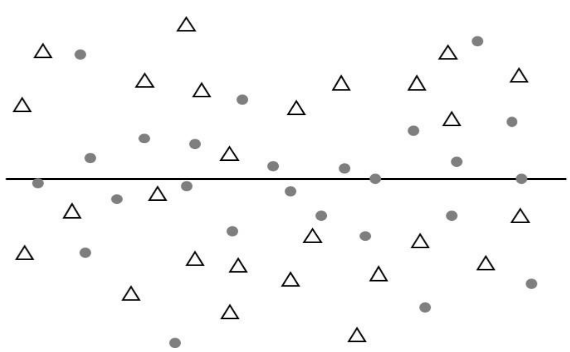
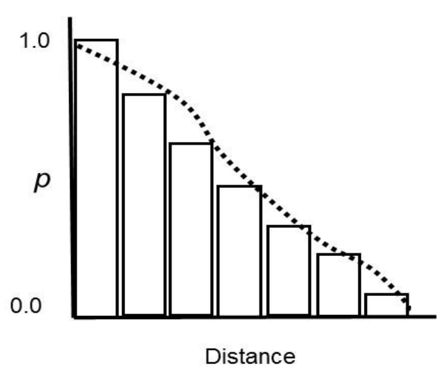
Lab 9 – Introduction to Program R and Distance Sampling for the Zombie Apocalypse.

Lab overview: This lab is designed to use the principles of distance sampling to evaluate the detection rate and density of zombies in a not-too-distant future post-apocalyptic scenario. The data you will be working with today were collected by survivors of the initial zombie virus outbreak, and consist of occurrence of zombies along two 1000 meter transect lines in a wooded landscape, not unlike that surrounding the University of Maine. Today we will analyze those data and derive estimates of zombie abundance and detection rates along the transect lines.

Recall that distance sampling is a technique for estimating animal (or zombie) abundance/density measured along transect lines or from fixed points. Observation of animals along the line (or at a point) are recorded along with a measure of their perpendicular distance from the line. We assume that animals on the line are detected perfectly (p=1.0), and that our ability to detect animals decreases as the animals are located further from the line. Based on this detection function we can correct for imperfect detection of individuals (right figure below), and estimate the proportion that were present but missed (Left figure – circles are detected animals, triangles are missed animals). Distance sampling is commonly used to estimate animal density (abundance/unit area) at different geographic locations or as a function of different habitat types.





First up – Introduction to R: For today's lab I'm also going to introduce you to a new software for this course and possibly new to you in general, program R. R is a commonly used and exceptionally powerful software for completing an almost limitless suite of tasks. It can be used for statistical analysis, data management, graphing, GIS, writing, and analyzing demographic data, among many other possibilities. R is an open source software that can be downloaded and used by anyone free of charge (https://www.r-project.org/). Learning to use R could be the subject of an entire semester (or multiple semesters) and today we will barely even scratch the surface. But, I hope you leave this lab with at least a passing familiarity with how R works, and that will require a short bit of R background to start.

R is a command line based program, where you enter prompts in the form of written command statements in order to perform operations, rather than the point-and-click operation of many other software. The benefit to this is that you can complete just about any operation you can dream of, rather than being confined by the pre-set options that the software's developers designed when they wrote the program. The downside is that correctly coding programs in R requires that you understand the programming language and enter the appropriate command correctly in order to perform the operation you would like to accomplish. This is basically equivalent to learning a new spoken language, and so there is a very large learning curve inherent to working effectively in R. Fortunately, there is a huge volume of online resources available for learning the language and troubleshooting your code; pretty much any problem you run into can be answered with some clever Googling. Today I want to work through a few basic operations just to introduce you to some concepts that will be needed for analyzing the distance data.

1. We will work in R via a secondary program, RStudio, which will to organize your code and interface with the R software. RStudio is also free software (https://www.rstudio.com/), and it works in conjunction with R (i.e. you need to have them both installed on your computer to use RStudio). For starters, open RStudio, then go to File->New File->RScript. You should be left with and open dialog screen in the upper left corner that will store and save any commands you type. Below that is a second window labeled "Console". This is the actual R command window. You can also type commands directly here, but they will not be saved in your new script file. In the script window (upper left) type the following:

   Name<- "My name is John"

   Feel free to include your name or just pretend to be John. In the upper right corner, you'll see a green arrow next to "Run". Press this, and you should see it automatically pasted into the console below. Now click into the console (lower) window and type the word Name. It should return [1] "My name is John". Programming in R is referred to as "object-oriented, where you create objects and give those objects "attributes". You have just created an object, called Name, which contains one attribute, the statement "My name is John". A few important things to note here are that the "<-" command is the generic function R uses to define an object. Also note that R is case sensitive. Try typing name with a lower case N into the command window. You should get red warning message alerting you to the fact that the object ‘name’ doesn’t exist, because the n is not capitalized. Case sensitivity is very important and will be the source of much frustration for you if you don’t pay careful attention to your coding.

