Analysis of Environmental Data

Deterministic (Mathematical) Functions

(Written by Kevin McGarigal, but borrowed heavily from Ben Bolker, Ecological Models and Data (2007) and Michael Crawley, The R Book (2007))

The purpose of this lab exercise is to familiarize you with a range of useful deterministic (or mathematical) functions in R. Deterministic functions are used to define the ecological process under consideration, while probability distributions (next lab) are used to define the stochastic component(s) of the model (i.e., the error distributions). Together, these two components form the basis for most parametric statistical models. It is not practical to provide a comprehensive inventory of deterministic functions, since there is literally no limit number of possibilities. Rather, the intent of this lab is to introduce you to deterministic functions by way of a few examples and give you some of the R skills needed to develop and/or examine other functions. See Bolker (2007) and Crawley (2007) for excellent descriptions of a large number of functions. Here is outline of what is included in this lab exercise:

1. Set up your R work session

Open R and set the current working directory to your local workspace, for example:

```
setwd('c:/work/stats/ecodata/lab/deterministic/')
```

Load the biostats library (which is not a formal library) by typing, substituting the appropriate path:

```
source('.../biostats.R')
```

2. What is a deterministic (mathematical) function

Statistical models are comprised of a deterministic model(s) and a stochastic model(s). The deterministic part is the average, or expected pattern in the absence of any kind of randomness or measurement error (i.e., stochasticity). The deterministic model can be phenomenological (i.e., simple relationship based on the observed patterns in the data), mechanistic (i.e., relationship based on underlying environmental theory), or even a complex individual-based simulation model. Importantly, the deterministic model is intended to represent the underlying environmental process, and estimating the parameters of this model is usually the focus of statistical modeling.

It is usually a good idea to have an a priori idea of the expected form of the deterministic model, since this explicitly ties the model to environmental theory. A deterministic model constructed in

this manner, such that the model parameters have a mechanistic relationship to an environmental process, is sometimes referred to as a "mechanistic" model. In this case, the model parameters have a direct environmental interpretation. However, there may be times when a purely phenomenological description of the data is sufficient; that is, when the deterministic model is derived based on the patterns observed in the data and not on underlying theory per se. In this case, the model parameters do not have an explicit environmental interpretation. In either case, you need to know something about a wide range of mathematical functions and how to define and explore them in R. A few examples will serve to illustrate the methods for writing and examining mathematical functions.

3. Linear function – Oregon birds example

In this example, we will create and examine a simple *linear function* to describe the relationship between the extent of late-successional forest and the relative abundance of brown creepers across 30 subbasins in the Oregon Coast Range.

First, import the data set(s) and check the results by typing:

```
birds<-read.csv('bird.sub.csv',header=TRUE)
hab<-read.csv('hab.sub.csv',header=TRUE)
str(birds)
str(hab)
```

The data represent standardized breeding bird counts (birds) and a variety of habitat variables (hab) for the 30 landscapes (n=30). See the corresponding metadata file (birdhab.meta.pdf) for a description of the data.

Next, it is convenient to merge the bird and habitat data into a single file by typing:

```
birdhab<-merge(hab,birds,by=c('basin','sub'))
str(birdhab)
```

In this case, the relational fields that link the bird and habitat data sets are 'basin' (3 unique basins) and 'sub' (10 unique subbasins in each basin). The resulting data frame contains a lot of variables, but for our purposes we are only interested in two: (1) brown creeper abundance (BRCR), and (2) the amount of late-successional forest (ls).

Let's begin by plotting the data:

```
plot(birdhab$ls,birdhab$BRCR)
```

The first thing that you should notice about the plot is that brown creeper abundance appears to increase linearly with increasing percentage of the landscape comprised of late-successional forest. Thus, it would be logical to propose a simple linear model to describe this relationship; i.e., to represent the deterministic component of a statistical model for this data. Note that this is a phenomenological description of the relationship as we derived this deterministic model by

observing the patterns in the data rather than hypothesizing this relationship a priori (i.e., before looking at the data). The second thing that you should notice is that the relationship is not perfectly linear; i.e., there is considerable variability about the trend. This represents the stochastic component of the model. For now, we will ignore this stochastic component and focus solely on describing the deterministic component.

Before going on, let's pause and learn how to change the look of the plot. Specifically, let's change the range of the plot axes, modify the axis labels, add a title, change the plotting symbol and color and change the size of the axis labels and title from the default. Conveniently, the plot function has lots of arguments that allow us to control these and many more aspects of the plot. Try the following:

```
plot(birdhab$ls,birdhab$BRCR,xlim=c(0,120),ylim=c(0,1.2),
  xlab='large sawtimber',ylab='Brown Creeper abundance',
  main='linear model example',cex=1.5,cex.lab=1.5,cex.main=2,
  pch=19,col='gray')
```

The xlim and ylim arguments change the min and max of the x and y axis, respectively. The xlab and ylab arguments change the labels of the x and y axis, respectively, and are specified as text strings in quotes. The main argument adds a plot title and is specified as a text string in quotes. The cex argument expands the plotting symbol to (1.5) times the default size. The cex.lab and cex.main expand the size of the axis labels and main title, respectively, to (1.5) and (2) times their defaults, respectively. The pch argument changes the plotting symbol to type (19), which is a solid circle. Lastly, the col argument changes the color of the plotting symbol to (gray), also specified as a text string in quotes.

It is extremely useful to know how to use R to calculate and plot curves (or in this case a straight line), since this will allow you to visualize a function and interpret its parameters – at least phenomenologically in terms of how the parameters affect the shape of the function.

