Noise, nonlinearity, and model-free prediction

SBM

October 16, 2018

Often in ecology, particularly applied ecology we think about models with considerable process stochasticity. Where does this come from? Most of the time when I ask this, people respond by saying some thing about the weather / climate which is kind of funny since all of the models used to predict the weather and climate are deterministic. So how do we reconcile this?

Unobserved state variables

There are two important ways in which nonrandom things end up being treated as random. The first is as an inevitable (?) consequence of having incomplete data. Take for instance the following figure

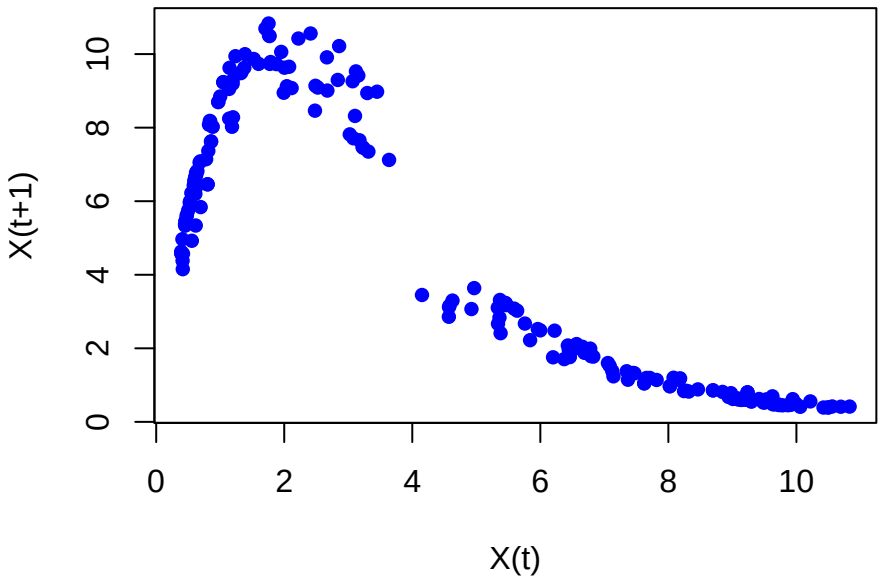

Figure 1: Some mildly 'noisy' data. The blue points represent population sizes observed over 200 generations

This plot looks a bit noisy, though it's pretty good for ecological data. We'd be well justified in fitting a model to it that allowed for some process error.

That said, these data come from a totally deterministic model, albeit one that is 2 dimensional. So the apparent 'stochasticity' arises because we are ignoring the other state variable. Specifically, Figure 1 was generated from

$$(1)\quad \begin{aligned} x_{t+1} &= x_t e^{r - a x_t - b y_t} \\ y_{t+1} &= y_t e^{r - a y_t + b x_t} \end{aligned}$$

with r=2.75, a=0.5 and b=0.07. A fully 2-d version of this function is plotted in Figure 2 below

