Song Evolution Model Walk-Through[*]

C# Version: 1.0

C. Robinson

May 2019

[*]Published in conjunction with Robinson and Creanza 2019 (Pending)

Contents

1 Overview of the Model

1.1 Description

The Song Evolution Model initializes a population of male birds with several song-learning traits and one song, which exist in a matrix with discrete boundaries. Each time step, birds can (A) die, or survive to potentially (B) learn and/or (C) father chicks.

A) Birds are first chosen to die randomly. However, birds with greater learning age thresholds and can learn for longer periods or time have a greater chance of being chosen to die than those with smaller learning age thresholds. The amount this chance is increased is dependent on both the size of the learning age threshold and the value of the learning penalty parameter. The number of birds chosen to die is either 1) a set percentage or 2) a proportion of the birds in each age group depending on the maximum age of the population.

B) If oblique learning is enabled, any remaining birds that are younger than their learning age threshold have the capacity to learn. Learners chose a tutor male from the population and learn all or part of the tutor's song using one of four strategies: 1) Add, 2) AddForget, 3) Forget, or 4) Consensus.

C) After the learning step, vacancies left by deceased birds are filled by generating chicks fathered by the remaining birds. Birds are chosen as fathers based on sexual selection pressures which include: 1) preference for larger repertoire sizes, 2) preference for repertoires that match a song template, and/or 3) uniform preference for factors not coded into the model, such as ornate plumage or dance (called "noise preference" in the model). Fathers may also be preferentially chosen based on their location in the matrix. Birds chosen to be fathers produce one or more chicks, which inherit their father's song-learning traits with a parameterized amount of random noise. Chicks start with an empty song template and, if enabled, vertically learn their father's song with an accuracy based on their song-learning ability.

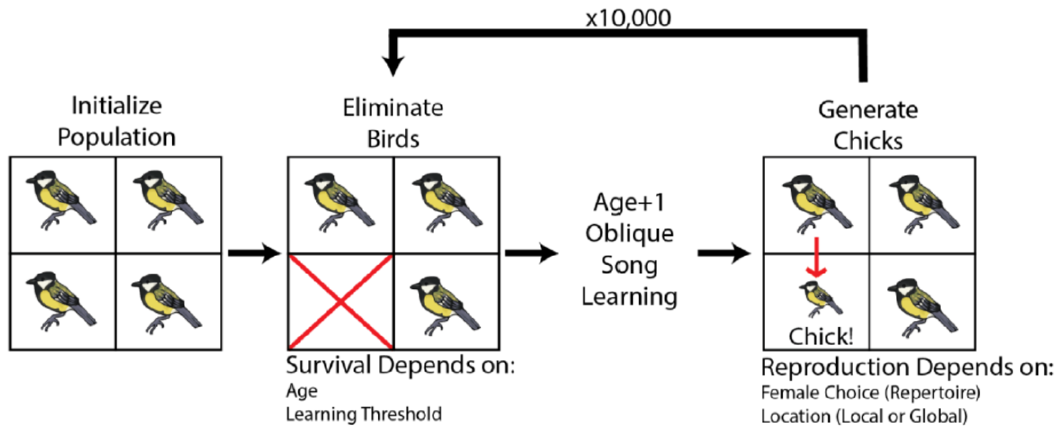


Figure 1: Schematic of the model.

1.2 Conceptualizing the Bird Matrix

Although a matrix is never literally created during the simulation, the code operates as though birds are in a matrix with discrete edges. **Figure 2** shows the assumed positions of birds [0,99] if the matrix dimensions are 10 rows by 10 columns. Thus, if birds are considered to be local when they are one "step" or territory away from a target bird, corners will have three local territories (illustrated by cell 0 in panel A), edges will have 5 local territories (illustrated by cell 6), and center pieces will have 8 local territories(illustrated by cell 67) (blue in **Figure 2A**). The combination of the blue and red territories in **Figure 2A** shows the local birds if local territories can be two steps away. When there is more than one dialect in the simulation, dialect regions are constructed such that each region contains an equal number of birds, and the region itself is as square as possible (**Figure 2B** shows 4 dialects). This means that the number of dialects in a matrix must be a factor of the total number of territories. For the 10x10 matrix example, that means that 1, 2, 4, 5, 10, 20, 25, 50, and 100 are acceptable numbers of dialects. However, adding more than 5 dialects to such a small matrix is likely to negate the purpose of adding dialects.

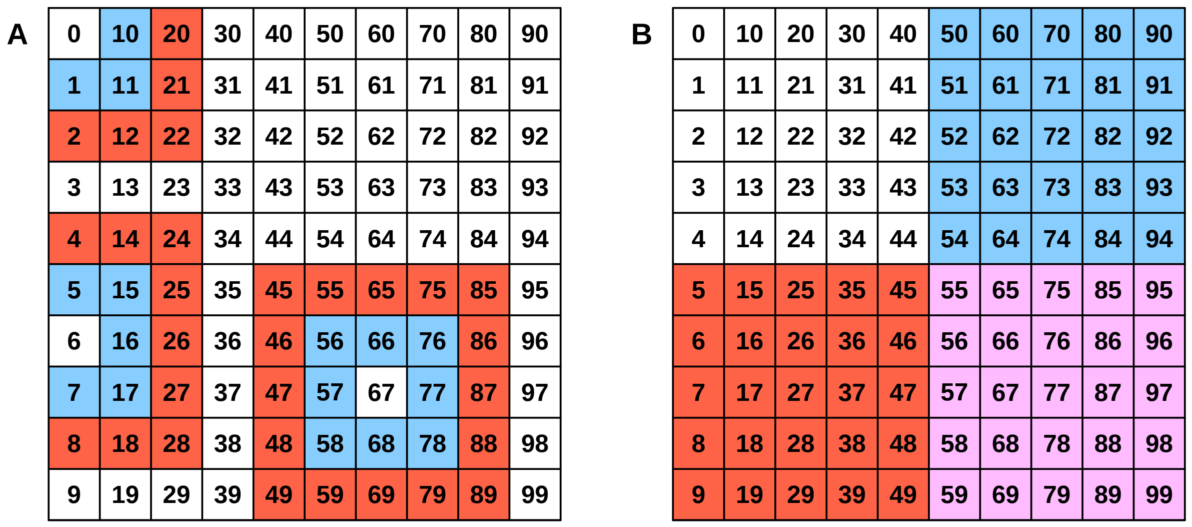


Figure 2: Matrix (A) shows which territories are considered local when Par.Steps=1 (blue) or 2 (blue and red) for a corner (0), edge (6) and center piece (67). Matrix(B) shows how 4 dialects would be positioned in this matrix. Dialect 1=white, 2=red, 3=blue, 4=purple.

1.3 Implementing a Type II Survival Curve

1.3.1 Overview of Type II Survival Curves

Chicks often have a low chance of surviving their first year (e.g. [1], [2], [3], and [4]). However, after the first year, mortality in birds is best modeled by a type II survival curve [5], wherein the mortality rate is not dependant on an organism's age. Instead, a constant proportion of individuals are lost every year. On a semi-log plot, where the y axis has been log-transformed, a Type II survival curve appears to be a straight line (Fig 3A).

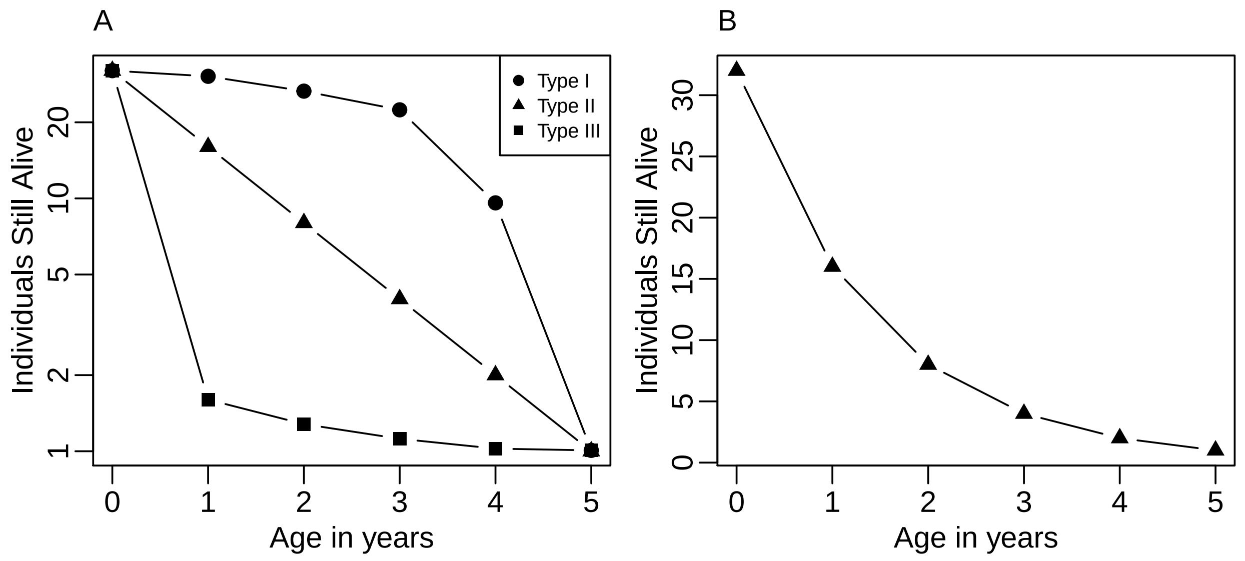


Figure 3: Survival Curves. A) Type I survival curves model organisms with a low mortality rate when young with a high mortality when older. Type III survival curves fit organisms with a high mortality rate when young, and a low mortality rate when older. Type II survival curves show no age dependence; a constant proportion of individuals die every year. In semi-log space, Type II curves appear linear. In linear space, Type II curves are not linear and approach, but never reach, 0.