A simple linear function has the following form:

$$y = a + b*x$$

where a is the intercept (value of y when x=0) and b is the slope.

As a first step, we can simply use R as a calculator to plug values into functions. For example, we can compute the value of y for a given value of x and the values for parameters a and b, as follows:

```
a<-0
b<-.01
x<-50
y<-a + b*x
y
```

Since most functions in R operate on vectors (or "are vectorized"), we can calculate values for a

range of inputs or parameters with a single command. For example, it may be more useful to calculate the value of y for a range of x values, which we can do by first creating a vector of x values and then plugging this into the equation as before:

```
xvec<-seq(0,100)
y<-a + b*xvec
y
```

Next simplest, we can use the curve() function to have R compute and plot values for a range on inputs: use add=TRUE to add curves to an existing plot. Let's start by plotting the original data and then overlay the line we defined above:

```
plot(birdhab$ls,birdhab$BRCR)
curve(0+.01*x,from=0,to=100, add=TRUE)
```

Here, the curve function evaluates the supplied function (in this case the explicit linear function) for the variable x over the range specified (in this case from 0-100). Note, not specifying add=TRUE will cause a new plot to be drawn.

Let's add a second line to the plot, but with a different intercept and slope:

```
curve(.05+.007*x,from=0,to=100,lty=2,add=TRUE)
```

Note, we can continue modifying the model parameters until we get the best eyeball fit to the data, but this would be extremely time-consuming and highly subjective. Instead, we can use objective statistical procedures to come up with the "best" estimate of the parameters (more on estimation procedures later).

Lastly, we can define the deterministic function first and then call it from inside the curve() function, as follows:

```
linear<-function(x,a=0,b=1) a + b*x
curve(linear(x,a=.05,b=.008),from=0,to=100,lty=3,add=TRUE)
```

Here, we used the function() command to define a function called *linear* containing three arguments: x is the data to submit to the function; a and b are the intercept and slope parameters of the linear equation. In this case, we also specified default values for the parameters a and b. Once defined, this function can be used by other functions that take functions as input. For example, the first argument of the curve() function is the function to be plotted. In this case, we simply called the linear function and supplied values for the parameters a and b. Note, if we had omitted values for a and b, the defaults would have been used; prove this to yourself.

Lastly, let's add a legend to the plot since we have three different lines plotted. To do this, we use the legend() function as follows:

```
legend(5,0.9,legend=c('a=0,b=.01','a=.05,b=.007','a=.05,b=.008'),
```

```
lty=c(1,2,3),bty='n',cex=1.5)
```

The first two arguments specify the x and y axis coordinates of where you want to place the legend (note, you can also use convenient shortcuts such as 'topleft'). The legend argument is a vector of text strings with the labels for the legend elements, here giving the parameter values for each of the corresponding lines. The lty argument is a vector of same length as the legend vector, 3 in this case, specifying the line type for the corresponding lines. The bty argument is used here to specify no box ('n') around the legend. And as before, the cex argument expands the character size to (1.5) times the default.

4. Logistic function – Oregon birds example

In this example, we will create and examine a simple *logistic function* to describe the relationship between total basal area of trees and the presence/absence of brown creepers across 1046 sample plots in the Oregon Coast Range.

First, import the data set(s) and check the results by typing:

```
birds<-read.csv('bird.sta.csv',header=TRUE)
hab<-read.csv('hab.sta.csv',header=TRUE)
str(birds)
str(hab)
```

The data represent standardized breeding bird counts (bird) and a variety of habitat variables (hab) for the 1046 sample plots (n=1046). See the corresponding metadata file (birdhab.meta.pdf) for a description of the data. Note the difference between this and the previous example. Here, the observations represent 50-m radius circular plots, whereas in the previous example the observations represented ~300-ha subbasins.

Next, it is convenient to merge the bird and habitat data into a single file by typing:

```
birdhab<-merge(hab,birds,by=c('basin','sub','sta'))
```

In this case, the relational fields that link the bird and habitat data sets are 'basin' (3 unique basins), 'sub' (10 unique subbasins in each basin), and 'sta' (32-37 unique stations in each subbasin). The resulting data frame contains a lot of variables, but for our purposes we are only interested in two: (1) brown creeper abundance (BRCR), and (2) total basal area of trees (ba.tot).

First, we need to transform the bird count data into presence/absence data (i.e. binary). For this purpose, we can use the data.trans() function in the biostats library, as follows:

```
bin<-data.trans(birdhab,method='power',var='AMCR:YRWA',exp=0,plot=FALSE)
```

The power transformation with an exponent of zero transforms any non-zero number to 1. Let's begin by plotting the data:

```
plot(bin$ba.tat,bin$BRCR)
```

The first thing that you should notice about the plot is that the points on the y-axis are either 0 or 1, representing the absence and presence, respectively, of brown creepers. The second thing that you should notice is that brown creeper presence appears to increase somewhat with increasing total basal area. Thus, it would be logical to propose a simple logistic model to describe this relationship.

A simple 2-parameter logistic function has the following form:

$$y = \frac{e^{a+b\cdot x}}{1 + e^{a+b\cdot x}}$$

where *a* is the "location" parameter that shifts the curve left or right, and *b* is the 'scale' parameter that controls the steepness of the curve. In this case, *y* represents the probability of 'presence'.