2. Next let’s create an object that contains a list. Type the following into your script:

```
snacks<- as.list(c(‘chips’, 'carrots', 'apples', 'candy'))
```

The as.list() command tells R you are making a list, and the c() command combines the four words into that list. View the object in the console window, and you should see it contains a list of the 4 different values. Now we’ll create a second object as a list that defines whether these snacks are considered healthy.

```
healthy<- as.list(c(‘N’,’Y’,’Y’,’N’))
```

If we want to combine these into a single object with multiple attributes, we can use the following:

```
food<- as.data.frame(cbind(snacks, healthy))
```

Take a look at this object. The cbind() operator is a function in R that combines the lists as though they were each columns (i.e. binds by columns, c), and as.data.frame() defines the object as a data frame. Alternatively we can combine the lists as rows using the command rbind()

```
food<- as.data.frame(rbind(snacks, healthy))
```

Now take a look at food again and see how it has changed. Notice that it has changed the orientation of information contained in the table. It’s very important to recognize that R will overwrite your object without question if you tell it to. Switch it back to the original column-bound version:

```
food<- as.data.frame(cbind(snacks, healthy))
```

R uses an ordering notation, which is very similar to what we will learn in lecture regarding a position in a matrix. This is indicated by the [1,] and [,1] labels. For example, a position of [1,2] would return the second attribute in the first row of the data frame. You can ask R to return specific pieces of information based on their position. For example:

```
food[3,2]
```

Will give us the value of “Y”, which is contained in the 2nd position of the 3rd row. You could also return an entire row of data:

```
food[4,]
```

Finally, we can use column headings to identify specific attributes within our data frame by combining the object name with the column label as follows.

```
food$healthy
```

3. We can also use R to work with numeric values. For example, type or paste the following code into your script and run it:

```
mass<- as.numeric(c(100, 120, 105, 101, 99, 87, 102, 92))

temp<- as.numeric(c(12, 13, 11.5, 11, 10, 9, 12, 8))

mass.temp<- as.data.frame(cbind(mass, temp))

mass.temp
```

4. We can use R to compute basic statistics, for example a simple linear regression of our mass and temperature data using the lm() command for a linear model:

```
reg1<- lm(mass~temp)

summary(reg1)
```

The summary command is a generic command that summarizes the attributes of an object. In this case the regression model “reg1” is an object that contains the statistical outputs of the regression of mass and temp. You can use summary() on any object though; try it for your data frame mass.temp.

5. R has a large number of nice graphing features. For example, use the plot() command to produce a scatter plot with temperature on the x-axis and mass on the y-axis:

```
plot(temp, mass)
```

and the abline() feature will fit our regression line based on the coefficients from the regression output:

```
abline(reg1)
```

6. R can also be used to complete mathematical operations and these can be integrated into working with data frames. For example, lets add a new attribute column to mass.temp that converts mass (currently in grams) to kilograms:

```
mass.temp$kg<- mass/1000

mass.temp
```

7. A few final things about R before we start our distance exercise. **READ THE FOLLOWING BUT DON'T TYPE IN ANY CODE!!** R contains many base commands that are always active in the program, such as the lm() command for a linear regression. However, R users can develop specialized commands by writing their own "packages" that perform specific functions. Sometimes it's necessary to install and load a new package into R to accomplish a specific operation. This is done using the install.packages() command, and then loading the package as library(). Today we will be using the package "unmarked" (https://cran.r-project.org/web/packages/unmarked/unmarked.pdf) which contains commands that allow for, among other things, analysis of distance sampling, occupancy modelling, and repeated count data.

Next up – Zombie Distance Analysis: On Blackboard you will find a file called Zdistance16.R. This file provides the completed code for analyzing the zombie distance sampling data. You can open the file in RStudio, so you will not need to type in code directly for the remainder of the lab. Notice that the file contains two pieces of information, first is the actual code to be run using R, and second is a series of comments I've made that describe what each piece of code accomplishes. You will recognize the comments based on the double # that precedes them. This is a generic feature of R that allows you to include information in your code that is not actually read by the software, much like the "*/ /*" notation that we saw in the input files for MARK. The steps I'll provide below will parallel the progression of the code, and in the code I'll add comments using a quadruple hashtag (####) when it's time to progress onto the next step.