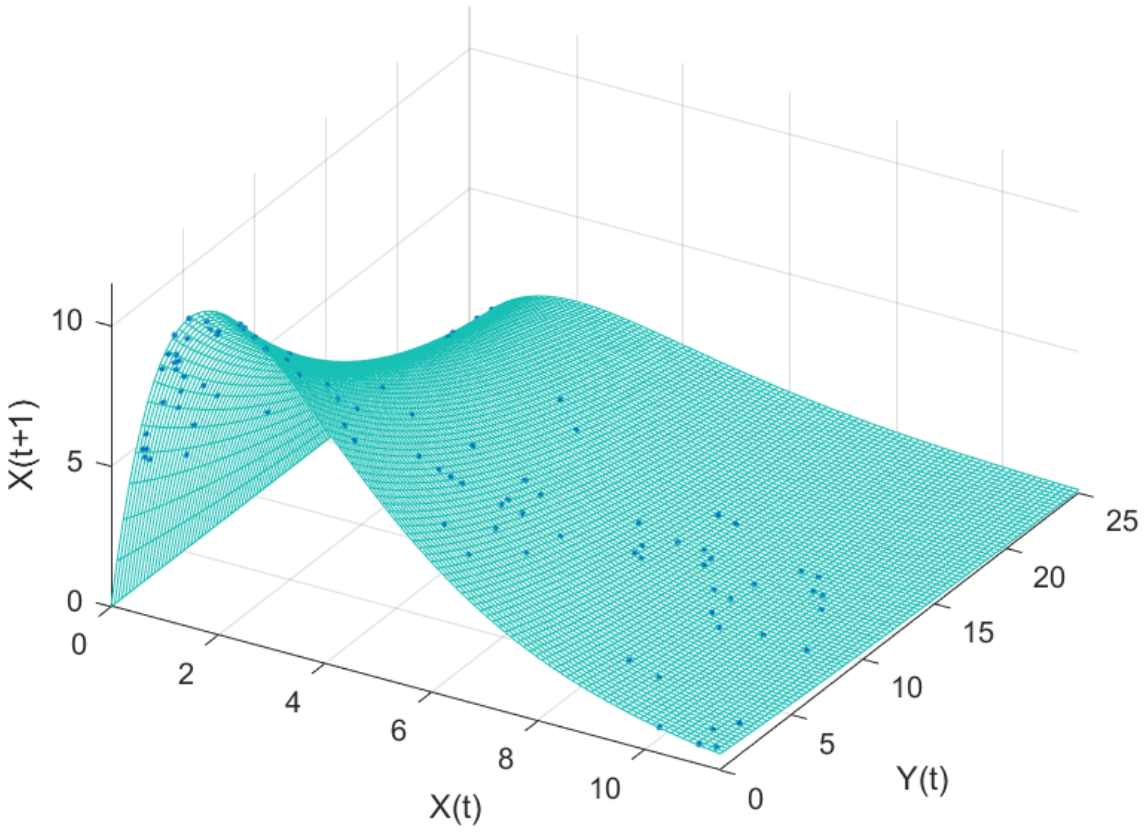

Figure 2: The surface illustrates the deterministic function used to generate the previous figure. The points are generated the same way, but are clearly noiseless when viewed in 2-d

In most ecosystems there are **many** species. Moreover, within each species there are differences among individuals, in e.g. energetic reserves, body size, age, genes, etc. All of these can affect dynamics. Even in a well studied system, we probably keep track of far fewer state variables than required to completely reconstruct what's going on. So one, probably ubiquitous, source of process 'noise' are the unmeasured state variables.

Imprecise initial conditions

And then, there are the things that we treat as random because we just cant do any better. Take, for instance, the classic example of a random event that we were all taught in grade school, tossing a coin. We start with the coin, flip it into the air so that it rotates reasonably fast, and check which side is facing up when the coin eventually comes to rest. Our usual assumption is that the coin is 'fair', i.e. that both sides are equally likely to face up.

But what makes this *random*? The physics of the problem can be made very simple: the coin leaves from a fixed height, x_0 (m) with a given vertical velocity, v_0 (m/s) and gravity draws it back to earth with constant acceleration $-g$ (m/s^2). Since the height at time t is given by $x(t) = x_0 + v_0 t - 1/2gt^2$, the time aloft (i.e. before the height is again x_0) is set by $t = 2v_0/g$. If we neglect air resistance, we can assume that the

angular velocity stays constant through time (at rate ν) so that the angle at time t is $\theta(t) = \theta_0 + \nu t$. Let's say that we started it horizontally with heads up and we'll call this $\theta_0 = 0$. So, if it lands at time $t = 2v_0/g$, the angle will be $\theta = 2\nu v_0/g$. So, where did we ever get the idea that this was random??!

At this point we could try to allow for bouncing etc, but that's really hard and totally unncessary. Let's just pretend that we'll call it 'heads' if it lands with θ between 0 and π and if it lands with $\theta \in [\pi, 2\pi]$ we'll call it 'tails'. Of course, there's no reason for $\theta = 2\nu v_0/g$ to be in $[0, 2\pi]$, so we need to be a little more careful. If we kept on going into the interval $\theta \in [2\pi, 3\pi]$ we'd have 'heads' again and for $\theta \in [3\pi, 4\pi]$ we'd have 'tails.' We can keep going like this, so really what we want to say is that 'heads' corresponds to $\theta \in [2n\pi, (2n+1)\pi]$ where n is any integer, starting from 0. Similarly, 'tails' means that $\theta \in [(2n+1)\pi, 2(n+1)\pi]$. So, let's make a plot of the outcome of our coin toss, as a function of our initial vertical and angular velocities.