Despite the appearance of linearity on a semi-log plot **Figure 3A**, on normal axes, a type II survival curve is indeed a curve, as can been seen in linear space **Figure 3B**. This curve fits an exponential equation of the form:

$$y = n_1 P^x \tag{1}$$

Where:

$$n_1 = \text{the number of age 1 adult males in a generation} \tag{1a}$$

$$P = \text{the proportion of adult males lost at each time step} \tag{1b}$$

and

$$0 < P < 1 \tag{1c}$$

1.3.2 Implementation and Model Assumptions

To mimic the mortality rates seen in real birds, our model allows a user-defined proportion of male chicks to survive to year 1 and then calculates mortality at subsequent ages using a Type II survival curve. This affects two components of our model. It is used to calculate what proportion of males in a given age step will survive to the next age step. It is also used to determine the initial distribution of males present in each age group at the start of the simulation. It is used at this initialization step, because if the initial population of birds is inappropriately distributed throughout the age classes, this leads to different numbers of birds dying in each time step. This, in turn, creates chaotic generation sizes. This unwanted chaos can be avoided if the ages of the initial generation of birds are distributed as defined by (1), which would be the case if the population had reached a mortality equilibrium.
The design of our model imposes the following constraints:

1. There is a set number of territories as paramaterized by the user. Each territory is claimed by a single male bird. There are no males without territories. Thus, we know total population of male birds (N).

2. There is maximum lifespan for all birds, which is parameterized by the user (m). No birds are permitted to survive past age m.

3. Because Equation (1) never reaches zero, a death threshold must be defined (t) at which point the probability of survival is considered to be 0.

4. In adult birds, the proportion of birds lost at a given time step (P) is equal to the proportion lost at every other time step in a generation except from age group 0 to 1.

5. The proportion of chicks lost in the 0 to 1 step is parameterized by the user (P_c)

If n_0 is known, which is the case for all generations after the initial time step, we can also calculate n_1 ($C * n_0$). Subsequent survival (n_2 to $n_m + 1$) is calculated using (1) ($|_0^{m+1}$). This requires knowing P, which can be calculated in this case, because at age $n_{m+1} = t$. However, we do not know the constants n_0 or P for generating the initial age distribution of adults. Thus, additional logic is required to solve the equation in that case.

1.3.3 Finding P and n0 for the Initial Age Distribution

Because of the way we have defined our model, we know that the total number of males in the population (N) must be equal to the number of birds in each age step (n_i) summed:

$$N = \sum_{i=0}^{m+1} n_i \tag{2}$$

For an example case, when the maximum age for the population is 2 years ($m = 2$):

$$N = n_0 + n_1 + n_2 + n_3 \tag{3}$$

Where n_0 is the number of newly generated chicks that survive, and n_1 and n_2 are the number of adults from previous generations remaining at ages 1 and 2 respectively. n_3 is the number of birds of age 3, which logically would be 0, because no birds live to age 3 when $m = 2$. However, type-II survival curves can only approach 0, so n_3 is a non-zero value that will be equivalent to the death threshold t. We know that n_1 is some proportion of n_0, n_2 is some proportion of n_1, and n_3 is some proportion of n_2. We can use the property of multiplicative identity to rewrite this equation as:

$$N = n_0 + \frac{n_1}{n_0} n_0 + \frac{n_2}{n_1} n_1 + \frac{n_3}{n_2} n_2 \tag{4}$$

To simplify, these proportions can be rewritten as p_{i-1}:

$$N = n_0 + p_1 n_0 + p_2 n_1 + p_3 n_2 \tag{5}$$

$n_i = p_i n_0$, so the equation cane be rewritten such that only n_0 remains:

$$N = n_0 + p_1 n_0 + p_1 p_2 n_0 + p_1 p_2 p_3 n_0 \tag{6}$$

Assume $p_1 = P_c$, the proportion of chicks that survive, which is a known value. All other proportions are equal, because one of the defining traits of a Type II survival curve is that a constant proportion of individuals die at each time step. To take advantage of this, we will assume that the population has stabilized its generation size, such that n_0 was equivalent for each generation. Therefore, p_2 and p_3 are equivalent (P). The equation can now be rewritten as:

$$N = n_0 + P_c n_0 + P_c P n_0 + P_c P^2 n_0 \tag{7}$$

We can take this example and generalize it to:

$$N = n_0 + n_0 P_c \sum_{i=0}^{m} P^i \tag{8}$$

At this point, n_0 and P are still unknowns. However, as explained in Section 1.3.2:

$$t = n_{m+1} \tag{9}$$

We showed in equations (3)-(7) that we can know the number of individuals in any age class using the following:

$$n_i = n_0 P_c P^{i-1} \tag{10}$$

This logic combined with (9) allows us to create the equation:

$$t = P_c P^m n_0 \tag{11}$$

Solving (11) for n_0 allows us to rewrite (8) as:

$$N = \frac{t}{P_c P^m} + \frac{t}{P_c P^m} P_c \sum_{i=0}^{m} P^i \tag{12}$$

This can be simplified and rewritten as:

$$0 = (t - N) + t\frac{1}{P} + t\frac{1}{P}^2 + ... + (t + \frac{t}{P_c})\frac{1}{P}^m \tag{13}$$

Thus, the equation can be solved for $1/P$, to obtain a real, positive root that can be converted to P. Thus, only unknown variable in (11) is now n_0, so this equation can be solved when given user defined t, m, P_c and N.

2 Using the Code

2.1 Programs

The bulk of the code exists in the SongEvolutionModelLibrary, which is called by various programs that run the simulations in parallel and collate the data. All of these programs load existing parameter files (**.SEMPs**). If a user wishes to perform a different type of simulation or obtain different data than what is possible given these programs, we advise that the user create a new C# project that uses the SongEvolutionModelLibrary namespace.

2.1.1 IntervalSims

This program runs a full simulation, but only saves data at time step 0 and every [Frequency]th step after that. Averages are saved, not the raw values for every male in the population. Takes up to 5 arguments:

1. ParamPath (Required): Path to folder of .SEMPs
2. OutputPath (Required): Path to folder where data should be saved
3. MaxParallel (Default=4): Maximum number of simultaneous processes
4. Frequency (Default=200): How many time steps pass before data is saved
5. Repeats(Default=50): Times to repeat a simulation for a .SEMP

2.1.2 InvasionSims

This program runs a [burnIn] step simulation, then picks one or more random birds in the matrix to be invaders. Each invader has one of its song learning traits ([type]) altered, and its age is set to 1. The simulation continues until 1) invaders are extinguished, 2) they overtake the population, or 3) 400 time steps have been completed, whichever happens first. Returns the average value for the target song learning trait [type] and the number of steps completed post invasion. The inheritance noise for [type] should be set to 0, or else this simulation may not run as expected (you are warned if inheritance is not 0). Takes up to 8 arguments:

1. ParamPath (Required): Path to a folder of .SEMPs
2. OutputPath (Required): Path to folder where data should be saved
3. Type (Required): The trait to test invasion on. Can be Learning, Accuracy, Forget, or Invent.
4. InvaderStat (Required): Set value for invader learning trait
5. NumInvaders (Default=1): Number of invaders created
6. MaxParallel (Default=4): Maximum number of simultaneous processes

7. Repeats(Default=50): Times to repeat a simulation for a .SEMP
8. BurnIn (Default=500): Time steps completed before introducing invaders

2.1.3 LearningInvasion

This program serially calls the InvasionSims program to run the all of invasion experiments in Robinson and Creanza 2019 full citation pending publication. Given a folder that contains the sub-folders "Closed" (Learning Threshold = 0.25), "DelayedClosed" (Learning Threshold = 1), and "Open" (Learning Threshold = 2), it allows every .SEMP in a sub-folder to be tested with 1 or 4 invaders. Invaders will have one of the other two learning age thresholds; i.e. invaders entering a closed population will have learning age thresholds of either 1 or 2 (4 simulation sets for each .SEMP). Each simulation set is run 1000 times with a BurnIn of 1000. Takes 2 arguments:

1. args[0] (Required): Path to InvasionSims.dll
2. args[1] (Required): Path to the folder described above

Note: Learning Invasion is finicky about file paths. It did not work with relative file paths in Linux, so I imagine it will also not take relative paths on Macs. Relative paths seemed to work fine on Windows 10, but absolute paths worked fine for the OSs I could test however. The program also did not work when the path included special characters on Linux (e.g. C#) and spaces may also cause problems.

2.2 SongEvolutionModelLibrary Classes

1. **Simulations.cs** contains the code required to run a specific type of simulation and returns the generated data. Each simulation type generates an instance of SimParams (Par), which is in turn used to make an instance of the Population class (Pop). Pop is then modified based on the information in Par for Par.NumSims time steps using the BirthDeathCycle class functions. Depending on the simulation type, the data is either returned as a WriteData class or an InvasionData structure.