As before, we can use R as a calculator to plug specific values into the logistic function, as follows:

```
a<--2 #note, this is minus 2
b<-.05
x<-50
y<-(exp(a + b*x)/(1 + exp(a + b*x)))
y
```

Or we can compute the y's for a vector of x's, as follows:

```
xvec<-seq(0,200)
y<-(exp(a + b*xvec)/(1 + exp(a + b*xvec)))
y
```

Or we can define a function and then use curve() to plot the function for a range of input values and add it to the original scatter plot, as follows:

```
plot(bin$ba.tot,bin$BRCR)
logistic<-function(x,a,b) (exp(a + b*x)/(1 + exp(a + b*x)))
curve(logistic(x,a=-2,b=.05),from=0,to=200,add=TRUE)
```

It is often useful to know about the limits of functions and whether the function increases or decreases toward them: the limiting slope. Does the function shoot up or down (a derivative that "blows up" to positive or negative infinity), change linearly (a derivative that reaches a positive or negative constant limiting value), or flatten out (a derivative with limit 0)? To figure this out, we need to take the derivative with respect to x and then find its limit at the edges of the range.

We can use the deriv() function to calculate derivatives; it will give us a function that will compute the derivative for specified values of x (x=50 in the example below). But we have to use expression()

to stop R from trying to interpret the formula:

```
logis<-expression(exp(a + b*x)/(1 + exp(a + b*x)))
dfun<-deriv(logis,'x',function.arg=TRUE)
dfun(50)
```

Or we can plot the derivative for a range of inputs, as follows:

```
xvec<-seq(0,200)
y<-dfun(xvec)
plot(xvec,attr(y,'gradient'),type='l',ylab='derivative')
abline(h=0,col='red')
```

Note, we had to use attr(...,'gradient') to retrieve the derivative from the y object. The plot shows us that the derivative is maximum when x is approximately 50 – the inflection point in the logistic curve – and that it asymptotically approaches 0 as x gets large.

5. Ricker function – Striped bass example

In this last example, we will create and examine a *ricker function* to describe the relationship between striped bass stock and recruitment over a 24 year period. The Ricker function is a common model for density-dependent population growth; if per capita fecundity decreases exponentially with density, then overall population growth will follow the Ricker function.

First, import the data set(s) and check the results by typing:

```
striper<-read.csv('striperSR.csv',header=TRUE)
str(striper)
```

The data represent estimated stock levels (number of females) and recruits (age 1 numbers appropriately lagged to match the producing stock) and corresponding standard deviations in the estimates for a 24 year survey period. See the corresponding metadata file (striper.meta.pdf) for a description of the data.

Let's begin by plotting the data:

```
plot(striper$stock,striper$recruits)
```

The plot indicates that the number of recruits initially increases steadily with increasing stock, but then appears to peak and decline when the stock gets relatively large. The Ricker function is one of several functions that has been used to define this sort of relationship, and has the following form:

$$y = a \cdot x \cdot e^{-b \cdot x}$$

where it starts off growing linearly with slope a and has its maximum at $x=1/b$. The Ricker function

is widely used as a phenomenological model for environmental variables that start at zero, increase to a peak, and decrease gradually back to zero.

As before, we can use R as a calculator to plug specific values into the Ricker function, as follows:

```
a<-0.5
b<-1/50000
x<-20000
y<-a*x*exp(-b*x)
y
```

Or we can compute the y's for a vector of x's, as follows:

```
xvec<-seq(0,80000,length=100)
y<-a*xvec*exp(-b*xvec)
y
```

Or we can define a function and then use curve() to plot the function for a range of input values and add it to the original scatter plot, as follows:

```
plot(striper$stock,striper$recruits)
ricker<-function(x,a=1,b=1) a*x*exp(-b*x)
curve(ricker(x,a=.7,b=.00002),from=0,to=80000,add=TRUE)
```

As before, we can calculate and plot the derivative for a range of values, as follows:

```
rick<-expression(a*x*exp(-b*x))
dfun<-deriv(rick,'x',function.arg=TRUE)
xvec<-seq(0,80000,length=100)
y<-dfun(xvec)
plot(xvec,attr(y,'gradient'),type='l',ylab='derivative')
abline(h=0,col='red')
```

Further, we can extract the value of "stock" corresponding to the peak of the Ricker curve where the first derivative is zero, and then add to the plot a vertical line at the peak, as follows:

```
d1<-attr(y,'gradient')
(inflection<-xvec[d1>=-.001 & d1<=.001])
abline(v=inflection,col='blue',lty=2)
```

Can you prove to yourself that the Ricker curve asymptotically approaches zero as x gets large?

6. Bestiary of deterministic functions

This section provides a bestiary of functions that are useful in environmental modeling (taken directly from Bolker), but realize that there are numerous other alternatives. Use curve() to examine