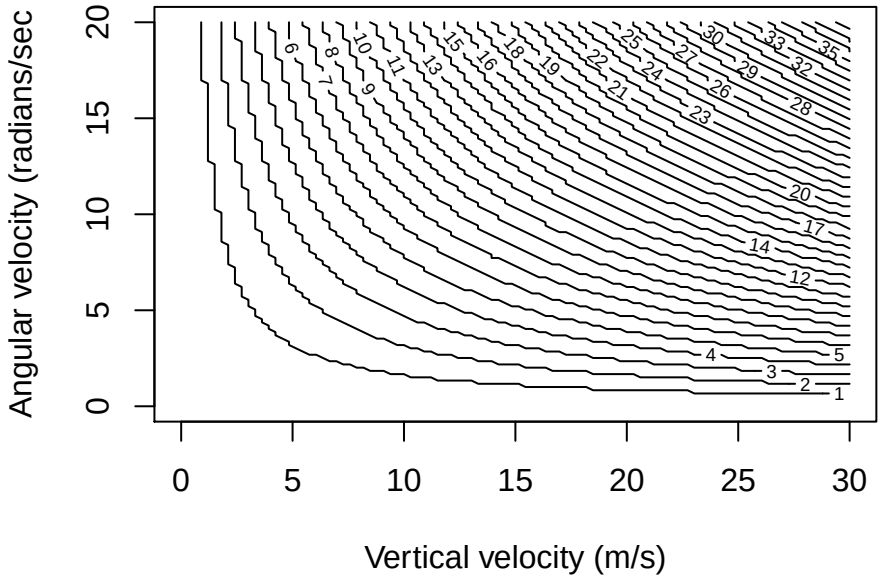

Figure 3: Outcome of coin toss as a function of the initial upward and rotational velocities. The contours separate regions of H and T.

The countours here indicate a switch from heads to tails and vice-versa, starting from heads. That is between the axes and the coutnour labeled "1" we have heads, between 1 and 2, we have tails, etc. Down in the bottom left, where both the rotation and vertical velocity are low, there are big gaps between contours. This agrees with our experience on the playground as kids where if we tossed a coin slow enough we could predict with good accuracy which side it would land on. However, as we increase the vertical and angular velocities (i.e. heading toward the upper right side of the plot), the spacing between successive contours becomes very fine, such that it is hard to know which side will come up. Again, this agrees with our practical experience of coin tossing as children.

So, why do we treat coin tosses as random? If we know the initial and angular velocities with infinite precision, there is no uncertainty. But in practice, our precision is always finite. As a consequence we can alway increase the velocity to a point where the contours are finer than our precision will allow us to differentiate, effectively making it impossible to tell which side will come out 'up'. You can see this effect in Figure 4 below, where the red and blue boxes represent the same degree of uncertainty around two different sets of initial conditions. For the red box at the bottom right, we can be nearly sure that, despite our uncertainty, we will get heads.

But in the blue box in the upper right, the same degree of uncertainty encompases several different bands with nearly equal areas corresponding to heads and tails. So if our uncertainty in initial conditions puts us inthe blue box, we'd have roughly equal chances of heads or tails.

This puts me in mind of times insisting that my playground pals "flip the coin fast enough so it's random." But, really, the 'randomness' in the coin toss - our archtypal example of a random experiment - comes entirely from our lack of information about the initial conditions.