2. **SimParams.cs** is used to construct Par. Par can be generated *de novo* using the defaults in SimParams, or by modifying the arguments in SimParams (warning: there are over 40 arguments!). Alternatively, Par can be loaded from a .SEMP text file generated by either the C# or R version of the model.

3. **Population.cs** is used to construct Pop and requires a SimParams object. Population contains a series of lists and arrays for each trait a male bird/territory can have in a simulation. Age, Local, and MaleSong, FemaleSong, and Match are generated using Ages.cs, Locations.cs, and Songs.cs respectively.

4. **Ages.cs** is used to randomly assign an appropriate age to every male bird in the population.

5. **Locations.cs** is used to generate a data structure that lists which territories in the bird matrix are considered “local” to a target territory, and which territories are N steps further away.

6. **Songs.cs** is used to generate the MaleSong for each territory in the matrix. If Par.MatchPreference > 0, this class will also generate FemaleSong for each territory in the matrix and calculate the Match between the MaleSong and FemaleSong belonging to each territory.

7. **BirthDeathCycle.cs** is used to modify Pop each time step. It calculates which birds will die, calls Learning.cs functions to allow birds to learn obliquely, chooses fathers, and calls Pop.ReplaceBird() to exchange dead birds with newly born chicks.

8. **Learning.cs** contains the functions for both vertical and oblique learning. This includes testing which males are capable of learning, choosing tutors, and modifying learner songs.

9. **WriteData.cs** is used to save data from the simulation and write it to .csv files for subsequent analysis.

10. **Utils.cs** contains functions for use in the programs to simplify saving data and loading .SEMPs.

2.3 .SEMP Files

<u>S</u>ong **<u>E</u>**volution **<u>M</u>**odel **<u>P</u>**aramterset (.SEMP) files are text files in the format of [parameter]=[value]. Both the R and C# versions of the model can generate and read .SEMPS that are cross-language compatible. The parameters in .SEMPs are:

R=Rows in the bird matrix
int, [3,∞]; Default=20

C=Columns in the bird matrix
int, [3, ∞]; Default=20

numBirds=NumBirds, number of birds in the matrix
int, R*C

Steps=Steps, number of territories away that are considered local
int [1,max(R,C)-1]; Default=1

RSize0=InitialSyllableRepertoireSize, the number of syllables a bird has a 90% chance to know at the start of the simulation
int, [1,MaxRSize]; Default=5

PerROh=PercentSyllableOverhang, this fraction*RSize0 is the number of syllables that a bird has a 10% and 1% chance to know at the start of the simulation
float, [0,(MaxRsize-RSize0)/(2*RSize0)]; Default=0.2

MaxRSize=MaxSyllableRepertoireSize, the limit to the number of syllables an individual bird in the population can learn; Default=500

Acc0=InitialAccuracy, the mean starting value for the population accuracy
float, [MinAcc,MaxAcc]; Default=0.7

IAccN=InheritedAccuracyNoise, the range around the mode (initial accuracy or the father's accuracy) that can be sampled to pull an initial male's accuracy or a chick's inherited accuracy, respectively
float, [0,(MaxAcc-MinAcc)/2]; Default=0.15

MinAcc=MinimumAccuracy, the absolute minimum value for accuracy
float, [0,MaxAcc); Default=0

MaxAcc=MaximumAccuracy, the absolute maximum value for accuracy
float, (MinAcc,1]; Default=1

MAge=MaxAge, the maximum age
int,[1,∞]; Default=20

LrnThrsh0=InitialLearningThreshold, the mean starting value for the population learning age
float, [MinLrn,MaxLrn]; Default=2

ILrnN=InheritedLearningThresholdNoise, the range around the mode (initial learning threshold or the father's learning threshold) that can be sampled to pull an initial male's learning threshold or a chick's inherited learning threshold, respectively
float, [0,(MaxLrn-MinLrn)/2]; Default=0.25

MinLrn=MinimumLearningThreshold, the absolute minimum value for the learning age threshold
float, [0,MaxLrn); Default=0

MaxLrn=MaximumLearningThreshold, the absolute maximum value for the learning age threshold
float, (MinLrn,MaxAge]; Default=MaxAge

CtI0=InitialChancetoInvent, the mean starting value for the population chance to invent; It is not the final percentage
float, [MinCtI,MaxCtI]; Default=10

ICtIN=InheritedChancetoInventNoise, the range around the mode (initial chance to invent or the father's chance to invent) that can be sampled to pull an initial male's chance to invent or a chick's inherited chance to invent, respectively
float, [0,(MaxCtI-MinCtI)/2]; Default=0

MinCtI=MinumumChancetoInvent, the absolute minimum value for chance to invent
float, [0,MaxCtI); Default=0

MaxCtI=MaximumChancetoInvent, the absolute maximum value for chance to invent
float, (MinCtI,1]; Default=1

CtF0=InitialChancetoForget, the mean starting value for the population chance to forget
float, [0,MaxCtF/2]; Default=0.2

ICtFN=InheritedChancetoForgetNoise, the range around the mode (initial chance to forget or the father's chance to forget) that can be sampled to pull an initial male's chance to forget or a chick's inherited chance to forget, respectively
float, [0,(MaxCtF-MinCtF)/2]; Default=0

MinFtI=MinumumChancetoForget, the absolute minimum value for chance to forget
float, [0,MaxCtF); Default=0

MaxFtI=MaximumChancetoForget, the absolute maximum value for chance to forget
float, (MinCtF,1]; Default=1

LisThrsh=ListeningThreshold, if less than 1, the percentage of a song that a learner hears. 0.999 is treated as 100%. If an integer greater than 1, the number of syllables a learner hears
float $[0,0.999]\cup$ints $[1,\infty]$; Default=7

FLisThrsh=FatherListeningThreshold, if less than 1, the percentage of a song that a son hears. 0.999 is treated as 100%. If an integer greater than 1, the number of syllables a son hears
float [0,0.999]∪ints [1,∞]; Default=0.999

MinLrnSyl=MinLearnedSyllables, the minimum number of syllables a male learns when a percentage strategy is employed for one or both of the listening thresholds
ints [0,∞]; Default=7

EnSuc=EncounterSucess, the chance that a learner meets a tutor
float [0,1]; Default=0.95

Lpen=LearningPenalty, the magnitude of the survival disadvantage placed on birds with learning thresholds greater than 1
float, [0,∞]; Default=0.5

DStrat=DeathStrategy, if true, death is based on the dynamics of a population with a type II survival curve, if false, every timestep PDead*numBirds birds die
bool ; Default=true

PDead=PercentDeath, if DStrat=false, the percentage of birds that die each time step.
float[0.01,0.9]; Default=0.1

DeadThrsh=DeathThreshold, when the number of birds in a generation is less than or equal to the Death Threshold, they are all considered dead; a key component for calculating the type II survival curve. This calculation may prevent birds from reaching the MaxAge if the threshold is set below 1.
float[1,∞]; Default=1

Pc=ChickSurvival, the percentage of chicks that survive to age 1
float [.1,1]; Default=0.3

InitProp=InitialSurvival, the proportion of adult birds that will survive to the next age. It is calculated internally based on the MaxAge, ChickSurvival, and DeathThreshold. Do not modify this in .SEMPs unless you want the birds to start in a non-equilibrium state, with nonuniform generations sizes.
float, **See Calculating a Type II Survival Curve**

ScopeB=LocalBreeding, whether mates are chosen from a local pool, or among all males
bool; Default=false

ScopeT=LocalTutor, whether tutors are chosen from a local pool, or among all males
bool; Default=false

Consen=Consensus, whether birds use the consensus strategy in learning. Automatically sets add=true and forget=true, but changing either one to false manually

will be accepted by the code.
bool

ConsenS=ConsensusStrategy, which variant of the consensus strategy to use. Percentage means there is a linear association between the commonality of a syllable and whether a learner attempts to learn it. AllNone means a learner only learns syllables that all tutors sang. Conform means that a learner attempts to learn a syllable based on a sigmoidal relationship.
Default=Conform

Add=Add, whether birds use the add strategy in learning
bool

Forget=Forget, whether birds use the forget strategy in learning
bool

ConNoTut=NumTutorConsensusStrategy, if using the consensus strategy, the number of tutors to find
int [1,numBirds-1]; Default=3

OvrLrn=OverLearn, whether chicks use the overlearn strategy
bool; Default=false

OLNoTut= numTutorOverLearn if using the overlearn strategy, the number of tutors to find
int [1,numBirds-1]; Default=3

Obliq=ObliqueLearning, whether birds engage in oblique learning
bool; Default=true

Vert=VerticalLearning, whether birds engage in vertical learning
bool; Default=true

VertLrnCut=VerticalLearningCutOff, birds with a learning threshold below this value cannot learn vertically
float [0,MaxAge]; Default=0.25

RepPref=RepertoireSizePreference, the percentage of female choice devoted to finding males with the largest repertoire sizes
float [0,1-(MatPref+NoisePref)]; Default=1

LogScl=TRUE

MatPref= MatchPreference, the percentage of female choice devoted to finding males with repertoires that best match her template
float [0,1-(RepPref+NoisePref)]; Default=0

NoisePref=NoisePreference, the percentage of female choice devoted to finding males who are superior in other traits (e.g. better dancing or territory quality)
float [0,1-(RepPref+MatPref)]; Default=0

UniMat=MatchUniform, whether all females in a dialect region have the same (true), or if there is noise between templates (false)
bool; Default=true

MScl=MatchScale, not currently implemented. This will hopefully allow for scaling the matches in a non-linear way in the future.