the shape of each of these functions. See Appendix for sample plots of each function.

```
#polynomial functions
linear<-function(x,a,b) a + b*x
quadratic<-function(x,a,b,c) a + b*x + c*x^2
cubic<-function(x,a,b,c,d) a + b*x + c*x^2 + d*x^3

#piecewise polynomials
threshold<-function(x,a1,a2,s) ifelse(x<s,a1,a2)
hockey<-function(x,a,s) ifelse(x<s,a*x,a*s)
piecewise<-function(x,a,b,s) ifelse(x<s,a*x,a*s+b*(x-s))

#rational functions (polynomials in fractions)
hyperbolic<-function(x,a,b) a/(b+x)
bevholt<-function(x,a,b) a*x/(b+x)
holling3<-function(x,a,b) a*x^2/(b^2 + x^2)
holling4<-function(x,a,b,c) a*x^2/(b + c*x + x^2)

#simple exponentials
exponential<-function(x,a,b) a*exp(b*x)
monomolecular<-function(x,a,b) a*(1-exp(-b*x))

#combinations of exponentials with other functions
ricker<-function(x,a,b) a*x*exp(-b*x)
powricker<-function(x,a,b,alpha) b*(x/a*exp(1-x/a))^alpha
tricker<-function(x,a,b,t,min=1e-04) {
    ifelse(x<t,min,b*((x-t)/a*exp(1-(x-t)/a)))
    }
logistic<-function(x,a,b) exp(a + b*x)/(1 + exp(a + b*x))
modlogistic<-function(x,eps,beta,phi) exp(eps*(phi-x))/(1+exp(beta*eps*(phi-x)))
norm2<-function(x,mu,sigma) 1/(sqrt(2*pi)*sigma)*exp(-((x-mu)^2/(2*sigma^2)))
norm3<-function(x,mu,sigma,hgt) hgt*exp(-((x-mu)^2)/(2*sigma^2))
halfnorm<-function(x,sigma,hgt) hgt*exp(-(x^2)/(2*sigma^2))

#power laws
power<-function(x,a,b) a*x^b
vonbert<-function(x,a,b,d,k) a*(1 - exp(-k*(a-d)*x))^(1/(1-d))
shepherd<-function(x,a,b,c) a*x / (b + x^c)
hassell<-function(x,a,b,c) a*x / ((b+x)^c)
```

7. Exercise – Marbled salamander dispersal

The purpose of the following exercise is to give you additional experience working with deterministic functions. You can work individually or in teams to complete the exercise.

Background.–The data for this exercise represent the dispersal of juvenile marbled salamanders from their natal ponds to neighboring ponds. The data were derived from a long-term study of marbled salamanders in western Massachusetts in which a cluster of 14 vernal pools were monitored continuously between 1999-2004. All juveniles were marked upon leaving their natal ponds. Subsequent recaptures at non-natal ponds were used to determine dispersal rates between ponds for first-time breeders (ftb). In the data set provided, the dispersal rates have been standardized to account for several factors, including the propensity for dispersal from each pond and the available distances between ponds owing to the unique configuration of ponds. The data set includes three variables: (1) dist.class = distance class, based on 100 m intervals; (2) disp.rate.ftb = standardized dispersal rate for first-time breeders, which can be interpreted as a relative dispersal probability; and (3) disp.rate.eb = standardized dispersal rate for experienced breeders, which can be interpreted as a relative dispersal probability.

Your assignment is to examine the data set provided (...\dispersal.csv) and find at least three alternative mathematical functions to describe the apparent relationship between juvenile dispersal and distance. Your specific tasks are as follows:

1. Examine the data set. Plot the relationship between juvenile dispersal (disp.rate.ftb) and distance (dist.class).
2. Select three different mathematical functions that provide plausible descriptions of the apparent dispersal-distance relationship. Each function must come from a different family of functions (see the bestiary of functions in section 6).
3. Fit the three functions to the data by eyeballing the fit; i.e., choose parameters for each of the models by trial and error. Overlay the curves on the original scatter plot. Be sure to label the axes and include an appropriate legend.
4. Hand in a one-page report containing your name, student ID, and a single figure (from step 3) containing a suitable figure caption.

Supplemental Exercise

The purpose of this supplemental exercise is to give you some initial experience in model fitting; i.e., in finding good estimates of your model parameters. While there are several alternative frameworks for estimating model parameters (to be discussed in detail in lecture), here we will use a simple brute force numerical optimization procedure known as Ordinary Least Squares. Specifically, you will build and implement a procedure for evaluating the goodness-of-fit of alternative parameterizations for one (or more) of your chosen deterministic functions, and compare the results to your subjective trial-and-error approach. Here are the suggested steps to follow:

1. Choose one of the three deterministic functions you selected above, but for simplicity sake make sure it is a two-parameter function (not three). Make sure the function is stored in memory; e.g.,

```
halfnorm<-function(x,sigma,hgt) hgt*exp(-(x^2)/(2*sigma^2))
```

2. Store the observed values of the dependent variable (disp.rate.ftb) in an object (to aid in the clarity of the scripting below).

3. Based on your best parameterization of the deterministic function (obtained via trial-and-error above), compute and store the predicted values of the dependent variable. Hint: use the function object above and supply the vector of x values (the independent variable; dist.class in this case), and the values of the model parameters. Note, this should return a vector of the same length as the object created in step 2 (verify this).

4. Compute the sums-of-squares. Note, one standard objective criterion for fitting models is to find the parameters that minimize the sum of the squared errors (SSE), where the error is defined as the difference between observed and predicted values. Here, compute the errors (or residuals) by taking the difference between the observed and predicted values. Note, subtracting a vector from another vector of the same length will do the subtraction on an element-by-element basis, such that the first element of the first vector will be subtracted from the first of the element of the second vector, and the second element of the first vector will be subtracted from the second element of the second vector, and so on, returning a vector of differences. Next, square these values (again, squaring the vector will result in squaring each element separately) and sum them. You now have your objective measure of how well your model parameterization fit the data – the lower the SSE, the better the fit.