1. First we need to input the distance data into R. You'll find the data saved as the csv file 'distall' on Blackboard. It includes two columns of information. The first delineates whether the transect line fell in deciduous or coniferous forest, and the second gives the distance for each observed zombie.

2. There are two options for importing the file to R. First you can use the code based on the read.csv() command provided in step 1 to import the data. To do so you will need to change the file path name to reflect the actual directory your

file is stored in. Alternatively, use the “Import Dataset” option which can be accessed from the Tools menu or from the icon in the upper right window of R commander. If you go this route use the “From Text File” option, name the file “zDIST” (this is very important – if you name it something else it the remainder of the code will need to be modified), and be sure to use the Headings=Yes option and to specify that the file uses a comma separator (note these should be defaults). Click Import.

3. Now we need to load the unmarked package. Do this by running the install.packages() and library() commands. You may be prompted after a second to select a “CRAN Mirror”, which is just the server that the package will be pulled from. As far as I know it makes no difference which you select. I always choose the USA (CA1) mirror, which is housed at the University of California at Berkeley. If you are not asked to do this, just proceed with step 4.

4. Here, we’re going to create a new field that defines a difference between the two transects and identifies that the new grouping variable “transect” as a factor. The following three lines of code will identify transect as a covariate that unmarked will recognize.

5. This chunk of code will create a data frame, yDat, which is formatted correctly for a distance sampling analysis. When you look at it you’ll notice the observations have been pooled into 5 discrete bins in 100 m increments, and they are separated among the two transects.

6. Using the yDat and covs data frames, we’ll construct an input file that will contain all the data for our distance sampling analysis and will contain some other relevant information like transect length and the fact that this was a line-based sample, as opposed to point-based. I’ve also provided some code that will produce a histogram of your data to show how your detections varied based on distance from the transect line. Recall that in a distance analysis we assume that animal (zombie) distribution is random with respect to the line, but that our ability to detect the zombies decreases as we move away from the line. Hopefully your histogram is consistent with the later assumption.

   Also notice that when you create these figures in the plot window you can export them as image files using the “Export” feature.

7. Our next step will be to identify the best distance-detection function, given your data. Again, we assume that detection declines with distance from the line,

however, the specific way that detection changes with distance can be variable among studies for a variety of reasons. For example, detection may be high very close to the line but then may very rapidly decline. This might occur in very dense vegetation or in highly variable terrain. In contrast detection may decline slowly and progressively with distance, for example in even terrain or with more moderate vegetation. There are three general distance-detection functions we can use to approximate this: the half normal, hazard, and negative exponential functions. If you run the code associated with step #7, you can see what each of these three functions look like.

Notice that these three functions require different parameters. The half-normal is, like the name implies, half of a normal distribution which can be characterized by a single parameter, sigma (or the variance of the distribution). The hazard function requires two parameters that define the shape and scale of the curve, and the negative exponential requires one parameter describing the rate of decline. For the purpose of these examples I've defined these all arbitrarily with a value of 10 for each respective parameter.

8. To test which of the three distance-detection functions are most appropriate given our data, we can run three null models and only change the shape of the distance-detection function. This will also be your first chance to run a distance model. Take a quick look at the code. Importantly, the ~1 ~1 component is where we define the structure of the model, and "1" defines an intercept-only model. The first term is associated with the detection parameter, and the second associated with the density parameter. Here we are creating three objects, each of them distance sampling models with an intercept-only structure on both detection and density parameters, where the only difference is the shape of the distance-detection function. *Note – if you get an error message saying that the Hessian is singular, you can ignore it and move on.

9. Now, using the same principles of AIC model selection that we've used in past weeks, we can compare the relative fit of our three alternative models, given our data. The first batch of code here combines the model results into a single object, null.fit, and the next three statements produce an AIC table, provides us with the parameter estimates from each model, and the third gives the standard errors for the estimates. You will want to check two different things with these results. First, use AIC to determine which model provides the best fit to the data. Second check the standard errors for each model to see that the parameters are actually being estimated. A returned value of "NaN" indicates convergence was not reached on the parameter, suggesting the data were insufficient to fit the model in general. Notice that "NA" means something different – that the specific

parameter is not relevant to the given model. The "best" model will be the one that produces the lowest AIC score while also estimating all necessary parameters. Based on these results, determine which function best fits your specific data. Notice that because you all collected individual data, you will likely find that some differences in results when compared to your class mates.