Dial=NumDialects, the number of different dialects in the matrix. Different dialects have no overlap in the syllable space
int [0,NumBirds], must be a factor of NumBirds, Default=1

MDial=MaleDialects, whether males in the starting generation have dialects matching females. If "None," male songs are generated in the first slots of the syllable space, putting them in the same syllable space as dialect 1. If "Same," a male's song is a duplicate his female's song template. If "Similar," a male's song is generated within the same syllable space as his female's song template, but a new song is generated, so they are not duplicates. Note that is there is only one dialect, then "None" and "Similar" produce the same effect.
string ["None","Same","Similar"]; Default="None"

FEvo=FemaleEvolution, whether females can change their templates over time. If true, when a male dies, his female also dies. When a male chick is created, a new female with a template matching different father is created.
bool; Default=false

ChoMate=ChooseMate, allows females to pick a new mate based on their preferences each time step.
bool; Default=FALSE

SMat=SaveMatch, whether to save matching data. If null, the code decides whether to save based on other parameters.
nullable bool; Default=null

SAcc=SaveAccuracy, whether to save accuracy data. If null, the code decides whether to save based on other parameters.
nullable bool; Default=null

SLrn=SaveLearningThreshold, whether to save learning age threshold data. If null, the code decides whether to save based on other parameters.
nullable bool; Default=null

SCtI=SaveChancetoInvent, whether to save chance to invent data. If null, the code decides whether to save based on other parameters.
nullable bool; Default=null

SCtF=SaveChancetoForget, whether to save chance to forget data. If null, the code decides whether to save based on other parameters.
nullable bool; Default=null

SNam=SaveNames, whether to save name and father name GUID data.
bool; Default=false

SAge=SaveAge, whether to save age data.
bool; Default=false

SMSng=SaveMSong, whether to save male song data.
bool; Default=false

SFSng=SaveFSong, whether to save female song data.
bool; Default=false

SimStep=SimStep, the current step in the simulation. Used for error handling and reset to 1 when the parameters are saved.
int [1,∞]; Default=1

nSim=NumSim, the number of timestpes to run in a simulation.
int[1,∞]; Default=1000

Seed=Seed, for reproducibility. 0 is not an option, because NA is converted to 0 in this code due to strong typing, and thus 0 is considered to mean that no seed was passed.
int [1,∞] or NA; Default=NA

2.4 Vignette

This is a quick set of directions to get one of the programs running on an example .SEMP file. Repeat steps 4 and 5 to prepare the other programs to run on your machine. Be aware that steps 3 and 5 build the code for a specific OS. If you try to move the release build to a different OS, it will not work; you have to rebuild it on that OS. Note that LearningInvasion relies on InvasionSims to function.

1. If your computer is not already set up to use C# code, do so:
   https://docs.microsoft.com/en-us/dotnet/core/tutorials/with-visual-studio-code

2. Download the SEM Folder from GitHub:
   CreanzaLab/SongEvolutionModel/Csharp-Library-and-Programs

3. Open the SongEvolutionModelLibrary Folder in Visual Studio (VS) Code. Do not be concerned if the IDE shows that the code has errors; there are missing files that will be generated when the project is built. If not already open, open VSCode's terminal. Type into the terminal:

   ```
   dotnet build --configuration Release
   ```

   There should now be a new folder at SongEvolutionModelLibrary/bin/Release/netcoreapp2.1 that contains three files: a .deps, .dll, and .pdb.

4. Open the IntervalSims.csproj file in Visual Studio Code and make sure that line 18 points to the correct location of SongEvolutionModelLibrary.csproj on your computer. This should be the case if you did not change the folder structure from GitHub. If it does not match, then update the path.

5. Open the IntervalSims Folder in Visual Studio Code. Again, do not be concerned about errors. Type into the terminal:

   ```
   dotnet build --configuration Release
   ```

   There should now be a new folder at IntervalSims/bin/Release/netcoreapp2.1 that contains seven files: a .deps, 2 .dlls, 2 .JSON files, and 2 .pdbs.

6. Open the IntervalSims/bin/Release/netcoreapp2.1 folder. Run the code using the following command:

   ```
   dotnet IntervalSims.dll ..\..\..\..\..\Vignette ..\..\..\..\..\Output 1
   ```

   As explained in the programs section, the arguments after IntervalSims are: 1) the path to a folder of .SEMP files, the folder in which to output the data, the maximum parallelization, the frequency with which to get data from time steps, and the number of times to repeat the simulation. This combination uses the folder structure from GitHub to run one Vignette.SEMP one time, taking data each time step.

   The terminal should print the following (numbers will change depending on how long the simulation took to run):

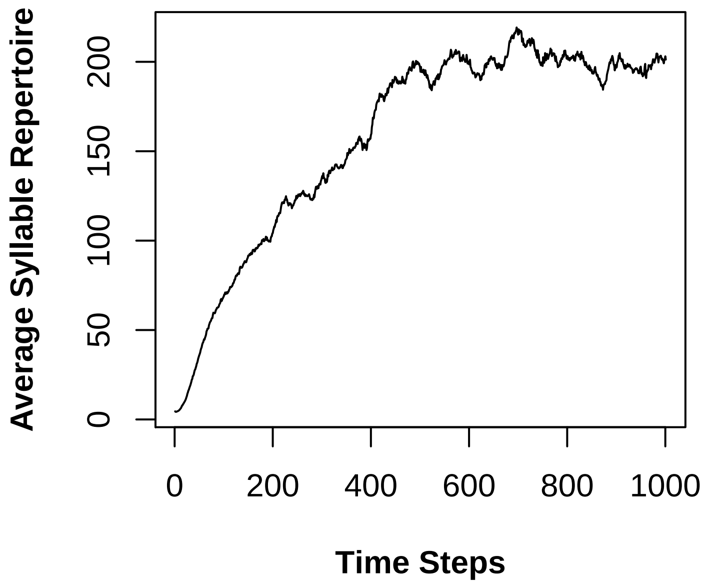
```
Start
..\..\..\..\..\Vignette\Vignette.semp
..\..\..\..\..\Vignette\Vignette.semp-2989
3004
All simulations completed.
```

7. Once the simulation completes, check the Output folder. There should be four files, 3 .csvs and 1 .SEMP. Acc = Accuracy, LrnThsh=Learning Threshold/length of the learning window, and SylRep = Syllable Repertoire Size.

8. Use your favorite plotting software or coding language to display the data. If you use the following code in R:

```
SylRep <- unlist(read.csv('VignetteSylRep.csv'))
plot(1:1000,SylRep, type=``l", xlab=``Time Steps",
     ylab=``Average Syllable Repertoire", font.lab=2)
```

It should produce a plot that looks like this:



2.5 How do I write my own program using the library?

In the programs folder, there is an annotated template to get you started. To get a debugging version, build it with:

```
dotnet build
```

Once you are ready to modify the template, look at the documentation for the .SEMPs and the SimParams class to know what arguments in the SimParams class to change, and look at the Simulations class to see what types of simulations are pre-built. If you need an entirely new simulation type, look at the documentation and code for Simulations.cs to figure out how to work with the data to get what you wanted. The code for function Simulations.LearningInvasion() provides a decent example of how to add more complex actions to the simulation logic.

3 Code Documentation

3.1 Simulations.cs

3.1.1 Overview

Simulations.cs contains 3 public functions, 1 public structure, and 4 private functions. It exists to allow the user to start several pre-made variants of the simulation with one line of code.

3.1.2 .Basic()

This public function runs a simulation where data is saved at every time step. The data is returned as a WriteData object. Takes 2 arguments.

1. par (SimParams; Required): Par
2. writeAll (bool; Default=true): If true, the data for every bird is saved, if false, the averages are saved

3.1.3 .Interval()

This public function runs a simulation where data is saved at time step 0 and every [frequency] time step after that. The data is returned as a WriteData object. Takes 3 arguments.

1. par (SimParams; Required): Par
2. frequency (int; Default=200): the Nth time step to save
3. writeAll (bool; Default=true): If true, the data for every bird is saved, if false, the averages are saved

3.1.4 .Invasion()

This public function runs a simulation for [burnIn] number of steps, then introduces [numInvaders] invaders. Invaders are created by reassigning the [type] of a randomly chosen male to [invaderStat] and resetting his age to 1. The simulation is continued until the invaders overtake the population, are removed from the population, or 400 time steps pass. The result is returned as an InvasionData structure, containing Steps (int), the number of time steps since the invaders were introduced, and TraitAve (float), the average [trait] at the end of the simulation. Calls .CheckStatValue(), .CreateInvader(), .CountCategories(), and .GetAverage() to ensure that the input invader stat is within the boundaries for its trait type and the correct trait type is used for other operations. Takes 5 arguments.

1. par (SimParams; Required): Par
2. type (string; Required): trait to invade. Can be Learning, Accuracy, Forget, or Invent.
3. invaderStat (float; Required): trait value for the invader

4. numInvaders (int; Default=1): the number of invaders to create
5. burnIn (int; Default=500): the number of time steps to run the simulation before introducing invaders

3.1.5 .InvasionData()

This public structure contains 1 int and 1 float, and is used to return invasion data to the main program. The constructor takes 2 arguments.