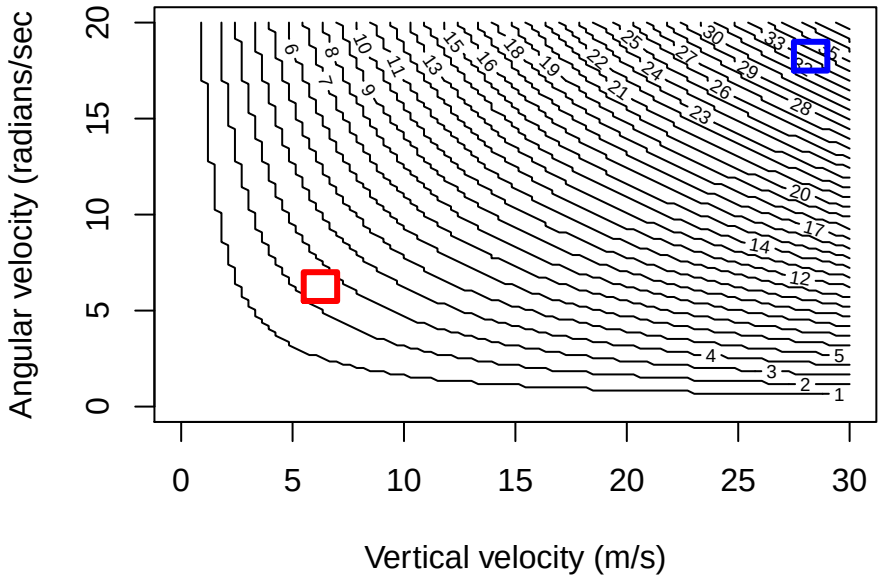

Figure 4: Coin toss again. Here the blue and red boxes represent uncertainty in the initial conditions. The boxes are the same size, representing the same degree of uncertainty, but there are far more possibilities in the blue one.

Let's take a step back. At this point we've said that we might as well treat the outcome of a coin toss as random because we can't know the initial conditions precisely enough... and this is for a situation where the physics is absurdly simple! Imagine what happens when the dynamics are more complex. The contours stop being nice smooth curves and start being fractals, so that the areas where we 'know' the outcome become much more fine grained.

Quasi-Ecological Example

This sort of 'initial value' indeterminacy happens even in the most trivial of ecological models - e.g. the logistic map: $x_{t+1} = rx_t(1 - x_t)$ where x_t is population size at time t and r is the population growth rate. Now, no reasonable biologist believes that this model gives a good apprixomation for real ecological dynamics. But it is simple enough that it allows us to easily see what is going on. For instance, by iterating the logistic map several times to construct the sequence of k-step ahead maps it becomes clear pretty quickly why small amounts of uncertainty lead quickly to indeterminacy. Figure 5 illustrate what happens when we start with a bunch of points close to each other, but not quite equal.

```r
#logistic map
r<-3.95  #max population growth rate. Must be in [0,4]
f<-function(x) {r*x*(1-x)} #logistic map with K=1
T<-11 #number of steps to iterate

x<-array(data=NA,dim=c(T,500))
x[1,]<-rnorm (500)*.01+0.25 #random initial conditions
for (i in 1:(T-1)) {x[i+1,]<-f(x[i,])} #iterate map

#plot some stuff
sa=c(1,2,4,11)
par(mfrow=c(2,2))
for (i in 1:4){hist(x[sa[i],], xlim=c(0,1), col="black",xlab=paste("x(",sa[i]-1,")"),main=paste(sa[i]-1
```

