutput4.rda")

# scatterplots simulation 2

logUQkids = sapply(mcGrid, function(x)

log(quantile(x[2, ], probs = 0.75), base = 10))

UQCut = cut(logUQkids, breaks = c(-0.1, 0.5, 2, max(logUQkids)) )

color3 = c("#b3cde3aa", "#8856a7aa", "#810f7caa")

colors = color3[UQCut]

library(scatterplot3d)

sdp = scatterplot3d(x = paramGrid[ , 1], y = paramGrid[ , 2],

z = logUQkids, pch = 15, color = colors,

xlab = "Lambda", ylab = "Kappa",

zlab = "Upper Quartile Offspring",

angle = 120, type="h")

legend("topleft", inset = .08, bty = "n", cex = 0.8,

legend = c("[0, 0.5)", "[0.5, 2)", "[2, 5)"),

fill = color3)

# 3D surface/ 2D contour maps of proprotions

mcGridAlive = sapply(mcGrid, function(oneParamSet) {

sum((oneParamSet[1,] == 20) | (oneParamSet[2,] > 1000)) /

length(oneParamSet[2,]) })

z=matrix(mcGridAlive, nrow = length(lambdas),

ncol = length(kappas))

x=lambdas

y=kappas

data <- data.frame(x=lambdas, y=kappas, z)

df.dataframe <- data.frame(x=lambdas, y=kappas, z=matrix(mcGridAlive, nrow = length(lambdas),

ncol = length(kappas)))

library(plotly)

plot\_ly(df.dataframe, x=x, y=y, z=z, type="surface")

mcGridProp20kids = sapply(mcGrid, function(oneParamSet) {

sum(oneParamSet[2,] > 19) / length(oneParamSet[2,]) })

mcGridProp20kidsMat = matrix(mcGridProp20kids,

nrow = length(lambdas),

ncol = length(kappas))

z=matrix(mcGridProp20kids,

nrow = length(lambdas),

ncol = length(kappas))

df2.dataframe <- data.frame(x=lambdas, y=kappas, z=matrix(mcGridProp20kids,

nrow = length(lambdas),

ncol = length(kappas)))

p <- plot\_ly(z = ~z, type = "contour", xaxis = "lambdas", yaxis = "kappas")

library(plotly)

plot\_ly(df2.dataframe, x=x, y=y, z=z, type="surface")