1. Steps(int; Default=default(int)): number of time steps since invaders were introduced
2. TraitAve (float; Default=default(float)): the average value of the population at the end of the simulation for the trait changed in invaders

3.1.6 .CheckStatValue()

This private functions throws an error unless a valid [type] was called and [stat] is not greater than the max or less than the min of [type], or warns the user if the inheritance for [type] is not equal to zero. It takes 3 arguments:

1. par (SimParams; Required): Par
2. type (string; Required): the invasion trait
3. stat (float; Required): the invasion stat

3.1.7 .CreateInvader()

This private function resets a male's [type] to [stat], and his age to 1. Returns a Population and takes 4 arguments:

1. pop (Population; Required): Pop
2. type (string; Required): the invasion trait
3. index (int; Required): the index of the male to modify
4. stat (float; Required): the invasion stat

3.1.8 .CountCategories()

This private function counts how many discrete values there are in the population for [type]. Returns an int and takes 2 arguments:

1. pop (Population; Required): Pop
2. type (string; Required): the invasion trait

3.1.9 .GetAverage()

This private function gets the average value for [type]. Returns a float and takes 2 arguments:

1. pop (Population; Required): Pop
2. type (string; Required): the invasion trait

3.2 SimParams.cs

3.2.1 Overview

SimParams.cs constructs a SimParams object of parameters that define how a simulation should be conducted. It contains 1 constructor, 8 private functions for generating Par and error checking it, and 6 public functions related to random sampling. The constructor uses the same arguments explained in detail in the **.SEMP section** in Pascal Case. it also takes three additionally arguments:

1. reload (bool; Default=false): whether parameters should be loaded form a .SEMP instead of specified by other arguments.
2. path (string; Default=default(string)): path to .SEMP file.
3. errorCheck (bool; Default=true): whether to perform error checking on the inputs. Unnecessary if using unmodified, .SEMPS.

There are three properties not contained within the .SEMPs, which are generated when Par is created. These are:

1. AllSyls (HashSet<int>): A numbered list of all syllables a male can learn
2. SongCore (List<float>): A list of probabilities; the chance a male or female will know a given syllable at the start of the simulation
3. Rand (Random): The Random object generated based on Par.Seed

3.2.2 .MakeCore()

This private function calculates how many syllables a male can possibly know at the start of a simulation and the probability that he will learn each one. It takes two arguments and returns a List<float>.

1. percentSyllableOverhang (float; Required): Par.PercentSyllableOverhang
2. initialSyllableRepertoire (int; Required): Par.InitialSyllableRepertoire

3.2.3 .CalculateProportion()

This private function calculates the proportion of adult birds that will survive to the next age group for the males created in time step 0. If Par.DeathStrategy=true, it calls .Polynomial() to convert a set of coefficients that relate survival rate to the population size into a function that will then be solved for the roots. See Type II Survival Curve. Takes 4 arguments and returns a float.

1. numBirds (int; Required): Par.NumBirds
2. deathThreshold (float; Required): Par.DeathThreshold
3. chickSurvival (float; Required): Par.ChickSurvival
4. maxAge (int; Required): Par.MaxAge

3.2.4 .Polynomial()

This private function takes a set of coefficients and formats them as a function. It takes 2 arguments and returns a func<double, double>.

1. coefficients (double[]; Required): the coefficients for survival in each age group
2. in_x (double; Required): length of the coefficients.

3.2.5 .TestRequirement()

This private function tests whether to set nullable bools for Save[trait] from null to true or false based on other parameters. It takes 3 arguments and returns a bool.

1. test (nullable bool; Required): a Save[trait] term
2. depedancy1 (float; Required): inheritance noise term for [trait]
3. depedancy2 (bool; false): additional possibilities that require [trait] to be saved

3.2.6 .ErrorCheck()

This private function is an organizer that calls .CheckMin(), .CheckMax(), and .CheckTrait() on appropriate variables as well as tests for other extraneous problems. Takes no arguments and returns errors and warnings is certain conditions are met.

3.2.7 .CheckMin()

This private function checks whether [trait] is set below its absolute minimum. Takes 3 arguments, and returns an error if conditions are met.

1. trait (float; Required): The value to test.
2. traitName (string; Required): Name of value to return in error message.
3. min (float; Required): the minimum value [trait] can be equal to.

3.2.8 .CheckMax()

This private function checks whether [trait] is set above its absolute maximum. Takes 3 arguments, and returns an error if conditions are met.

1. trait (float; Required): The value to test.
2. traitName (string; Required): Name of value to return in error message.
3. max (float; Required): the maximum value [trait] can be equal to.

3.2.9 .CheckTrait()

This private function checks whether the values for [name] are sensible. Takes up to 6 arguments, and returns an error if conditions are met.

1. initial (float; Required): The user-defined initial value for [name].
2. noise (float; Required): the user-defined noise for [name].
3. min (float; Required): the user-defined minimum for [name].
4. max (float; Required): the user-defined maximum for [name] .
5. name (string; Required): trait to return in error message.
6. absMax (float; Default = 1): the maximum value [name] can be set to.

3.2.10 .NextN()

This public function calls Rand.Next(N), where N is passed as an argument. Returns an int.

1. N (int; Required): the max value .Next() can return.

3.2.11 .NextFloat()

This public function converts the output from Random.NextDouble() to float. It takes no arguments and returns a float.

3.2.12 .RandomSampleEqualNoReplace()

This public function pulls [k] random numbers without replacement with equal probability. Takes 2 arguments and returns an int[] that contains the **VALUES** of [list].

1. list (<int>; Required): list of values.
2. k (int <= list.Count; Required): the number of items to sample

3.2.13 .RandomSampleEqualReplace()

This public function pulls [k] random numbers with replacement with equal probability. Takes two arguments and returns an int[] that contains the **INDICIES** of [list].

1. list (<int>; Required): list of values.
2. k (int; Required): the number of items to sample

3.2.14 .RandomSampleUnequal()

This public function pulls [k] random numbers with or without replacement with unequal probability. Implements the algorithm described in [6]. Takes 3 arguments and returns an int[] that contains the **INDICIES** of [list].

1. probs (float[], Required): the probabilities for each item to sample
2. n (int; Required): the number of items to sample
3. withReplacement (bool; false): Whether items are sampled with replacement

3.2.15 .BackPropValue()

This private function is used to give appropriate probability values to trees generated in .randomSampleUnequal(). It takes 3 arguments and modifies [tree].

1. tree (float[]; Required): tree to modify
2. j (int; Required): position in tree
3. v (float; Required): probability

3.3 Population.cs

3.3.1 Overview

Population.cs exists to generate an object of class Population, which is the bird population at time step 0. It contains 1 constructor, 4 public functions and 2 private functions The population class contains properties for song-learning traits and songs. These are:

1. Age (int[]): Age of the male. Generated by Ages.cs.
2. SyllableRepertoire (int[]): Stores the total number of syllables each male knows
3. Local (List<List<int>>): Stores which territories are local and local+N steps away from a target territory. Generated by Locations.cs if Par.LocalBreed or Par.LocalTutor is true.
4. Name (string[]): GUIDs for birds. Generated if Par.SaveNames=true
5. FatherName (string[]): GUIDs inherited from the father. Set to NA for timestep 0. Generated if Par.SaveNames=true
6. Accuracy (float[]): How well a male can learn syllables. Generated using .InitialDistributions().
7. LearningThreshold (float[]): Length of time that a male can continue to learn. Generated using .InitialDistributions().
8. ChanceInvent (float[]): Male's chance to invent a new syllable if learning a tutor syllable fails. Generated using .InitialDistributions().
9. ChanceForget (float[]): Male's chance to forget syllables not sung by a tutor. Generated using .InitialDistributions().
10. Match (float[]): How similar a male's song repertoire is to his female's song template. Generated by Song.cs only if Par.SaveMatch=true.
11. MaleSong (List<int>): Complete syllable repertoires of males. Generated by Song.cs. Unlike the R version, syllables are stored as ID numbers in [0, Par.MaxSyllableRepertoire-1], and not as an array of 1s and 0s.
12. FemaleSong (List <int>): Syllable/Song templates known by females. Generated by Song.cs only if Par.SaveMatch=true. Unlike the R version, syllables are stored as ID numbers in [0,Par.MaxSyllableRepertoire-1], and not as an array of 1s and 0s.
13. SurvivalChance (List<float>): The survival probabilities for all generations of birds.
14. SurvivalStore (float): The adult survival probability for the youngest generation (age=0). This stores the survival value for one time step prior to loading into SurvivalChance, as chicks survive based on Par.ChickSurvival.

3.3.2 .InitialDistributions()

This private function sets values for [trait] for all birds at time step 0. If Par.Inherited[Trait]Noise=0, each male is assigned Par.Initial[Trait]. Otherwise, the function creates a beta distribution with mode=Par.Initial[Trait] and Min and Max either based on the mode +/− Par.Inherited[Trait]Noise or the absolute max and/or min for [Trait] if the mode+/−noise is too large (max) or small (min). Traits for individuals birds are pulled from this distribution. Takes 5 arguments and returns a float[].