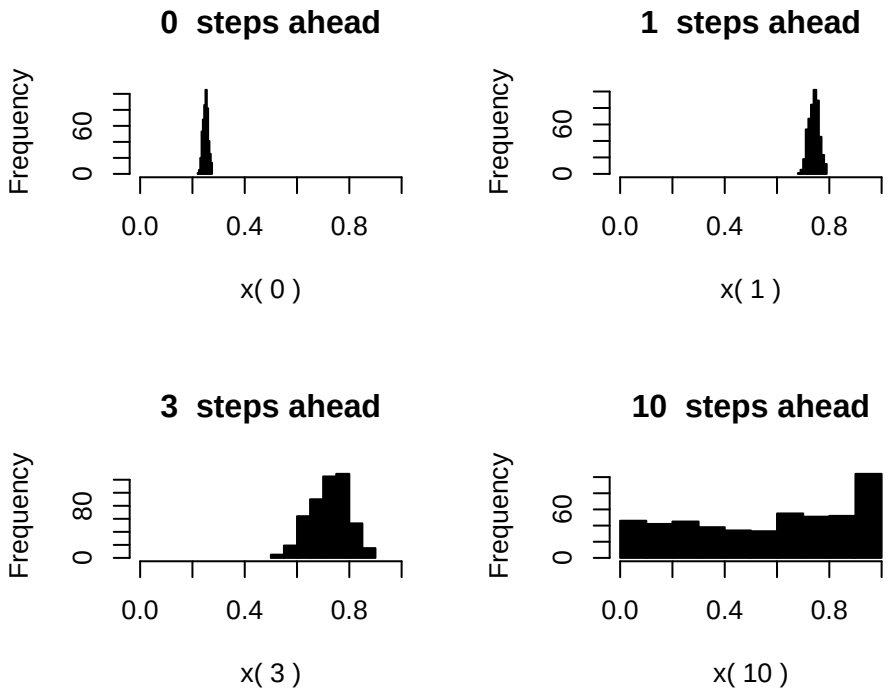

Figure 5: Iterates of the logistic map, several steps ahead. The first panel indicates the starting values which are centered on x=0.25 +/- 0.01. Even through the map is the same for all starting values, they spread out rapidly

Another way of visualizing initial-value sensitivity is to plot the k-step ahead map. If the map is given by $x_{t+1} = f(x_t)$ then the two-step ahead map is given by $x_{t+2} = f(f(x_t))$ and so on. The easiest way to compute this is to define a fairly dense grid for x and apply f to all of those points for k iterations. Figure 6 shows an example of this and how it turns out.

The map gets progressively more wiggly with each setp into the future. Although it is totally deterministic, we can see how a small amount of uncertainty in the initial state would lead to apparent unpredictability by 10 steps out.

One important consequence of this is that if our measurements of a complex system are too far apart in time, it will seem totally unpredictable. On the other hand, if this is the case, an obvious solution is to sample

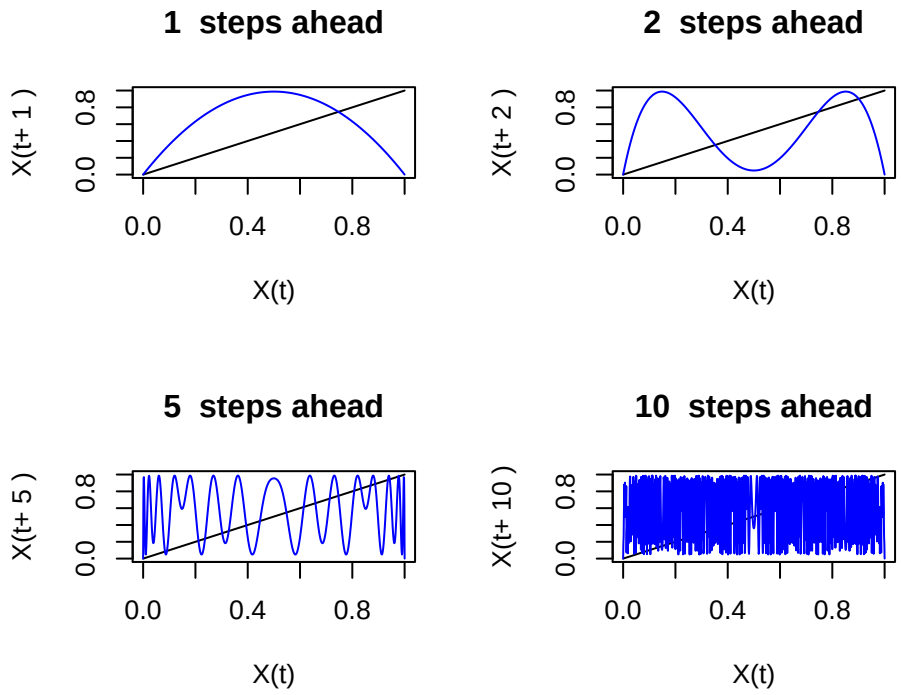

Figure 6: k-step ahead maps for the logistic map. Each curve represents the function turning x(t) into x(t+k)

more frequently!

Actually there's another neat implication of this - if we run this system out long enough into the future, we'll end up at something called the **invariant measure** which is the nonlinear dynamics equivalent of a stationary distribution. Just like the stationary distribution of a Markov process, the logistic maps the invariant measure back into itself. For most models this is not possible to obtain analytically, though it is possible to show that when $r = 4$ the invariant measure for the logistic is $\beta(1/2, 1/2)$. For other values of r this is not so simple - Figure 7 shows the invariant measure for the logistic map with $r = 3.95$. The implication is that if we went out into the world and sampled the abundance of alot of independent populations growing according to the logistic map - this is what we should expect to see.