10. After finding the best distance-detection function, the next step will be to test some other alternative models. Really here we only have one other variable to consider, and that is the difference among the two transects that you ran (which also reflects the difference among the two observers). The four lines of code given here each include a different combination of either running intercept-only structure to detection or density, or allowing detection and/or density to differ among the two transects. Remember that the model structure is given by the notation ~ detection ~ density, where ~1 reflects a null model and ~Trans gives a model where either detection or density is allowed to differ among the two transects. To come full circle to previous weeks, the linear model for a ~Trans structure to the detection function would look something like the following:

$$Detection = \beta_0 + \beta_1 * Transect$$

    where Beta 0 would be the model intercept, and Beta 1 is the effect of transect number on detection. Thus, when calling the summary() function on the model p.Trans, the sigma(Intercept) Estimate is the value for Beta 0 (model intercept) and the sigmaTrans2 Estimate value provides the effect of being Transect 2, where Transect 1 is defined strictly by the intercept (remembering that Transect Number is a factor, making it equivalent to a dummy variable in our prior weeks).

    Based on your assessment of the AIC results and coefficient estimates from this 4-model set, determine whether there were differences in either zombie detection or density between the two transect lines, and what those differences were (i.e. which transect was higher or lower in detection or density, if at all.

11. A major objective of a distance analysis is to actually generate an estimate of density (animals per unit area) and abundance (total number along the transect line). For our distance analysis in unmarked, our actual response variable is density, and that is what our model results report. To generate a density estimate, we will use the backTransform() function, which will produce and report the density estimate. In this case if we use the Null model density is averaged across the two transect lines.

12. Next we can convert density into an actual estimate of the total number of zombies found along the transect lines. This will require doing a small amount of math to take the estimate of density and convert it to abundance given the size of the area we sampled. Remember the data are based on two 1000 meter transects, and our maximum distance was 50 m from the line. However, to compute the actual abundance of zombies we've got to first compute the effective transect width. This uses the distance-detection function to compute the actual functional area that we covered during the survey, accounting for the fact that detection declines as we move further away from the survey line. The 'integrate' function is what computes the area under the detection curve by, you guessed it, integrating the half-normal function.

13. Based on the effective width we can convert density to an estimate of abundance, and the second batch of code here will use bootstrapping to derive a standard error for that estimate.

14. The last batch of code will cover how to plot the detection estimates to produce a distance-detection graph such as that contained on the first page of the handout.

Assignment. We've just ran a fairly involved Distance Sampling analysis in program R. From it, you should have assessed the appropriate distance-detection function, evaluated support for differences among the two transect lines, estimated the abundance of zombies along the transect lines, and plotted your detection functions. We also covered the basic fundamentals of working in program R. Based on what you've learned during the lab, answer/complete the following questions and turn them in as the assignment for this week.

1. In the first part of the lab you ran a linear regression comparing a measure of mass to a measure of temperature. Was there an effect of temperature on mass, was this effect statistically significant, and what was the direction of the effect?
2. Which distance-detection function did you find was best supported by the data? What does the shape of this function imply about your ability to detect zombies in the Maine woods? Please include with this answer the AIC table* that summarized the results of your three distance-detection models, and your final plot** of detection by distance with the overlaid distance-detection function (i.e. the graph from step 14).
3. A colleague of yours working in a major urban area collected similar data, and found that a hazard function with a shape parameter=3.9 and a scale parameter =8.2 best fit the zombie distance-detection relationship in her system. What does this imply about zombie detection in an urban setting versus what we found in a

forest environment? How would that information be relevant for survival during a zombie viral outbreak?

4. Did you find any differences among the two transect lines with respect to density or detection? If so, what were those differences? Use supporting information from your AIC table or parameter estimates to support your answer, and also include your final AIC table (the second one from the analysis).
5. What was your final estimate of zombie abundance? What was your estimate of zombie density? If we surveyed a 100 ha woodlot, what would you predict was the total abundance of zombies within the entire woodlot? What inherent assumption are you making when extrapolating the estimates from your two transect lines to the woodlot as a whole?

* Notice that AIC tables can be copied out of R into excel using the write.table() command. If you type ?write.table as an R command line, you will find instructions for exporting using this command. The same general formatting rules we've used all semester apply.

** Graphs can be saved as .jpeg files using the "Export" feature in the graphing window.