1. numBirds (int; Required): Par.NumBirds
2. noise (float; Required): Par.Inherited[Trait]Noise
3. initial (float; Required): Par.Initial[Trait]
4. max (float; Required): Par.Max[Trait]
5. min (float; Required): Par.Min[Trait]

3.3.3 .ReplaceBird()

This public function overwrites the values for a dead bird with the values for a chick generated based on inheritance from a father. Calls .Inheritance() for traits inherited with noise. If VerticalLearning is true, .VerticalLearning() is called. Otherwise, chicks are assigned a blank song with no syllables. Takes 3 arguments and modifies the instance of Population that calls it.

1. par (SimParams; Required): Par
2. father (int; Required): the index of the father
3. territory (int; Required): the index of a dead male/the soon-to-be-generated chick

3.3.4 .Inheritance()

This private function creates a distribution from which to pull a chick's [trait] based on the father's [trait]. The logic is the same as in .InitialDistributions(), except that the father's value is used as the mode. Takes 5 arguments and returns a float.

1. min (float; Required): Par.Min[Trait]
2. max (float; Required): Par.Max[Trait]
3. initial (float; Required): Par.Initial[Trait]
4. noise (float; Required): Par.Inherited[Trait]Noise
5. rand (Random; Required): Par.Rand

3.3.5 .Print()

This public function exists for diagnostic purposes. It writes all the traits for a given territory to the console. Takes 1 argument and writes to the console.

1. birb (int; Required): the territory of the bird/birb you want to print data from.

3.3.6 .SongFragments()

This public function exists for diagnostic purposes. It writes all syllables known by each female or male bird in the population to the console. Takes 1 argument and writes to the console.

1. sex (string [“Male”, “Female”]; “Male”): Whether to print male or female repertoires.

3.3.7 .UniqueSyllables()

This public function exists for diagnostic purposes. It separately writes the unique syllables known by males and females to the console. Takes 0 arguments and writes to the console.

3.4 Ages.cs

3.4.1 Overview

Ages.cs contains 1 public function that is supported by 3 private functions. It exists to create the initial age distribution for the population, using one of two methods. If a Par.AgeDeath = false, then an approximately equal number of males will be in each age group at the start of the simulation. Otherwise, the three private functions are used to calculate how many males should be in each age class to satisfy the conditions of a type II survival curve.

3.4.2 .InitialAgeDistribution()

This public function tests which death strategy is being implemented and either 1) randomly assigns a uniform distribution of ages from 0 to Par.MaxAge or 2) calculates how many males should be in each age group to fit the assumptions a type II survival curve. Returns an array of ages. Takes 1 argument.

1. par (SimParams; Required): Par

3.4.3 .GetAgeRates()

This private function first calls .CalculateAllGenerations() to get the number of birds that would be in each age group, where Par.DeathThreshold number of birds are considered dead. These numbers are converted into fractions, which are then corrected, so that all birds are considered alive. These fractions are returned. Takes 1 argument.

1. par (SimParams; Required): Par

3.4.4 .CalculateAllGenerations()

This private function calculates and returns the number of birds that should be in each age group given the chick survival rate (Par.ChickSurvival), the max age (Par.MaxAge), the proportion of adults that survive to the next age (Par.InitialSurvival), and the death threshold (par.DeathThreshold). This ultimatey calculates how many birds are in each age group when assuming a number of birds equal to the death threshold are dead. Takes 1 argument.

1. par (SimParams; Required): Par

3.4.5 .GetAgeGroup()

This private function assigns the number of birds that must be in each age group based on the values in ageRates (the integer portion of the AgeRates*Par.NumBirds). The remaining unassigned birds are randomly placed into an age group with a probability based on ageRates. Returns the age of each bird in the population. Takes 2 arguments.

1. par (SimParams; Required): Par
2. ageRates (float[]; Required): the output from .GetAgeRates()

3.5 Locations.cs

3.5.1 Overview

Locations.cs contains 3 public functions and 3 private functions. These functions are split into performing one of two tasks: 1) establishing a data structure that defines how near territories are to one another (which territories are "local" and which are further away) and 2) using location data to pick males that are local to a target territory.

3.5.2 .FinalDirections()

This public function is an organizer that calls .FirstStepDirections() to calculate which territories are within one step of one another. If Par.Steps >1, it next calls .NextStepDirections() to increase the range of local territories from 1 to Par.Steps. Next, .NextStepDirections() is called again to calculate which territories are Steps+[1,max(Par.Rows, Par.Cols)] to store which birds the next nearest. This creates the data structure for Pop.Local. Takes 1 argument and returns a List<List<int>>.

1. par (SimParams; Required): Par

3.5.3 .FirstStepDirections()

This private function calculates which territories are within one square of other territories based on the dimensions of the bird matrix (**See Conceptualizing the Bird Matrix**). Takes 1 argument and returns a List<List<int>>.

1. par (SimParams; Required): Par

3.5.4 .NextStepDirections()

This private function takes the current location structure (N steps) and adds to it the territories one step further away (N+1 steps). Takes 2 arguments and returns a List<List<int>>.

1. currentStep (List<List<int>>; Required): the data structure containing the current number of steps traversed (N=[1, max(Par.Rows, Par.Cols)-1]).
2. firstStep (List<List<int>>; Required): the output from .FirstStepDirections().

3.5.5 .GetLocalBirds ()

This public function calls .LocalSearch() to get the nearest birds that are available (e.g. not dead). It then randomly samples numBirds from that group. Takes up to 6 arguments and returns an int[].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. target (int; Required): the territory around which local birds should be found

4. unavailable (HashSet<int>; Required): birds that cannot be chosen

5. numBirds (int, Default=1): the number of birds to sample

6. probs (float[]; Default=null): the probability of picking each bird if the probabilities are not equal

3.5.6 .LocalSearch()

This private function tests whether enough local birds are available, and if not, extends the range of local by 1 until there are enough local birds. Takes up to 5 arguments and returns a List<int>.

1. par (SimParams; Required): Par

2. pop (Population; Required): Pop

3. territory (int; Required): the territory around which local birds should be found

4. unavailable (HashSet<int>; Required): birds that cannot be chosen

5. needed (int, Default=1): the number of birds to sample

3.5.7 .GetGlobalBirds()

This public function samples [numBirds] tutors from the available, global birds. It takes up to 6 arguments and returns a List<int>.

1. par (SimParams; Required): Par

2. pop (Population; Required): Pop

3. learner (int; Required): the learner

4. unavailable (HashSet<int>; Required): b that cannot be chosen

5. numBirds (int, Default=1): the number of birds to sample

6. probs (float[]; Default=null): the probability of picking each bird if the probabilities are not equal

3.6 Songs.cs

3.6.1 Overview

Songs.cs contains 1 constructor, 2 public functions, 5 private functions, and 1 private structure. Most of the functions are used at time step zero to generate a Song class object. Based on Par, the Song constructor generates male songs and possibly female songs if Par.SaveMatch=true. If female songs are generated, male songs may be generated to match dialects in the female songs depending on the value of Par.MaleDialects. If Par.ChooseMate=true, females are given the chance to pick mates based on their song, rather than being randomly assigned. Afterwards, the matches between male and female songs is calculated. The class contains three properties.

1. FemaleSongs (List<int>[])
2. MaleSongs (List<int>[])
3. Match (float[])

3.6.2 .GenerateFemaleSongs()

The private function calls .GenerateNovelSong() to create identical female songs if Par.MatchUniform=true, or noisy female songs if Par.MatchUniform=false. If Par.Dialects>1, it calls .EstablishDialects(). Takes 1 argument and returns a List<int>[].

1. par (SimParams; Required): Par

3.6.3 .GenerateNovelSongs()

This private function tests whether a bird learns each syllable in Par.SongCore by generating a random float [0,1). If the float is less than the value for a syllable in Par.SongCore, that syllable is learned. Takes 1 argument and returns a List<int>.

1. par (SimParams; Required): Par

3.6.4 .EstablishDialects()

This private function creates regional dialects in territory space that are of equal size and are as square as possible given the number of desired dialects and the dimensions of the bird matrix. It chooses the dimensions of a dialect region using the prime factorization created via .PrimeFactorization(). Once chosen, the code calculates which territories fall into each dialect region, and shifts the syllable repertoire to the right; the number ID of known syllables are increased by Par.SongCore.length multiplied by the number of the dialect region -1. Thus, given three dialects with Par.SongCore of length 10, dialect region 1 would be unchanged (IDs [0,9]), while the dialect region 2 syllable IDs are increased by 10 (IDs [10,19]), and dialect region 3 syllables are increased by 20 (IDs [20,29]). Takes 2 arguments and returns a List<int>[].

1. par (SimParams; Required): Par
2. fSongs (List<int>[]; Required): female songs

3.6.5 .PrimeFactorization()

This private function calculates the prime factorization of birds. Takes 1 argument and returns a List<int>.

1. birds (int; Required): Par.NumBirds

3.6.6 .GetMatch()

This public function tests for the similarity between the repertoire of a male and the template of a female. The equation to test the match is:

$$Extra = max(NumMaleSyllables - NumFemaleSyllables, 0) \tag{14a}$$

$$Missing = NumFemaleSyllables - NumSylsShared \tag{14b}$$

$$Match = max(1 - \frac{Extra + Missing}{NumFemaleSyllables}, 0) \tag{14c}$$

Takes 3 arguments and returns a float.