5. Next, you need to evaluate a reasonable parameter space to search for the "best" values; i.e., the ones that minimize SSE. While there are efficient algorithms for doing this, you are going to use an inefficient brute force method that exhaustively searches the entire parameter space (and learn some R skills in the process).

    a. Create search vectors, one for each parameter, containing the sequence of parameter values you want to evaluate. Note, the sequences should be sufficiently broad in range and precise enough to find reasonable "optimal" values. For example, if you wanted to search values between 400 and 800 to the nearest integer for parameter 1, and values between 0.1 and 0.6 by increments of 0.01 for parameter 2, you could create the following two objects:

    ```
    param1<-seq(400, 800)
    param2<-seq(0.1, 0.6, by=0.01)
    ```

    b. Create a storage object to hold the result of your objective function (SSE) for each combination of parameter values. For example, you can create a matrix with the number of rows equal to the number of unique values of parameter 1 and the number of columns equal to the number of unique values of parameter 2 as follows:

    ```
    out<-matrix(NA, nrow=length(param1), ncol=length(param2))
    ```

    c. Create a double loop to loop through each combination of values of parameter 1 and 2. Here, we want to compute our objective function (SSE) for each unique combination of

parameter 1 values and parameter 2 values. One way to do this is to loop over the values of parameter 1 and then inside this loop, loop over the values of parameter 2. The instructor will go over loops with you in more detail, but here is the basic structure:

```
for(i in 1:length(param1)){
    for(j in 1:length(param2)){
        here you put the script for computing your objective function from steps 3-4 above
        out[i,j]<-the result of your objective function (e.g., SSE)
    }
}
```

Note, in the above script, the indices 'i' and 'j' are arbitrary, and param1 and param2 would be replaced with the names of the objects you created in step 5a.

6. Once you have completed step 5 you should have a matrix of values for your objective function (SSE) evaluated over the specified parameter space. Now you need to find the "best" parameter values by finding the set that minimized the objective function (SSE). Here is some code to this:

```
inds<-which(out==min(out), arr.ind=TRUE)
param1best<-param1[inds[,1]]
param2best<-param2[inds[,2]]
```

Note, in the script above, the which function returns the row and column index in which out==min(out); i.e. ,the row and column in which the objective function (SSE) has the minimum value.

7. Next, plot the objective function (SSE) as a surface to visualize how various combinations of parameter values perform. There are lots of ways to do this, but here we will use a simple contour plot, as follows:

```
contour(x=param1, y=param2, z=out, nlevels=50)
```

Note, in the script above, replace param1 and param2 with the names of the objects you created in step 5a. Also note that the contour function expects the 'z' value to be stored in a matrix, and the nlevels argument can be used to change the density of contour lines.

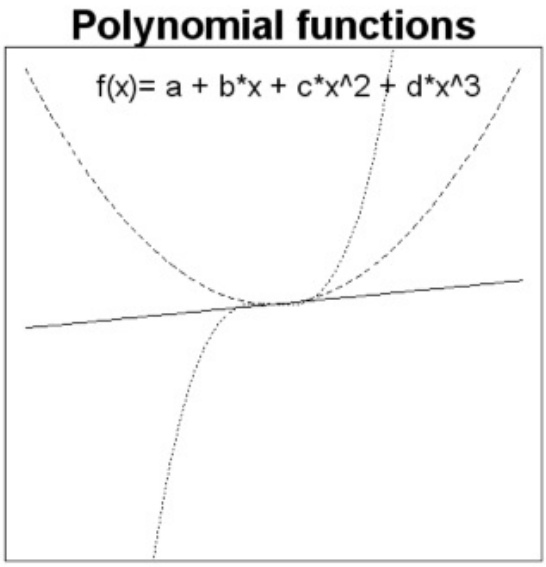
8. Next, add to your contour plot a point indicating the "best" estimates of your parameters based on your objective function and your brute force numerical optimization procedure. Use the low-level points() function to do this.

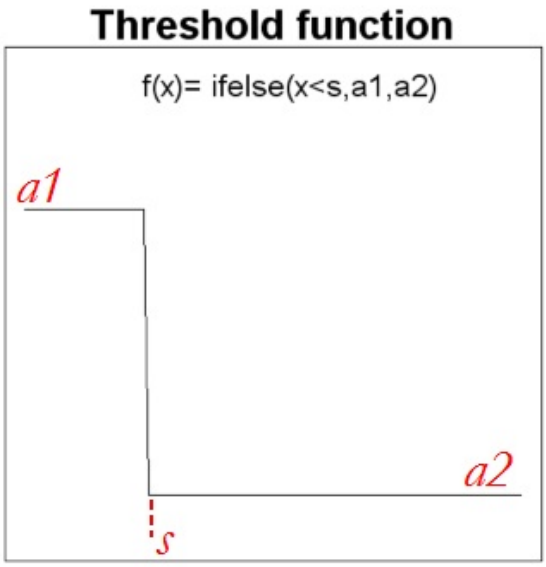
9. Next, create a scatterplot of your independent variable (dist.class) and dependent variable (disp.rate.ftb) with appropriate labeled axes (as in the main exercise above). Plot your original fitted curve from your subjective trial-and-error approach, and then add your best fitting curve based on the objective procedure above (changing line type or color) and add a legend.

