Analysis of Environmental Data

Stochastic Simulation

(Written by Kevin McGarigal, but borrowed heavily from Ben Bolker (2008))

The purpose of this lab exercise is to introduce you to techniques and ideas related to simulating environmental patterns in R. The main goals are to show you to generate patterns you can use to sharpen your intuition and test your estimation tools, show you how to estimate statistical power by simulation, and, with an example, illustrate the flexibility of R for simulating environmental patterns and processes. Of course it is impractical to illustrate the full range of uses of stochastic simulation. Rather, the intent here is to simply wet your appetite for the great utility of simulation. Here is an outline of what is included in this lab exercise:

1. Introduction

Simulation is sometimes called forward modeling, to emphasize that you pick a model and parameters and work forward to predict patterns in the data. Environmental scientists often use simulation to explore the patterns that emerge from environmental models. Often they use theoretical models without accompanying data, in order to understand qualitative patterns and plan future studies. But even if you have data, you might want to start by simulating your system.

You can use simulations to explore the functions and distributions you chose to quantify your data; in other words, to explore possible statistical models for your data. If you can choose parameters that make the simulated output from those functions and distributions look like your data, you can confirm that the models are reasonable – and simultaneously find a rough estimate of the parameters.

You can use simulated "data" from your system to test your estimation procedures. Since you never know the true answer to an environmental question – you only have imperfect measurements with which you're trying to get as close to the answer as possible – simulation is the only way to test whether you can correctly estimate the parameters of an environmental system. It's always a good idea to test such a best-case scenario, where you know that the functions and distributions you're using are correct, before you proceed to real data.

Power analysis is a specific kind of simulation testing where you explore how large a sample size you would need to get a reasonably precise estimate of your parameters. You can also use power analysis to explore how variations in experimental design would change your ability to answer environmental questions.

2. 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/simulation/')
```

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

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

3. Simulating static environmental processes

Static environmental processes, where the data represent a snapshot of some environmental system, are relatively easy to simulate. For static data, we can use a single function to simulate the deterministic process and then add heterogeneity or stochasticity. Often, however, we will chain together several different functions and probability distributions representing different stages in an environmental process to produce surprisingly complex and rich descriptions of environmental systems.

3.1. Linear regression model

Here we will illustrate the process of simulating a static environmental process using a simple linear regression model based on the now familiar Oregon birds data set. For this example, let's examine the relationship between brown creeper abundance and the extent of late-successional forest across 30 subbasins in the central Oregon Coast Range.

First, read in the bird and habitat data sets (see birdhab.meta.pdf for a complete description of the data sets), merge them, and plot the relationship between brown creeper abundance and the extent of late-successional forest, as follows:

```
birds<-read.csv('bird.sub.csv',header=TRUE)
hab<-read.csv('hab.sub.csv',header=TRUE)
birdhab<-merge(hab,birds,by=c('basin','sub'))
plot(birdhab$ls,birdhab$BRCR,pch=19)
```

What is the apparent relationship? How strong is it? Can we say with confidence that it is real or could it be a sampling artifact or a product of chance? To answer these questions we might fit a linear model with normal errors:

$$Y \sim \text{Normal}(a + bx, \sigma^2)$$

which specifies that Y is a random variable drawn from a normal distribution with a mean $a + bx$ and variance σ^2. This means that the i^{th} value of Y, y_i, is equal to $a + bx_i$ plus a normally distributed error with mean zero and variance σ^2. Let's fit this model using the lm() function in R, as follows:

```
fit1<-lm(BRCR~ls, data=birdhab)
fit1
```

Let's not worry about how we fit the model for now (that's coming later). Suffice it to say that the procedure used by the lm() function to fit this model finds us the "best" estimates of our moodel parameters, a, b and σ. Let's add the fitted line (i.e., predicted values) and a 95% confidence interval (again, let's not worry about how we find this interval for now) to the scatterplot, as follows:

```
abline(reg=fit1)
ci.lines(fit1)
```

Look's pretty good, doesn't it? But let's remember that the data represent a *single* snapshot of the environmental system. If the regression model is the truth, what might the data look like if we were to obtain another snapshot? This is where stochastic simulation comes in to play. We can simulate the system using our model and see what kind of range of data we might observe. This can give us great insight into the level of certainty we have in our model. This is in fact the theoretical basis for the frequentist approach to statistical inference, in which we evaluate the likelihood of our data given the model, which implicitly means how likely would we observe our original data if we were to repeatedly sample the system. This is exactly what simulation allows us to do – repeatedly sample the environmental system under the assumption that the model is the truth.

Let's create a simulation for our linear model. First, lets create a vector of values for x, which represents the percentage of the subbasin comprised of late-successional forest. Here, we could allow x to vary randomly between 0-100 to reflect sampling a different set of landscapes each time, or we could fix the values at the original values of x to reflect repeated random sampling of these same subbasins. Both approaches are legitimate, so the choice depends on the objective of the simulation. For our purpose, let's keep the original values of x and let only the brown creeper abundance to vary among simulations.