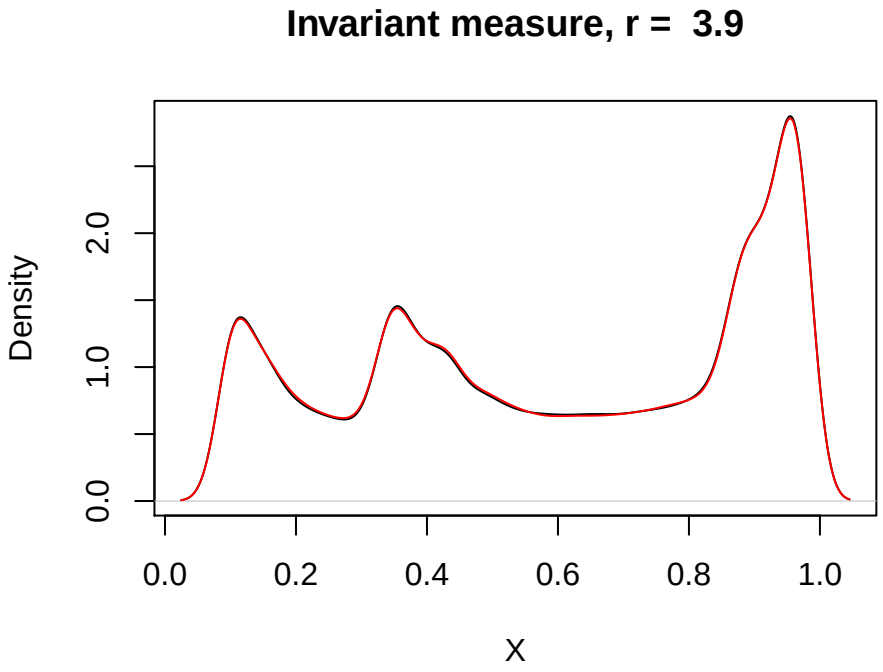

Figure 7: Invariant measure for the logistic map is shown in black. The red line shows the density one step into the future. Kinda crazy!

Initial condition sensitivity in continuous time

This divergence of trajectories due to small changes in initial values is not limited to 1-d discrete time maps. It also happens in continuous time models. Here's an example using a 3-species Rosenzweig-MacArthur model. The dynamics are given by

$\dot{x} = x(1-x) - c_1 \frac{xy}{h_1+x}$

$\dot{y} = c_1 \frac{xy}{h_1+x} - c_2 \frac{yz}{h_2+y} - m_2 y$

$\dot{z} = c_2 \frac{yz}{h_2+y} - m_3 z$

where x represents the density of producers, y is the density of consumers, and z is the density of predators. Ther parameters are set to $c_1 = 5, h_1 = 3, c_2 = 0.1, h_2 = 2, m_2 = 0.4$, and $m_3 = 0.008$. There are a bunch of ways to numerically solve an ODE in r - I used deSolve in the chunk below:

```
#Rosenzweig-Macarthur model via deSolve
times <- seq(0, 250, by = 0.1) #grid of times for output
parameters<-c(c1=5.0, h1=3.0, c2=.1, h2=2.0, m2=0.4,m3=.008);# nice chaotic jughandle
state <- c(X = 0.8,
           Y = 0.1,
           Z = 9)
#model definition
RM_ode<-function(t, state, parameters) {
  with(as.list(c(state, parameters)),{
    # rate of change
    dX <- X*(1-X)-c1*X*Y/(1+h1*X)
    dY <- c1*X*Y/(1+h1*X)-c2*Y*Z/(1+h2*Y)-m2*Y
    dZ <- c2*Y*Z/(1+h2*Y) - m3*Z
    # return the rate of change
    list(c(dX, dY, dZ))
  })   # end with(as.list ...
}

nt=20 #number of trajectories
xx<-array(data=NA,dim=c(length(times),nt))
yy<-array(data=NA,dim=c(length(times),nt))
zz<-array(data=NA,dim=c(length(times),nt))

w<-.01 #standard deviation in initial conditions
for (i in 1:nt){
  state <- c(X = 0.8*(1+w*rnorm(1)),
             Y = 0.1*(1+w*rnorm(1)),
             Z = 9*(1+w*rnorm(1)))
  out <- ode(y = state, times = times, func = RM_ode, parms = parameters)
  #relabel outputs to make plotting easier
  xx[,i]<-out[,2]
  yy[,i]<-out[,3]
  zz[,i]<-out[,4]
}
#generate a color palette so we can see different trajectories
cind<-seq(1:nt)*5
allcolrs<-colors()
colrs<-allcolrs[cind]

par(mfrow=c(2,2))
plot(times, xx[,1],xlab = "time", ylab = "X", main="R-M model",type="l", lty=1,lwd=1)
for (i in 1:nt){lines(times,xx[,i],lty=1,lwd=1,col=colrs[i])}
plot(times, yy[,1],xlab = "time", ylab = "Y",type="l",lty=1,lwd=1)
for (i in 1:nt){lines(times,yy[,i],lty=1,lwd=1,col=colrs[i])}
plot(times, zz[,1],xlab = "time", ylab = "Z",type="l",lty=1,lwd=1)
for (i in 1:nt){lines(times,zz[,i],lty=1,lwd=1,col=colrs[i])}
```