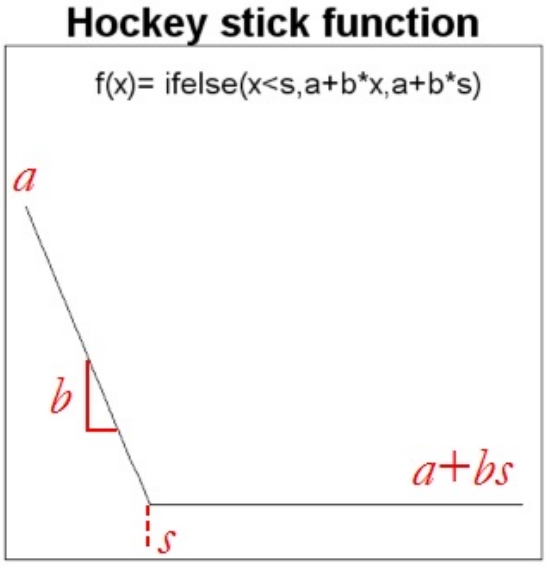
10. Lastly, repeat steps 5-6 above but this time change your objective function to "least absolute

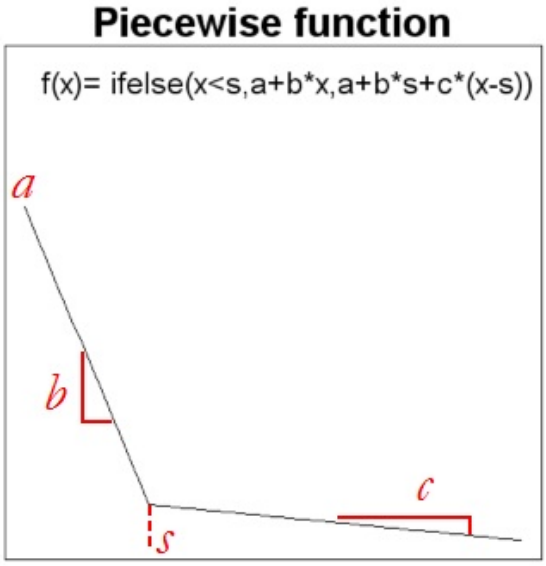
error" (LAE) instead of SSE. Here, sum the absolute value of the errors (use the abs() function to return the absolute values of the errors) instead of the squared errors. Add the best fit based on LAE to the scatter plot from step 9 and adjust the legend accordingly and turn this figure in for credit.

Appendix. Sample plots for a bestiary of deterministic functions.