```
xvec<-birdhab$ls
```

Next, let's store the intercept and slope values from the fitted linear model so that we can use them more conveniently in our simulation:

```
a<-coef(fit1)[1]
b<-coef(fit1)[2]
```

Note, there are several ways to extract the coefficients from the fitted lm() object. Here we used the coef() function which extracts coefficients from modeling functions. The indexing is used to extract the first and second elements of the vector of coefficients extracted from the fit1 object, which corresponds to the intercept (a) and slope (b) parameters, respectively.

Next, let's calculate the deterministic part of the model:

```
y.det<-a + b*xvec
```

Note, because xvec is a vector, the mathematical equation produces a vector for output. Y.det contains a single element, the predicted value of *Y*, for each element of xvec. Next, let's pick random normal deviates (errors) with the mean equal to the deterministic equation and the standard deviation equal to the residual standard error of the fitted model (i.e., the standard deviation of the residuals). First, we have to extract the residual standard error from the fitted object, which is stored in the summary() object in the component named sigma:

```
y.error<-summary(fit1)$sigma
```

Next, we generate a vector of random observations drawn from a normal distribution with a mean equal to the predicted value from our deterministic model, y.det, and a standard deviation equal to the residual standard error, y.error.

```
ysim<-rnorm(n=length(xvec),mean=y.det,sd=y.error)
```

Note, because y.det is a vector, we want to extract a vector of random numbers of equal length; consequently, we specify n=length(xvec), which means draw a vector of random numbers that contains the same number of elements as xvec, and for each random draw, let the mean equal the corresponding value of y.det and the standard deviation equal the constant y.error. The object ysim now holds a random sample of brown creeper abundances derived from our model. Let's add this new set of points to the scatterplot, as follows:

```
points(xvec,ysim,add=TRUE,col='red')
```

Is the new pattern of points the same as the original pattern? Are there any notable discrepancies? You might want to run the model a couple of more times to see how variable the results are? After running the model a few times, do you notice any problems with the model? In other words, does the model reproduce the patterns in the original data perfectly or are there issues with the spread of values or with the generation of illogical values?

One problem with the use of the *normal* distribution is that it is unbounded on the lower limit. Thus, negative values are possible. In this case, because the *y*-intercept is close to 0, the simulation is likely to produce negative values occasionally when $x \rightarrow 0$. Since brown creeper abundance cannot be negative, this is an undesirable behavior of the model. One way to fix this problem is to use the *gamma* distribution, which allows only positive values. I will not describe the gamma error model here, but the script is included for those that wish to try it. One issue that arises with the gamma distribution is that $y_i=0$ is not allowed, so we need to add a very small number to all the observations before we can use the model (see script).

3.2. Power analysis for the linear regression model

Power analysis in the narrowest sense means figuring out the (frequentist) statistical power, the probably of correctly rejecting the null hypothesis when it is false. While we are generally less concerned with power analysis in the conventional sense of hypothesis testing, we are very interested in the role of power analysis in addressing a much broader question: How do the quality and quantity of the data and the true properties (parameters) of the environmental system affect the

quality and of the answers to our questions about environmental systems?

For any real experiment or observation situation, we don't know what is really going on (the "true" model or parameters), so we don't have the information required to answer these questions from the data alone. But we can approach them by analysis or simulation. Historically, questions about statistical power could only be answered by sophisticated analyses, and only for standard statistical models and experimental designs such as one-way ANOVA or linear regression. Increases in computing power have extended power analysis to many new areas, and R's capability to run repeated stochastic simulations is a great help.

Here, we will illustrate the use of stochastic simulation for power analysis using the linear regression model above. Let's start by finding out whether we can reject the null hypothesis in a single experiment. To do this, we simulate a dat set with a given intercept and slope, and number of data points; run a linear regression; extract the *p*-value (recall, this represents the probability of observing our data if in fact it came from distribution described by the null model, which in this case means that brown creeper abundance is independent of ls or has no relationship to ls); and see whether it is less than our specified α criterion (usually 0.05), as follows:

```
xvec<-birdhab$ls
a<-coef(fit1)[1]
b<-coef(fit1)[2]
y.det<-a + b*xvec
y.error<-summary(fit1)$sigma
ysim<-rnorm(length(xvec),mean=y.det,sd=y.error)
fit<-lm(ysim~xvec)
summary(fit)$coefficients['xvec','Pr(>|t|)']
```

Note, many of the objects were defined previously, but to be safe, we defined them again here. Also, extracting *p*-values from R analyses can be tricky. In this case, the coefficients of the summary() of the linear fit are a matrix including the standard error, *t* statistic, and *p*-value for each parameter; I used matrix indexing based on the row and column names to pull out the specific value I wanted.

To estimate the probability of successfully rejecting the null hypothesis when it is false (the power), we have to repeat this procedure many times and calculate the proportion of the time that we reject the null hypothesis. First, we specify the number of simulations to run and set up a vector to hold the *p*-value for each simulation. Then, we repeat what we did above (without redefining some of the objects that have not changed, such as xvec, *a*, *b*, and y.error), each time saving the *p*-value in the storage vector:

```
nsim<-1000
pval<-numeric(nsim)
for(i in 1:nsim){
  y.det<-a + b*xvec
  ysim<-rnorm(length(xvec),mean=y.det,sd=y.error)
  fit<-lm(ysim~xvec)
  pval[i]<-summary(fit)$coefficients['xvec','Pr(>|t|)']
```