1. par (SimParams; Required): Par
2. maleSong (List<int>; Required): MaleSong for comparison
3. femaleSong (List<int>; Required): FemaleSong for comparison

3.6.7 .ChooseMates()

This public function calls .AssignFemale() to allow females to pick mates based on their song, and then reorganizes the original Population data based on the output from that function. Takes 2 arguments and modifies [pop].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop

3.6.8 .AssignFemale()

This private function assumes that males start unmated and available, scrambles the order of females, and then serially compares the match between one female and each male. The female then randomly chooses a male with probability based on the match to her song. Once paired, males are removed from the available pool. This process continues until all birds are paired. Takes 3 arguments and returns AssignResults.

1. par (SimParams; Required): Par
2. maleSong (List<int>; Required): Pop.MaleSong
3. femaleSong (List<int>; Required): Pop.FemaleSong

3.6.9 .AssignResults()

This private structure contains three arrays, and is used to return the data from .AssignFemales() to .Songs(). The constructor takes 3 arguments.

1. Match (float[]; Required): The matches between the reordered males and females
2. maleOrder (int[]; Required): The new order for males
3. femaleOrder (int[]; Required): The new order for females

3.7 BirthDeathCycle.cs

3.7.1 Overview

BirthDeathCycle.cs exists to modify a Population class to complete a simulation time step. It contains 1 public function and 7 private functions.

3.7.2 .Step()

This public function calls .AgeDeath() or .RandomDeath() to obtain a list of birds that have died. Remaining birds are given the opportunity to learn by calling Learning.ObliqueLearning() if Par.ObliqueLearning = true. These birds then have their age increased by 1. Fathers are chosen using ChooseMaleFathers(), and then chicks are generated using the Population.ReplaceBird(). If Par.OverLearn = true, then Learning.OverLearn() is called. If Par.FemaleEvolution = true, .ChooseFemaleFathers() is called and female song templates are modified to be identical to the father's song. If Par.ChooseMate = true, Songs.ChooseMates is called. Finally, if Par.AgeDeath = true, .UpdateDeathProbabilities() is called. This function takes 2 arguments and modifies [pop].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop

3.7.3 .AgeDeath()

This private function calls .LearningPenalty() to get the learning age threshold penalty for all birds. It then calculates how many males are in each age group ([0,Par.MaxAge]), and multiples this by the survival chance for that age group to calculate how many males must die in each age group. That many males are randomly sampled from the age group with probability based on the LearningPenalty. The indices of the chosen males are returned. Takes 2 arguments and returns an int[].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop

3.7.4 .RandomDeath()

This private function calls .LearningPenalty() to get the learning age threshold penalty for all birds. It then calculates how many males must die by multiplying Par.NumBirds by Par.PercentDeath. That many males are randomly sampled with probability based on the LearningPenalty. The indices of the chosen males are returned. Note that in these simulations, nothing is done to cull birds that are older than MaxAge. Takes 2 arguments and returns an int[].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop

3.7.5 .LearningPenalty()

This private function calculates the survival penalty (the chance of being chosen to die) for males. The calculation considers Par.MaxAge, so that males that learn until death are penalized more heavily than those that only learn a few years. The equation is:

$$\frac{Par.LearningPenalty}{(Par.MaxAge - 1)} * (LearningThreshold_{male} - 1) + 1 \tag{15}$$

Any males with a survival penalty <1 are reset to 1. Thus, only males with learning age thresholds greater than 1 are penalized. Takes 2 arguments and returns a float[].

1. par (SimParams; Required): Par
2. learnThreshold (Population; Required): Pop.LearningThreshold

3.7.6 .ChooseMaleFathers()

This private function picks fathers for male chicks from the pool of available birds. It calls .ReproductiveProbability() to get the chance that each male is chosen as a father. If Par.LocalBreed=true, only local males can be selected to father a chick in a target vacant territory (any territory where a bird died). Otherwise, males are randomly sampled globally. In both cases, one male can father multiple chicks. Takes 4 arguments and returns an int[].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. Vacant (int[]; Required): dead males
4. notVacant (List<int>; Required): males that are still alive

3.7.7 .ChooseFemaleFathers()

This private function picks fathers for female chicks from the pool of available birds. It calls .ReproductiveProbability() to get the chance that each male is chosen as a father. If Par.LocalBreed=true, only local males can be selected to father a chick in a target vacant territory (any territory where a bird died). Otherwise, males are randomly sampled globally. In both cases, one male can father multiple chicks. There is an additional check that prevents a female from having the same father as her prospective mate. Takes 4 arguments and returns an int[].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. Vacant (int[]; Required): dead males
4. notVacant (List<int>; Required): males that are still alive
5. maleFathers (int[]; Required)

3.7.8 .ReproductiveProbability()

This private function calculates a male's probability of being chosen as a mate based on female preferences and his song characteristics. Preferences are broken into three groups of bonuses: repertoire size preference, match preference, and noise preference. All available males are automatically given the noise preference bonus (Par.NoisePreference). Repertoire size preference is calculated by giving the male(s) with the largest repertoire the full value of Par.RepertoirePreference, and then applying a prorated bonus to other males. The male(s) with the smallest repertoire gets none of the bonus. The match bonus is applied by multiplying a male's match to his female by Par.MatchPreference. Because the three preferences must add to 1, the maximum bonus a male can earn is 1. Takes 3 arguments and returns a float[].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. UsableMales (int[]; Required): living males with songs

3.7.9 .UpdateDeathProbabilities()

This private function updates Pop.SurvivalChance and Pop.SurvivalStore to reflect the addition of newest generation of chicks and the loss of the oldest generation. Takes 3 arguments and modifies Pop.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. fatherInd (int[]; Required): males chosen to father male chicks

3.8 Learning.cs

3.8.1 Overview

Learning.cs compiles all the functions related to vertical and oblique learning. It contains 4 public functions, 12 private functions, and 1 private structure.

3.8.2 .CoreLearningProcess()

This public function allows a learner to add syllables from a tutor to his repertoire. A learner adds a syllable if a random float([0,1)) is less than the learner's learning accuracy. If a learner fails to learn a syllable, he then has a chance to invent a syllable equal to his chance to invent. If this occurs, a learner generates a syllable that is not in his repertoire or the portion of the tutor's repertoire that was heard by the learner at a random position in the syllable space. Takes 5 arguments and returns a List<int>.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. learner (int; Required): the index of the learner
4. learnerSong (List<int>; Required): the learner's current song
5. tutorSong (List<int>; Required): the set of syllables the learner can add to his repertoire

3.8.3 .VerticalLearning()

This public function creates a blank song for a new chick and calls .CoreLearningProcess() on this and the father's song. Takes 4 arguments and returns a List<int>.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. fatherInd (int; Required): the index of the father
4. chickInd (int; Required): the index of the chick repertoire

3.8.4 .ObliqueLearning()

This private function is a wrapper for the functions related to oblique learning. It calls .GetLearners() to see which birds in the population are eligible to learn. If Par.Consensus = true, it next calls .ChooseMultipleTutors() and ConsensusLearning(). Otherwise, it calls .ChooseTutors() and ListeningTest(). In either case, if Par.Add = true, .AddSyllables() is called and if Par.Forget=true, .ForgetSyllables() is called. Finally, .UpdateSongTraits() is called. Takes 4 arguments and modifies [pop].

1. par (SimParams; Required): Par

2. pop (Population; Required): Pop
3. vacant (int[]; Required): dead males
4. notVacant (List<int>; Required): living males repertoire

3.8.5 .ConsensusLearning()

This private function calls .ListeningTest() to get the syllables that a males hears from each of his tutors and then assembles a consensus song from them. The consensus song is passed to .ConsensusCalc() to calculate the chance a learn will attempt to learn a a given syllable. If a random float ([0,1)) is less than that value, it is added to the list of syllables a male will attempt to learn. Takes 4 arguments and returns a ConsensusResults structure.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. tutors (List<int>[]; Required): the tutors
4. learners (List<int>; Required): the learners

3.8.6 .ConsensusCalc()

This private function calculates the chance that a male will attempt to learn a syllable based on what Par.ConsensusStrategy is set to and returns a float. Where X is the percentage of tutors that sang a syllable:

Conform:

$$X - \frac{sin(2\pi * X)}{(2\pi)} \tag{16}$$

AllNone:

$$floor(X) \tag{17}$$

Percentage:

$$X \tag{18}$$

Takes 3 arguments.

1. par (SimParams; Required): Par
2. syl (int; Required): a syllable from the consensus song
3. collapsedSongs (List<int>; Required): list of all the syllables that a learner heard from all his tutors, includes duplicates

3.8.7 .ConsensusResults()

This private structure exists to return the results from .ConsensusLearning() to .Oblique-Learning(). Its constructor takes 2 arguments.

1. addSyls (List<int>[]): the syllables each learn will attempt to learn
2. ConsensusSong (List<int>[]): all of the syllables the learner heard

3.8.8 .GetLearners()

This private function calls .TestLearningThreshold() to get birds capable of learning, and then calls .CheckEncounter() on them, to test whether they met a tutor. If no males were capable of learning, a warning is triggered. Takes 3 arguments and returns a List<int>.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. notVacant (List<int>; Required): the males that are alive

3.8.9 .TestLearningThreshold()

This private function tests whether males are younger than their learning age threshold -1. If so, they are guaranteed to learn. Otherwise, if they are younger than their learning age threshold, then they have a learning age threshold - current age chance to learn. If a random float ([0,1)) Is less than this value, the male will learn. Takes 3 arguments and returns a List<int>.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. notVacant (List<int>; Required): the males that are alive

3.8.10 .CheckEncounter()

This private function tests whether a random float ([0,1)) is less than Par.EncouterSuccess for every bird to test whether that bird meets tutors. Takes 2 arguments and returns a List<int>.