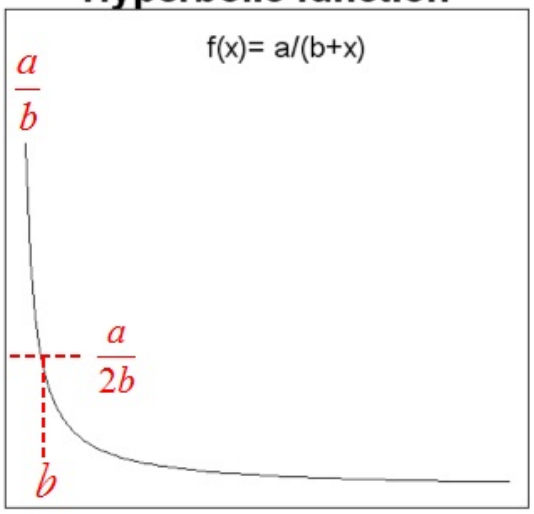




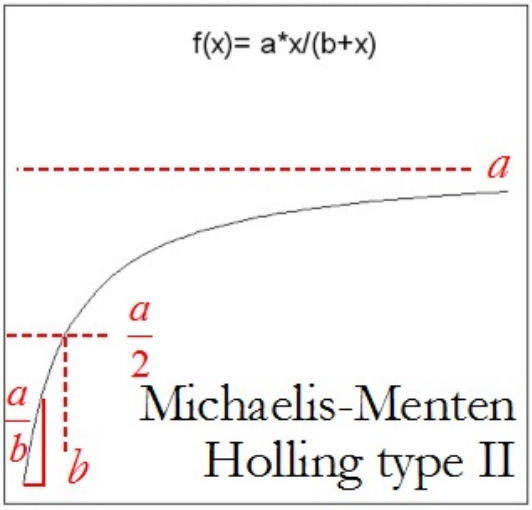




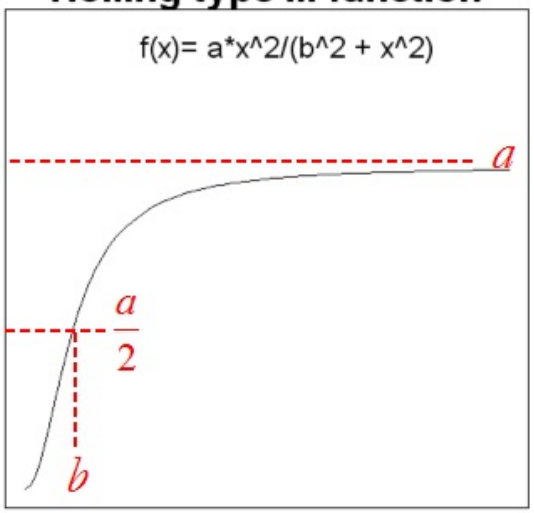
Hyperbolic function



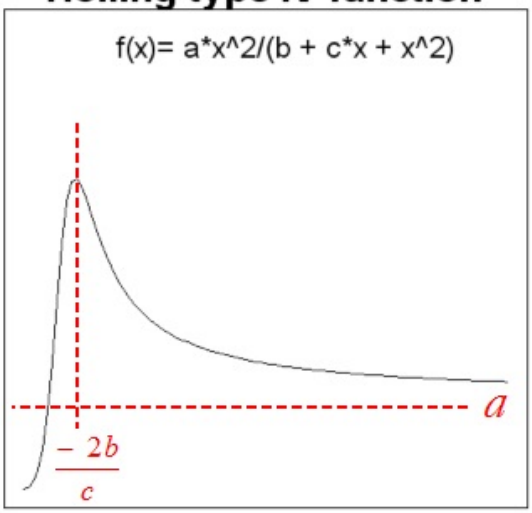
Beverton-Holt function



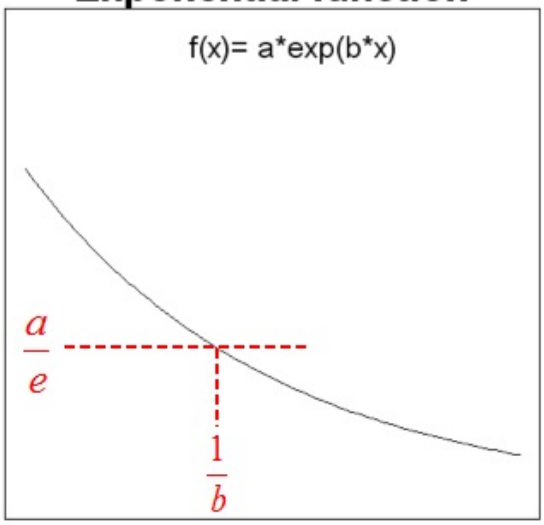
Holling type III function



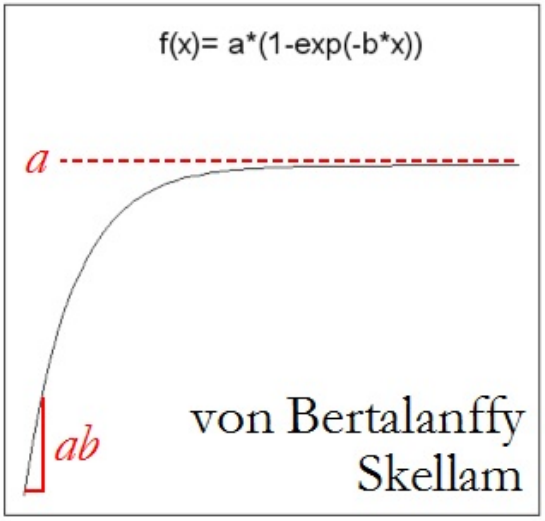
Holling type IV function



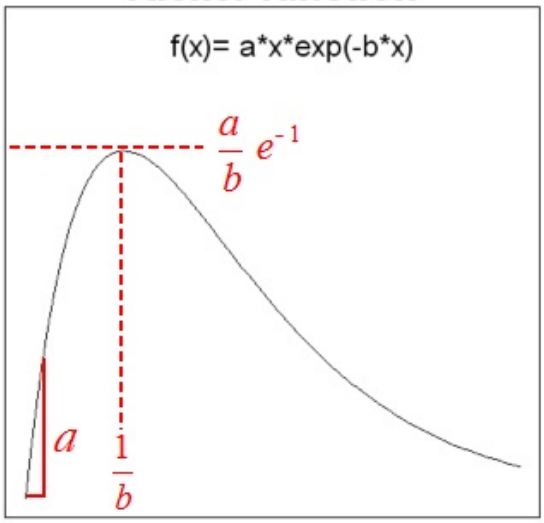
Exponential function



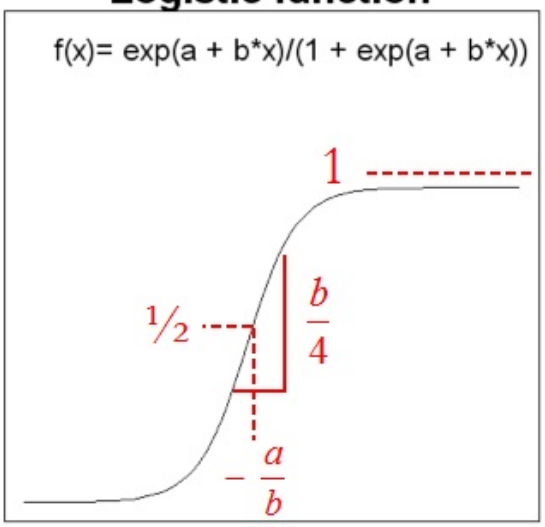
Monomolecular function