```
}
```

Next, we calculate the power by summing up how many times we rejected the null hypothesis at the specified α-level, and dividing by the number of simulations to convert it to a proportion:

```
sum(pval<0.05)/nsim
```

What is our power in this case? But this is the power to detect a slope of roughly b=0.006 with a sample size of N=30, given our specified statistical model. Usually we don't just want to know the power for a single experimental design. Rather, we want to know how the power changes as we change some aspect of the design such as the sample size or the effect size (slope, in this case). Thus, we have to repeat the entire procedure multiple times, each time changing some parameter of the simulation such as the slope or the sample size. Coding this in R usually involves nested "for" loops.

Here is an example to examine how power changes as a function of the slope:

```
nsim<-1000
pval<-numeric(nsim)
bvec<-seq(-.01,.01,by=0.001)
power.b<-numeric(length(bvec))
for(j in 1:length(bvec)){
  b<-bvec[j]
  for(i in 1:nsim){
    y.det<-a + b*xvec
    ysim<-rnorm(length(xvec),mean=y.det,sd=y.error)
    fit<-lm(ysim~xvec)
    pval[i]<-summary(fit)$coefficients['xvec','Pr(>|t|)']
  }
  power.b[j]<-sum(pval<0.05)/nsim
}
```

Note that this is basically the same function as before, but with the original loop (over simulations) nested within a loop over slope values. Thus, we needed to create a vector of slope values to evaluate (bvec) and a storage vector to hold the results (power.b). The power is computed for the first value of slope in bvec (as before) and the result is stored in the first position of the storage vector power.b. Each time through the outer loop, a new value of power is computed for the next value of slope. The result is a vector of power values for increasing values of slope.

Let's plot the result and add a vertical line to show the slope of our original data set:

```
plot(bvec,power.b,type='l',xlab='Effect size',ylab='Power')
abline(v=coef(fit1)[2],lty=2,col='red')
```

What is the power for a slope of say .002?

We can do the same thing for a gradient in sample sizes, as follows. Note, we need to define b back

to its original value because it was changed in the previous simulation.

```
b<-coef(fit1)[2]
nsim<-1000
pval<-numeric(nsim)
nvec<-seq(10,50)
power.n<-numeric(length(nvec))
for(j in 1:length(nvec)){
  xvec<-seq(0,100,length.out=nvec[j])
  for(i in 1:nsim){
    y.det<-a + b*xvec
    ysim<-rnorm(length(xvec),mean=y.det,sd=y.error)
    fit<-lm(ysim~xvec)
    pval[i]<-summary(fit)$coefficients['xvec','Pr(>|t|)']
  }
  power.n[j]<-sum(pval<0.05)/nsim
}
```

And plot the result as before:

```
plot(nvec,power.n,type='l',xlab='Sample size',ylab='Power')
abline(v=length(birdhab$ls),lty=2,col='red')
```

How much power is lost if we reduce the sample size from 30 to 20?

We could repeat this process for other parameters such as the error component of the model, but you get the idea. While we can do these power analysis simulations for one parameter at a time, it might be more interesting to vary combinations of parameters, say of slope and sample size, using yet another loop, saving the results in a matrix, and using contour() or persp() to plot the results. Try the following:

```
nsim<-1000
pval<-numeric(nsim)
bvec<-seq(-.01,.01,by=0.001)
nvec<-seq(10,50)
power.bn<-matrix(nrow=length(bvec),ncol=length(nvec))
for(k in 1:length(bvec)){
  b<-bvec[k]
  for(j in 1:length(nvec)){
    xvec<-seq(0,100,length.out=nvec[j])
    for(i in 1:nsim){
      y.det<-a + b*xvec
      ysim<-rnorm(length(xvec),mean=y.det,sd=y.error)
      fit<-lm(ysim~xvec)
      pval[i]<-summary(fit)$coefficients['xvec','Pr(>|t|)']
    }
```

```
    power.bn[k,j]<-sum(pval<0.05)/nsim
  }
}
```

Note, the only difference in this code is that we added a third outer loop and created a matrix to store the results, since we have a power result for each combination of slope and sample size.

Let's plot the result using the contour() function, as follows:

```
contour(x=bvec,y=nvec,z=power.bn)
```

Let's try a perspective plot using the persp() function, as follows:

```
persp(x=bvec,y=nvec,z=power.bn,col='lightblue',theta=30,phi=30,expand=.75, ticktype='detailed')
```

Or, if your really ambitious, try the code below to add a color gradient to the surface:

```
jet.colors<-colorRampPalette(c("blue","green"))
nbcol<-100
color<-jet.colors(nbcol)
nrz<-nrow(power.bn)
ncz<-ncol(power.bn)
zfacet<-power.bn[-1,-1] + power.bn[-1,-ncz] + power.bn[-nrz,-1] + power.bn[-nrz,-ncz]
facetcol<-cut(zfacet,nbcol)
persp(x=bvec,y=nvec,z=power.bn,col=color[facetcol],theta=30,phi=30,expand=.75,
    ticktype='detailed')
```

What does the power surface reveal about the relationship between slope and sample size? If you wanted say a power of >0.8 to detect a slope of b=.002, how large would your sample size need to be?

As you can see, stochastic simulation is an extremely powerful tool for examining power, even in this simple linear regression example where canned approaches exist. For more complex models, the coding is more complex, but the process is the same. The same basic tools learned in this example can be extended to more complex situations.

4. Simulating dynamic processes – population matrix model

The previous example dealt with a *static* environmental process; the data represented a snapshot of the environmental system. However, environmental systems are *dynamic* – they change over time. Therefore, many problems require a dynamic modeling approach. Dynamic models are models that describe how environmental processes drive populations (and other properties of ecosystems) to change over time. Dynamic models are a vast and increasingly complex subject and therefore one that we cannot hope to cover in this lab exercise. Here, we will simply introduce you to dynamic models by constructing a very simple population transition matrix model. Knowing how to simulate dynamic models is important because fitting dynamic models to data is so tricky that it is essential to

models to simulated data to confirm that the methods work.

Let's build a dynamic population model for a single closed population, say, a local population of marbled salamanders without immigration or emigration. We will represent four age classes: juvenile, subadult1, subadult2, and adult, corresponding to year 0, 1, 2, and 3+. To control the key population vital rates, fecundity (number of young produced per breeding individual per timestep) and survival (probability of surviving between timesteps), we will employ a Leslie transition matrix, which records the fecundity and survival rates for each age class in the population. We will not review the structure of a Leslie matrix here, so take some time to review the topic if you need to.

For this model, we need to build some functions. Let's start with a function to calculate fecundity:

```
calc.fec<-function(fec.mean=10,fec.sd=10){
    result<-rnorm(1,mean=fec.mean,sd=fec.sd)
    if(result<0) result<-0
    return(result)
}
```

This function has two arguments, fec.mean and fec.sd, which have default values assigned. These arguments are passed to the rnorm() function to supply the value for the mean and standard deviation (sd). We draw a single random normal deviate from a normal distribution with mean=fec.mean (10 by default) and sd=fec.sd (10 by default) and store the result in the object called 'result'. Because rnorm() can return negative values, we check the result and reset the value to 0 if it is negative. The function returns the final result.

Let's build a similar function to calculate adult survival:

```
calc.adult.surv<-function(adult.surv.mean=0.61,adult.surv.sd=0.05){
    result<-rnorm(1,mean=adult.surv.mean,sd=adult.surv.sd)
    if(result<0) result<-0
    return(result)
}
```

And another one to calculate juvenile survival:

```
calc.juv.surv<-function(juv.surv.mean=0.10,juv.surv.sd=0.03){
    result<-rnorm(1,mean=juv.surv.mean,sd=juv.surv.sd)
    if(result<0) result<-0
    return(result)
}
```

Lastly, we need a function to build a transition matrix from the supplied fecundity and survival rates:

```
build.transition.matrix<-function(fm=10,fs=10,asm=0.61,ass=0.05,jsm=0.1,jss=0.03){
    t<-matrix(0,nrow=4,ncol=4)
    t[1,4]<-calc.fec(fec.mean=fm,fec.sd=fs)
```

```
    t[4,4]<-calc.adult.surv(adult.surv.mean=asm,adult.surv.sd=ass)
    t[4,3]<-1
    t[3,2]<-1
    t[2,1]<-calc.juv.surv(juv.surv.mean=jsm,juv.surv.sd=jss)
    return(t)
}
```

This function is a bit tricky. It has arguments for each of the arguments of the fecundity and survival functions, which we pass to those functions called from within this function. For example, we name the argument 'fm', give it a default value of 10, and then assign it to the fec.mean argument of the calc.fec() function. This will pass the value of fm to the mean of the rnorm() function inside the calc.fec() function. Inside this function, we first create a 4x4 matrix filled with 0's. Then we assign the calculated value of fecundity to the 1st row, 4th column, which will be the fecundity rate for the 4th age class (adults). Then we assign the calculated adult survival rate to the 4th row, 4th column, and so on. Note that the two assignments of 1 are for the survival rates of the two subadult age classes. This is because our survival rate for the juvenile age class is really for the entire period from juvenile to adult, which takes 3 years generally. To keep the model simple and accommodate a 1 year timestep, we simply added two intermediate age classes and just "pass" individuals through those stages. For our purposes, it is not critical to understand this further.

To see how this works, let's build a trial transition matrix by running the function once using the default values:

```
build.transition.matrix()
```

Note that the vital rates are not exactly the default values specified. This is because when we build the transition matrix we draw vital rates from random normal distributions as specified in the corresponding functions. This way, every time we build a transition matrix we will get slightly different vital rates. This is the way we are incorporating stochasticity into the dynamic model.

Because we have named arguments for each of the vital rate parameters, we can easily change the parameters and generate a new transition matrix, for example:

```
build.transition.matrix(fm=10,fs=10,asm=0.41,ass=0.05,jsm=0.1,jss=0.03)
```

Here, we simply changed the adult survival rate from 0.61 (default) to 0.41, but you change any of them if you like.

Now we are ready to simulate population change over time. Let's begin by assigning the simulation length (number of timesteps) and an initial population age class structure:

```
tsteps<-100 #simulation length
pop<-c(40,2,2,20) #initial population age struccture
```

Next, we need to create a matrix to store the results. The matrix needs as many rows as timesteps and as many columns as age classes. Each row will represent the population age structure for a single

timestep. And we can name the columns for convenience, but this is not necessary:

```
output<-matrix(NA,nrow=tsteps,ncol=4) #create object to store results
colnames(output)<-c('juv','sub1','sub2','adult')
```

Next, we loop through the timesteps, each time we construct a new transition matrix and then multiply it by the population vector to create a new population vector:

```
for(i in 1:tsteps){
    t<-build.transition.matrix() #build random transition matrix
    pop<- t %*% pop #multiply transition matrix by population vector
    pop[1][pop[1]>300]<-300 #impose ceiling on year 1 cohort size
    output[i,]<-pop # store result
}
```

Here, we used the %*% to indicate matrix multiplication. In addition, we implemented a ceiling density dependence on juvenile cohort size. If the cohort exceeded 300 individuals, we chopped it back to 300. This is perhaps too simplistic a form of density dependence (and thus probably most unrealistic), but it will suffice for our example.

Let's plot the results using matplot() to plot the trajectories for all four age classes (columns of the output object):

```
matplot(1:tsteps,output,type='l',lty=1,col=5:1,main='Population simulation',
  xlab='Time step',ylab='Count')
```

The plot shows the trajectory of population change by age class for a single simulation. This is great, but it represents a single trajectory of a stochastic population model. Because of the stochastic components of the model, we would expect the trajectory to vary each time we run the model. We could simply run the above code again, and again, but that would be extremely tedious. Instead, let's create a function that contains everything we've done so that we will only need to call the function to generate a new simulation.

```
popsim<-function(tsteps=100,pop=c(40,2,2,20),ceiling=300,
  fm=10,fs=10,asm=0.61,ass=0.05,jsm=0.1,jss=0.03){
  output<-matrix(NA,nrow=tsteps,ncol=4)
  colnames(output)<-c('juv','sub1','sub2','adult')
  for(i in 1:tsteps){
     t<-build.transition.matrix(fm=fm,fs=fs,asm=asm,ass=ass,jsm=jsm,jss=jss)
     pop<- t %*% pop
     pop[1][pop[1]>ceiling]<-ceiling
     output[i,]<-pop
  }
  total<-apply(output,1,sum)
  output<-cbind(output,total)
  matplot(1:tsteps,output,type='l',lty=1,col=5:1,
```

```
    main='Population simulation',xlab='Time step',ylab='Count')
  legend('topright',inset=c(.01,.01),legend=colnames(output),lty=1,col=5:1)
}
```

Here, we named arguments for all the things we might want to quickly change in the simulation. So we have an argument for the number of timesteps (tsteps), the initial population structure (pop), ceiling level for the density dependence (ceiling), and all the other arguments for the fecundity and survival functions. Otherwise, we added a step to compute the total population size from the population age class structure. To do this, we used the apply() function to sum each row of the output object and then bound the result back to the output object. Lastly, we added a legend to the plot. Obviously, there are many other things we could do in this function to enhance the output, but this should suffice to demonstrate the flexibility we have in writing our own functions.

Now that we have a bonafide function, let's use it:

```
popsim()
```

Now let's see how easy it is to change any of the parameters and instantly see the result. For example, let's change the mean adult survival rate and the mean juvenile survival rate:

```
popsim(asm=0.4,jsm=0.05)
```

You can experiment with changing other parameters to see how the population behaves under different scenarios.

5. Final Exercise

The purpose of the following final exercise is to give you additional experience working with stochastic simulation models, but also building on the skills learned throughout this course. You will be assigned by the instructor to a team to complete one of the following exercises.

5.1. Power analysis for one-way analysis of variance

This exercise involves conducting a power analysis for a *one-way analysis variance* (ANOVA) for a real-world experiment involving testing the resistance of several brands of climbing ropes to several types of hand-saw blades commonly used for tree pruning.

Background.–The data set comes from a study by Dr. Brian Kane (Umass) in which he was interested in evaluating the differential resistance of various climbing rope types to accidental cutting by different hand-saw blades. The motivation behind the study was as follows. A professional tree pruner was cutting through a branch when his blade accidentally struck his climbing rope and severed it, causing him to fall to his death. A civil suit was filed against the rope manufacturer, who promptly ask Dr. Kane to investigate the problem. Dr. Kane set up the following controlled experiment. He designed a two-way factorial experiment with 6 different rope types and 4 different blade types (institutional standards) as the independent factors. In a replicated experiment involving 5 replicates of each combination of rope type and blade type (i.e., factorial design), he placed a fixed length of rope under tension, approximating the load of an average human body, and from a fixed height and angle let the blade fall and strike the rope in a manner that approximated the natural field conditions under which this might happen. For each trial, he measured several response variables, including the percent of the rope cut by the blade (p.cut), the percent of the rope strength lost as a result of the blade-induced damage, and several other measures of rope damage. For the purposes of this exercise, we will be concerned only with the main effect of rope type (i.e., we will ignore blade type) on the percent of rope cut (p.cut); in other words, a simple one-way ANOVA. But note that the procedures developed below could easily be extended to a two-way ANOVA, or even more complicated designs.

Dr. Kane wants to use the results of this study to help inform the design of additional experiments, perhaps to examine additional rope types, rope tension levels, and/or other rope conditions. His primary concern is to ensure that he designs an experiment that has adequate statistical power to detect an effect if in fact there is one. In addition, given that the subsequent experiments might involve slightly different treatments (e.g., additional rope types), he wants to entertain the possibility that the standard error among replicates might vary from what he observed in the first experiment. In other words, he would like to understand the statistical power of experiments in relation to varying sample size and error rates.

Objectives.–Your objective is to design a simulation, mirroring the conditions of Dr. Kane's experiment, that estimates statistical power to detect effect sizes equal to those observed in the original experiment for a reasonable range of sample sizes and error rates. Document all steps of the analysis and graphically present the results of your simulation.

Suggested steps.–Here is a suggested list of steps to take and a few key hints that you might need to

complete the exercise.

1. Set up your R work session and read in the raw data ('rope.csv')

2. Plot the data. Note, here, we are interested only in the factor rope.type and the response variable p.cut (percent of rope cut). Hint: try box plots by rope type.

3. Compute a one-way analysis of variance to test for differences in p.cut among rope types. Because ANOVA may not be familiar to you, I will briefly describe it here. Very simply, ANOVA in the context of this study involves determining whether the variability in response (p.cut) *among* treatments (rope.type) is significantly greater than the variability in response *within* treatments. To test whether this is so, the ratio of among to within variance is constructed as the test statistic and this is compared to one of the standard probability distributions, the F distribution, to determine if it is significantly greater than expected given the number of treatments and number of replicates. Here is how to compute the ANOVA manually.

   First, we need to know how many levels there are to our independent treatment factor, rope.type, and the total number of observations (i.e., sample size):

   ```
   ngroups<-length(levels(rope$rope.type))
   nobs<-nrow(rope)
   ```

   Next, we need to partition the total variance in the response variable into its components: among-group versus within-group, so that we can compute the ratio for our test statistic. Here, the trick is to realize that the total variance is the sum of the among-group variance and within-group variance, so by calculating any two of these components, we automatically know the third. We begin by calculating the "sums of squares" for the entire data set; i.e., the squared deviation of each observation from the overall mean, since this is easiest calculation. This is a measure of total variability in the data set, and we often abbreviate this as SSY:

   ```
   SSY<-sum((rope$p.cut-mean(rope$p.cut))^2)
   ```

   Next, we need to calculate either the "sums of squares within" or "sums of squares among". We will compute the former which is also called the "sums of squares error" and abbreviated SSE to reflect the fact that this is the variation that cannot be explained by our treatments (i.e., the model) and thus represents error. To do this requires some tricky scripting:

   ```
   SSE<-sum(tapply(rope$p.cut, INDEX=rope$rope.type, function(x) sum((x-mean(x))^2)))
   ```

   Here we used the tapply() function, interpreted as follows. Take the variable p.cut (first argument), and for each level of rope.type (second argument) apply the following function. In this case, we actually defined the desired function inside the apply function, but we could have defined this function separately and simply called it here. The function states, for x, which is the variable named in the first argument, p.cut in this case, calculate the following: take the value and subtract the mean and then square the difference, and then sum over all observations. In this case, however, the mean and sum are applied separately for observations in each group, so the

end result is a vector containing the sums of squared differences for each group. We wrap this whole thing in a sum() function to sum the within-group errors across groups. Note, the result is the sums of squares error pooled across groups.

Now we can compute the remaining variance component, "sums of squares among", which is often abbreviated SSA:

```
SSA<-SSY-SSE
```

The extent to which SSE is less than SSY is a reflection of the magnitude of the differences between the means. If SSE is much smaller than SSY, then most of the variation in the response is due to differences among groups (or levels of the independent factor). Another way of looking at this is that as the ratio of SSA to SSE increases, then an increasing amount of the variation is due to differences among groups.

It is tempting to think we are done, but we are not. We need to adjust the sums of squares to reflect the degrees of freedom available given the number of treatments (or levels of the independent factor) and the number of replicates per treatment. In our case, we have a total of 121 observations in all, so the total degrees of freedom are 121-1=120. We lose 1 d.f. because in calculating SSY we had to estimate one parameter from the data in advance, namely the overall mean. Five of the rope types had n=20 replications, so they each had 20-1=19 d.f.., for error because we estimate one parameter from the data for each rope type, namely the treatment means, in calculating their contribution to SSE. One of the rope types had n=21 replications, so it had 21-1=20 d.f. to contribute to the overall SSE. Overall, therefore, the error has 5x19 + 1x20 = 115 d.f.. Lastly, there were six rope types, so there are 6-1=5 d.f. for rope type (SSA).

The mean squares are obtained simply by dividing each sum of squares by its respective degrees of freedom, as follows:

```
MSA<-SSA/(ngroups-1)
MSE<-SSE/(nobs-ngroups)
```

By tradition, we do not calculate the total mean square. The test statistic is the *F*- ratio, defined as the among-group variance divided by the error variance:

```
F<-MSA/MSE
```

The *F*-ratio is used to test the null hypothesis that the treatment means are all the same. If we reject this null hypothesis, we accept the alternative hypothesis that *at least one of the means is significantly different from the others*. The question naturally arises at this point as to whether the observed *F*-ratio is a big number or not. If it is a big number, then we reject the null hypothesis. If it is not a big number, then we fail to reject the null hypothesis. As ever, we decide whether the test statistic is big or small by comparing it to the values from an *F* probability distribution, given the number of degrees of freedom in the numerator and the number of degrees of freedom in the denominator. Specifically, we want to know the Type 1 error rate (*p*-value); i.e., the probability of observing an F-ratio as large as ours given that the null hypothesis is true and

thus the treatment means are the same. To do this, we use the pf() function for cumulative probabilities of the *F* distribution, like this:

```
pval<-1-pf(F,ngroups-1,nobs-ngroups)
```

OK, now that you know how to compute the p-value for the ANOVA manually, here is how you do it the easy way using the built-in aov() function:

```
summary(aov(p.cut~rope.type,data=rope))
```

The summary of the aov object provides the ANOVA table with all the components we calculated above. You will need to be able to extract the *p*-value and the mean square error from the summary object, which you can do as follows:

```
pval<-summary(aov(p.cut~rope.type,data=rope))[[1]][1,5]
y.error<-summary(aov(p.cut~rope.type,data=rope))[[1]][2,3]
```

4. Next, create a single stochastic simulation of the model that mirrors the original study. Hint: there are three key parts to this task:

   (1) Create a *factor* for the independent variable that mirrors the structure of the original study in terms of number of treatment levels and number of replicates per treatment. Hint: assuming that you have defined the number of groups (ngroups) and the number of replicates (nreps), try the following:

   ```
   groups<-factor(rep(1:ngroups,each=nreps))
   ```

   (2) Create a numeric response variable that mirrors the effect size and the residual error of the original data set. Hint: there are two parts to consider: 1) the deterministic part of the model and 2) the error component of the model. For the deterministic part, try the following:

   ```
   grp.means<-tapply(rope$p.cut,INDEX=rope$rope.type,mean)
   y.det<-grp.means[groups]
   ```

   Now add some error to create the final vector. Hint, see the linear regression example. Also, watch out for negative values and decide how to deal with them.

   (3) Lastly, compute the p-value for the test of group differences for the random sample.

5. Next, once you have successfully simulated the data set once and computed the p-value, repeat the process many times and calculate the power of the test.

6. Lastly, once you have successfully calculated the power for a fixed sample size and error, calculate the power for a gradient in sample size (e.g., from 2 to 50 replicates per treatment) and error (e.g., from 0.01 to 0.1) and plot the results. Hint; to do this you will need to embed the above function inside two additional outer loops, one for sample size and one for error.

5.2. Power analysis for linear trend – date of flowering data

This exercise involves conducting a power analysis for a *linear trend* for a hypothetical observational study involving the date of flowering in a spring annual.

Background.–This problem is motivated by the growing awareness of climate change impacts on ecological systems. Among other impacts, it has been documented that recent climate changes, in particular, global warming, has led to measurable changes in plant phenology with unknown consequences for the affected organisms and the organisms directly or indirectly dependent on them. One of the more dramatic documented changes has been in the earlier timing of emergence or flowering of plants due to warmer temperatures. One of the many concerns with earlier flowering is that host-specific dependent species such as pollinators may not be synchronized with the phenological changes in the plants, and both the plant and the dependent species may decline as a result.

For the purposes of this exercise, let's assume that you are a biologist with an environmental agency and you are responsible for establishing a monitoring program to document trends in the timing of flowering of spring annuals to better understand the potential impacts of climate change. You have the good fortune of having a preliminary data set to work with. Specifically, a botonist working with the Trustees of Reservations by the name of Trilly has been monitoring the flowering dates of Trillium () over the past 10 years on one of the Trustees Reservations. Specifically, Trilly established 5 permanent plots distributed randomly across the Reservation and has been surveying them intensively during the spring emergence period each year. The resulting data set includes the Julian date of first flowering on each plot for each year in addition to a number of other environmental variables related to annual climatic conditions.

You are able to use the results of this small-scale study to help inform the design of a monitoring program for the state lands under your jurisdiction, perhaps to monitor a wide range of plant species in a wide range of biophysical settings. Your primary concern is to ensure that your study design has adequate statistical power to detect a trend over the next 10 years if in fact there is one. In addition, you are interested in evaluating the relative tradeoffs between the number sample plots (i.e., sample size) and the magnitude of the trend (i.e., the slope of the trend line), because there is a real cost to each additional sample plot but at the same time you would like to confirm the existence of even a minor trend.. In other words, you would like to understand the statistical power of experiments in relation to varying sample size and slope of the trend line.

Objectives.–Your objective is to design a simulation, mirroring the conditions of Trilly's experiment, that estimates statistical power to detect a trend of varying magnitudes for a reasonable range of sample sizes over a 10-year period. Document all steps of the analysis and graphically present the results of your simulation.

Suggested steps.–Here is a suggested list of steps to take and a few key hints that you might need to complete the exercise.

1. Set up your R work session and read in the raw data ('flower.csv')

2. Plot the data to determine the distribution of Julian dates. In addition, we might suspect that the plots differ in their mean Julian dates, reflecting differences in the biophysical setting among plots. Therefore, it will be useful to plot the distribution of mean Julian dates among plots and determine the probability distribution that best represents it, since you will need this to simulate new data. Hint: plot a histogram of Julian date first, just for kicks, then plot a histogram of the mean Julian date for each plot after subtracting the minimum mean Julian date (and plot it on the probability scale). Try the following:

```
plot.mu<-aggregate(flower$julian,by=list(flower$plot.id),mean)
hist(plot.mu$x-min(plot.mu$x),prob=TRUE)
```

   What probability distribution might describe the observed distribution? Can you estimate the parameters of that distribution? Try adding a curve to the histogram with using the curve() function, for example:

```
curve(dnorm(x,mean=4,sd=2),0,20,add=TRUE)
```

   What about the distribution of initial Julian dates; i.e., the Julian dates of first flowering during the first year of the study (tstep=0). To look at this, we need to extract the dates for the first year of the study. Try the following:

```
init<-flower$julian[flower$tsteps==0]
```

   Now, plot a histogram of this data and then add a curve to the histogram using the dnorm function as above, but this time use the sample mean and standard deviation.

3. Plot the data to evaluate whether a linear trend is appropriate. Note, here you will want to plot the trend separately for each plot, since you are interested in controlling for the differences among plots. Hint: you can simply plot a bivariate scatterplot of time against Julian data and color and symbolize the points for each plot separately. Alternatively, you might try plotting each plot separately using the groupedData() function in the nlme library, as follows:

```
library(nlme)
result<-groupedData(julian~tsteps|plot.id,data=flower)
plot(result)
```

   Does a linear trend seem reasonable for these plots?

4. Next, fit a linear regression model to the data, but account for the differences among plots. Note, there are many issues and many different statistical methods for analyzing trends, which is way beyond the scope of this lab. For example, in the current context, you could fit a separate trend line to each plot and simply look for consistency across plots or average the resulting slopes and intercepts across plots. Alternatively, you could add plot.id as a categorical (factor) fixed effect to the model, making it a multiple linear regression model or an analysis of covariance, which would allow the intercepts to varying among plots. Alternatively, you could use what is known as a "mixed effects" model in which you treat plot.id as a random effect,

essentially accounting for the differences among plots, and let either the slope or intercept or both vary among plots. In any of these models, you could ignore the serial autocorrelation in the errors that occurs when the response in one year can be partially predicted from the response in the previous year or years., or you could try to model the autocorrelation, for which there are several different methods. For practical purposes, you can keep it relatively simple and use the analysis of covariance approach described above, which allows you to focus on the slope of the trend while accounting for differences in the mean among plots in a relatively simple way, which I will refer to here as a "varying intercept model".

First, plot the data and fit a linear regression or trend line to each plot to confirm that a varying intercept model is reasonable. Hint: here you might first plot the data (as above) and then, in a loop, fit a linear trend to each plot and add it to the plot, like this:

```
plot(julian~tsteps,pch=19,col=plot.id,data=flower)
for(i in 1:5){
  fit<-lm(julian~tsteps,subset=plot.id==i,data=flower)
  abline(reg=fit,col=i)
}
```

Next, assuming that you accept the varying intercept model as reasonable, go ahead and fit an analysis of covariance model, as follows:

```
fit<-lm(julian~tsteps+factor(plot.id),data=flower) #try ancova model
summary(fit)
```

What do the results say about the slope of the overall trend (i.e., the estimate for tsteps) and its significance? Also, what is the residual standard error? You will need these estimates for your simulation.

5. Next, create a single stochastic simulation of the model that mirrors the original study. Hint: there are three key parts to this task – the first part is a bit tricky:

   (1) Create a data set that mirrors the structure of the original study in terms of number of plots and number of years. Hint: first you will need to create a vector of random initial Julian dates for the plots based on the distribution you described above, and call this vector y.init. You will also need to create a vector called tsteps that varies from 0 to 10 by 1 to mimic the original data. Lastly, you will need to define the number of plots (nreps), the common slope (slope), and the residual error (sigma). With these objects defined, try the following:

```
y.sim<-matrix(NA,nrow=length(tsteps),ncol=nreps)
y.sim[1,]<-y.init
    for(j in 2:length(tsteps)){
        y.sim[j,]<-sapply(y.init,function(x)
            x+(slope*tsteps[jj])+rnorm(1,mean=0,sd=sigma))
        }
y.sim<-as.data.frame(cbind(tsteps,y.sim))
```

```
y.sim<-reshape(y.sim,direction='long',varying=list(2:ncol(y.sim)),
    timevar='plot.id',v.names='julian')
```

Since this is the core of the simulation, make sure you understand what this is doing. Can you identify the deterministic part of the model and the error component of the model? A bunch of the script is just formatting the simulated data set to look like the original. For example, the reshape command takes the y.sim object in "wide" format and converts it to the "long" format consistent with the original data.

(2) Next, fit the analysis of covariance model, as above, and you may want to plot the data first to make sure it looks right (see previous code).

(3) Lastly, extract the computed p-value for the slope of the trend line, as above.

6. Next, once you have successfully simulated the data set once and computed the p-value, repeat the process many times and calculate the power of the test. Hint: to do this you will have to embed the core function above within an outer loop over the number of simulations.

7. Lastly, once you have successfully calculated the power for a fixed sample size and slope, calculate the power for a gradient in sample size (e.g., from 2 to 40 plots) and slope (e.g., from 0 to 0.5) and plot the results. Hint: to do this you will need to embed the function above in two additional outer loops, one for sample size and one for slope.