The point is that complex dynamics and unobserved state variables lead to *apparent* indeterminism. Complex dynamics set a hard upper bound on how well we can predict. But things are not quite that bad - we can gain alot of insight from looking at what's going on in the 'phase space' for the system, i.e. by re-plotting contemporaneous points from the time series in the x, y, z space.

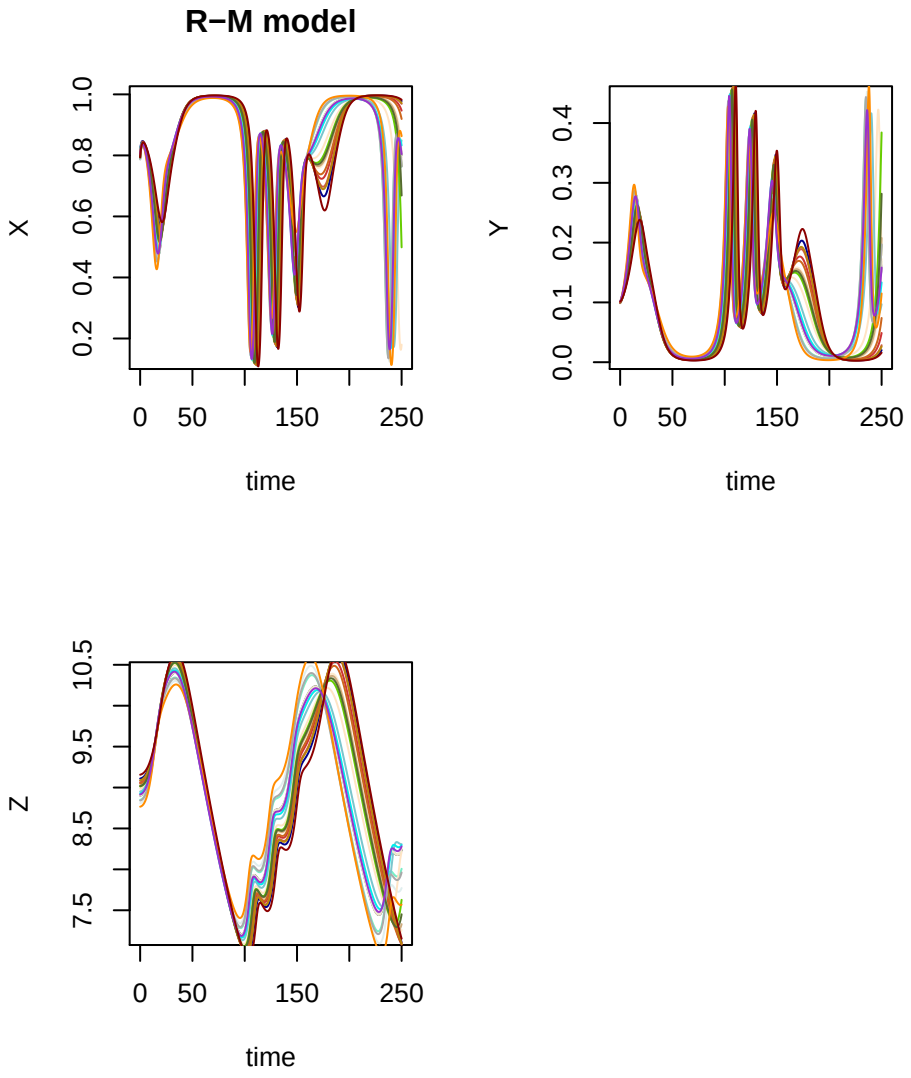

Figure 8: Producer, consumer, predator abundance through time. Each panel shows several solutions to the R-M model. The different colors indicates solutions starting from slightly different initial conditions (all sd = 1% of the mean). Note that trajectories are similar for ~100 days or so and then diverge fairly rapidly.