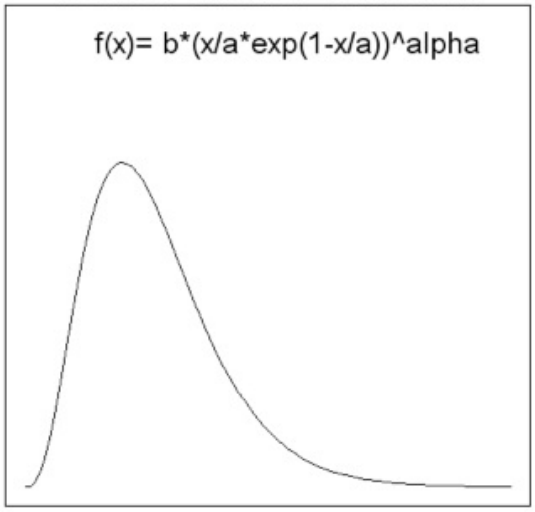
Ricker function



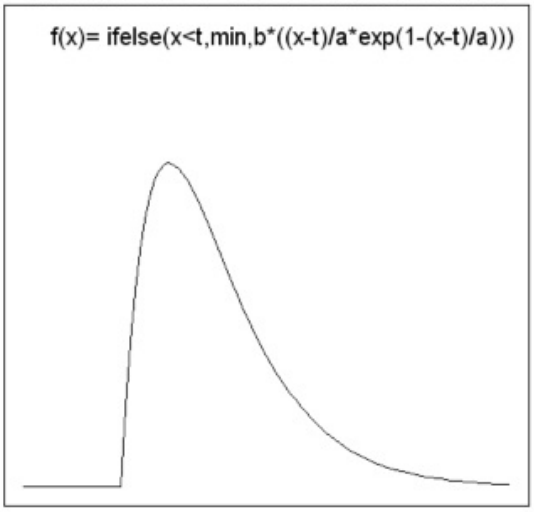
Logistic function



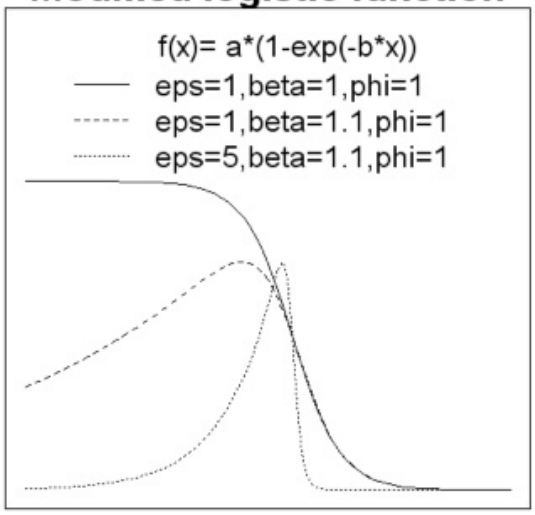
Power Ricker function



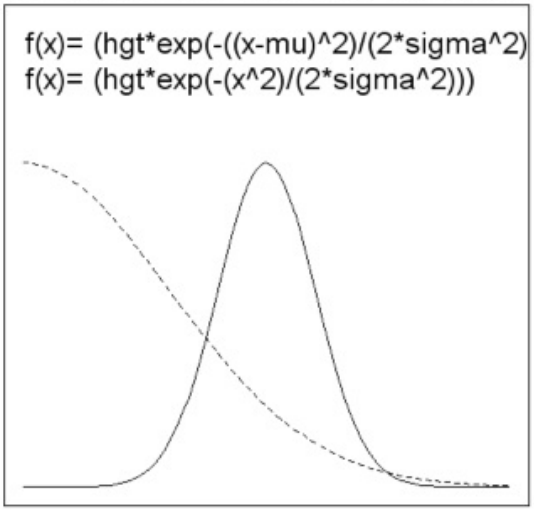
Truncated Ricker function



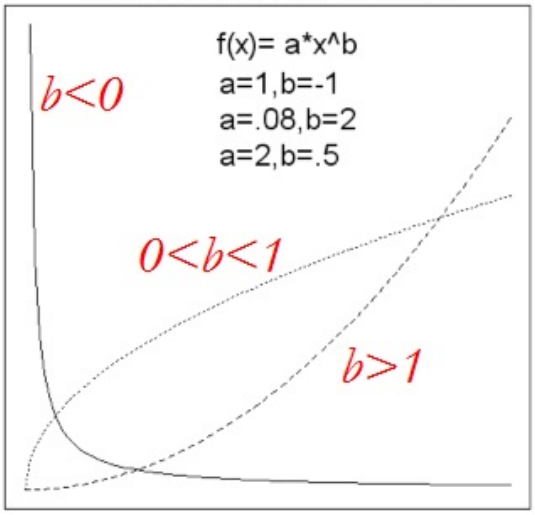
Modified logistic function



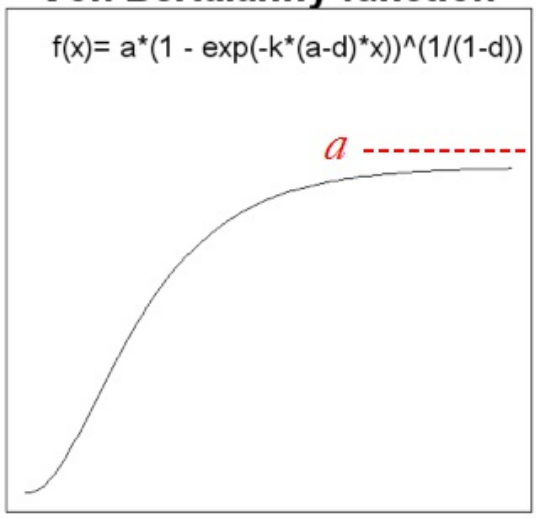
Normal & Half Normal functions



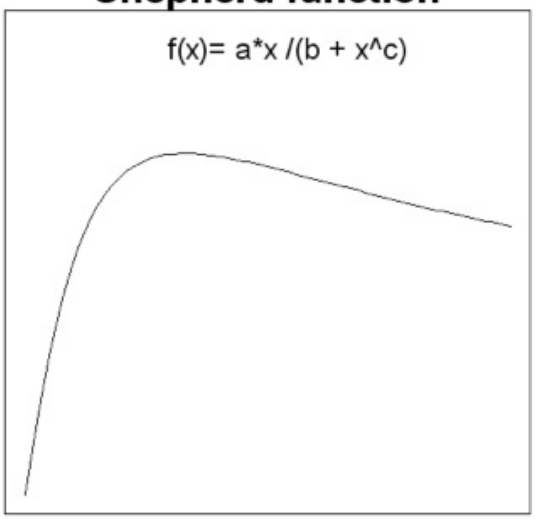
Power laws function



Von Bertalanffy function



Shepherd function



Hassell function