1. par (SimParams; Required): Par
2. capable (List<int>; Reuired): birds capable of learning

3.8.11 .AddSyllables()

This private function checks whether any syllables that a learner will attempt to learn are already in his repertoire. If so, they are removed. It next calls .CoreLearningProcess(). Takes 4 arguments and modifies [pop].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. learner (int; Required): the index of the learner
4. tutorSyls (List<int>; Required): the set of syllables the learner can add to his repertoire

3.8.12 .ForgetSyllables()

This private function checks whether any syllables in the learner's repertoire were not sung by the tutor(s). If so, they can be forgotten by calling .DropSyllables(). Takes 4 arguments and modifies [pop]. Note that for non-consensus learning, argument 4 would be identical to the list of syllables that the would male attempt to learn in .AddSyllables() if it were called. For consensus learning, argument 4 is every syllable a male heard (the entire consensus song), and this may be different from the list of syllables that the male would attempt to learn in .AddSyllables() if it were called.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. learner (int; Required): the index of the learner
4. tutorSyls (List<int>; Required): the set of syllables the learner heard

3.8.13 .UpdateSongTraits()

This private function updates Pop.SyllableRepertoire and Syllable.Match (if Par.SaveMatch = true) after learning is completed. Takes 3 arguments and modifies [pop].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. learners (List<int>; Required): the set of syllables the learner heard

3.8.14 .ChooseTutors()

This private function picks 1 tutor for a learner from the list of available tutors. Available tutors must be alive, older than age 0, know at least one sylable, and cannot be the target learner. If Par.LocalTutor = true, Locations.GetLocalBirds() is called. Otherwise, a tutor is randomly selected globally. All birds that fit the available criteria have an equal chance of being chosen. Takes 4 arguments and returns an int[].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. learners (List<int>; Required): the males that need tutors
4. notVacant (List<int>; Required): the males that are alive

3.8.15 .ChooseMultipleTutors()

This private function picks numTutor tutors for a learner from the list of available tutors. Available tutors must be alive, older than age 0, know at least one sylable, and cannot be the target learner. If Par.LocalTutor = true, Locations.GetLocalBirds() is called. Otherwise, a tutor is randomly selected globally. All birds that fit the available criteria have an equal chance of being chosen. Takes 5 arguments and returns a List<int>[].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. learners (List<int>; Required): the males that need tutors
4. notVacant (List<int>; Required): the males that are alive
5. numTutors (int; Required): the number of tutors each learner requires

3.8.16 .ListeningTest()

This private function calculates which syllables that a learner hears from a tutor. A learner will either hear all or a portion of a tutor's repertoire, depending on the value for Par.ListeningThreshold (oblique) or Par.FatherListeningThreshold (vertical). If only a portion of the repertoire is heard, the correct number of syllables are randomly chosen from the tutor's repertoire. The amount heard is decided but one of three methods: 1) if the listening threshold either 0.999 or equal to the maximum repertoire size, the entire repertoire is heard. If it is any other integer, than that is number of random syllables a learner hears form a tutor, unless the tutor's repertoire is smaller than that value. If it is any other floating point number, a learner hears:

$$MinLearnSylls + (TutorReprtoiresize - MinLearnSylls) * ListeningThreshold \quad (19)$$

The resulting value is rounded up to the next integer if the fractional remainder is greater than a randomly drawn number. If not, it is rounded down. Takes 4 arguments and returns a List<int>[].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. tutors (int[]; Required): chosen tutors
4. lisThresh (float; Required): listening threshold

3.8.17 .DropSyllables()

This private function tests whether a learner forgets a syllable he did not hear from his tutor(s). This occurs if a randomly drawn float ([0,1)) is less than the learner's chance to forget. Takes 4 arguments and modifies [song].

1. par (SimParams; Required): Par
2. song (List<int>; Required): the learner's current song
3. droppableSyls (List<int>; Required): syllables a learner may forget
4. chanceForget (float; Required): the learner's chance to forget

3.8.18 .OverLearn()

This public function allows chicks to learn their song by listening to many tutors. Par.OverlearnNoTutors tutors are chosen by calling .ChooseMultipleTutors(). .ListeningTest() is called to test which syllables a chick hears. .AddSyllables() and .UpdateSongTraits() are them called to allow the chicks to learn. Takes 4 arguments and modifies [pop].

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. vacant (int[]; Required): chicks
4. notVacant (List<int>; Required): living males

3.9 WriteData.cs

3.9.1 Overview

WriteData.cs exists to create a class that organizes the data for saving to .csv files. It contains 3 public and 3 private functions. Its properties match the names from the Population class, however they are StringBuilders.

3.9.2 .Write()

This public function is a wrapper that calls either .WriteAll() or .WriteAve() to update the WriteData object. It takes up to 3 arguments and modifies the current instance of WriteData.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop
3. writeAll (bool; Default=true): whether to write the data from each individual bird (true), or the population averages (false).

3.9.3 .WriteAll()

This private function converts the array data from each bird of each trait where Save[trait] = true to a comma separated string and appends it to the appropriate StringBuilder. Takes 2 arguments and modifies the current instance of WriteData.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop

3.9.4 .WriteAve()

This private function averages the array of each trait where Save[trait] = true and adds it to the appropriate StringBuilder. Takes 2 arguments and modifies the current instance of WriteData.

1. par (SimParams; Required): Par
2. pop (Population; Required): Pop

3.9.5 .Output()

This public function saves the current instance of WriteData as .csvs and may save Par as a .SEMP. Takes up to 4 arguments.

1. par (SimParams; Required): Par
2. filePath (string; Required): path to save location

3. tag (string; Required): the name to add to the files. Note that all files have their trait name by default, and tag is added before the trait name.

4. writePar (bool; Default=true): whether to write Par to a .SEMP

3.9.6 .WriteParams()

This private function saves SimParam instances as .SEMP files. It takes 3 arguments.

1. par (SimParams; Requierd): Par

2. filePath (string; Required): path to save location

3. tag (string; Required): the name to add before the .SEMP extension

3.9.7 .ConCat()

This public function allows for the concatenation of a new WriteData instance to the current instance. Please note that there is no error handling in this function, and concatenating unlike WriteData instances can have unintended consequences. Takes 2 arguments and modifies the current instance of WriteData.

1. new (WriteData; Required): data to concatenate onto current instance of WriteData.

2. par (SimParams; Requierd): Par

3.10 Utils.cs

3.10.1 Overview

Utils.cs only contains two public functions, which are used in programs rather than in the simulations themselves. These functions exist to simplify file processing.

3.10.2 .GetValidParams()

This public function takes all of the filenames in a folder and returns only those that are .SEMPS. Takes 1 argument.

1. path(Required): Path to a folder containing .SEMPs

3.10.3 .GetTag()

This public function pulls [name] from [name].SEMP for use in creating the file names of outputs created from name.SEMP. Takes 1 argument.

1. filename(Required)-The file path of [name].SEMP

4 Acknowledgements

Ray Yu wrote the seed code for the first version of the model in MATLAB. Jordan Singer helped develop the strategy and logic for implementing the type II survival curve, and Kevin Walsh wrote the C# code for the functions .RandomSampleUnequal(), .BackPropValue(), and .Polynomial(). We used the `BetaScaled.PERT()` function from MathNet.Numerics v4.5.1 (`https://numerics.mathdotnet.com/`).

5 References

[1] Naef-Daenze, B., Widmer, F., and Nube, M. (2001) .Differential post-fledging survival of great and coal tits in relation to their condition and fledging date. *Journal of Animal Ecology 70:*(5), 730-738. DOI: 10.1046/j.0021-8790.2001.00533.x

[2] Loison, A., Saether, B., Jerstad, K., and Rostad, O.W.. (2010). Disentangling the sources of variation in the survival of the European dipper. *Journal of Applied Statistics 1:*(4), 289-304. DOI: 10.1080/02664760120108665

[3] Stenzel, L.E., Page, G.W., Warriner, J.C., Warriner, J.S., George, D.E., Eyster, C.R., Ramer, B.A., and Neuman, K.K. (2007). PRBO Conservation Science, 3820 Cypress Drive Suite 11, Petaluma, California 94954, USA Kristina K. NeumanSurvival and natal dispersal of juvenile snowy plovers (*charadrius alexandrines*) in central coastal California. *The Auk 124:*(3), 1023-1036. DOI: 10.1642/0004-8038(2007)124[1023:SANDOJ]2.0.CO;2

[4] Tyler, G.A., and Green, R.E. (2003). Effects of weather on the survival and growth of corncrake *Crex crex* chicks. *Ibis 146:*(1), 69-76. DOI: 10.1111/j.1474-919X.2004.00225.x

[5] Begon, M., Harper, J.L., and Townsend, C.R. (1996). *Ecology: Individuals, populations and communities*, 3rd ed. Oxford:Blackwell Science Ltd.

[6] Wong, C.K. and Easton, M.C. (1980). An efficient method for weighted sampling without replacement. *SIAM Journal on Computing, 9:*(1), 111-113. DOI: 10.1137/0209009