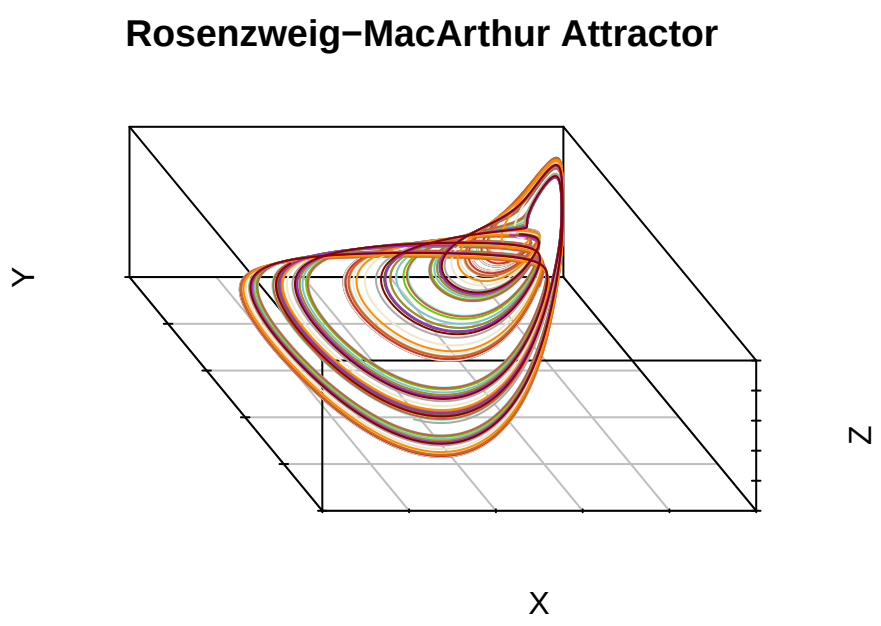

Figure 9: Phase-space for the R-M model. These are the same trajectories as in the previous figure, plotted in x,y,z coordinates.

The shape traced out by these lines is the attractor for the Rosenzweig-MacArthur model. The colors are the same as in the previous plot of the time series. From this we can see that the differences between those time series that were so pronounced in Figure 8 are effectively gone when we look at them in 3-d. Yes, they are not on top of one another, but they clearly fall on the same shape. In fancier mathematical terms they all lie in the same manifold.

This observation that things starting close together end up far apart is one of the hallmarks of nonlinear dynamics and chaos. Measuring the rate of divergence is one of the ways to test whether a given system is chaotic and provides us with an estimate of the 'forecast' horizon, i.e. the time over which we may hope to make useful predictions. Let's take two points that are close together at time 0 and track the distance between the trajectories that follow.

Just in case, the distance between trajectories i and j at time t, is calculated as

$$d(t) = \sqrt{[x_i(t) - x_j(t)]^2 + [y_i(t) - y_j(t)]^2 + [z_i(t) - z_j(t)]^2}$$

In Figure 10 the distance is plotted on a log scale and we've fit a linear model, $\ln[d] = \ln[d_0] + \lambda t$. The fit isnt spectacular (or even good, really), but the trend is clearly positive. It turns out that this slope is a reasonable estimate of the **Lyapunov exponent** which characterizes the approximately exponential divergence rate among initially nearby trajectories.

Actually, since this is a 3-d system, there are two other Lyapunov exponents, but we wont get into how to estimate these today. The time horizon over which we can accurately predict the future states of a system is set by the inverse of the dominant Lyapunov exponent. In this case, $\lambda \approx 0.01$ so the prediction horizon is on the order of 100 days or so which looks pretty reasonable when we look at the time series for each state variable.

Of course, there's a theoretical definition for the Lyapunov exponent, too. Let's say we start from point

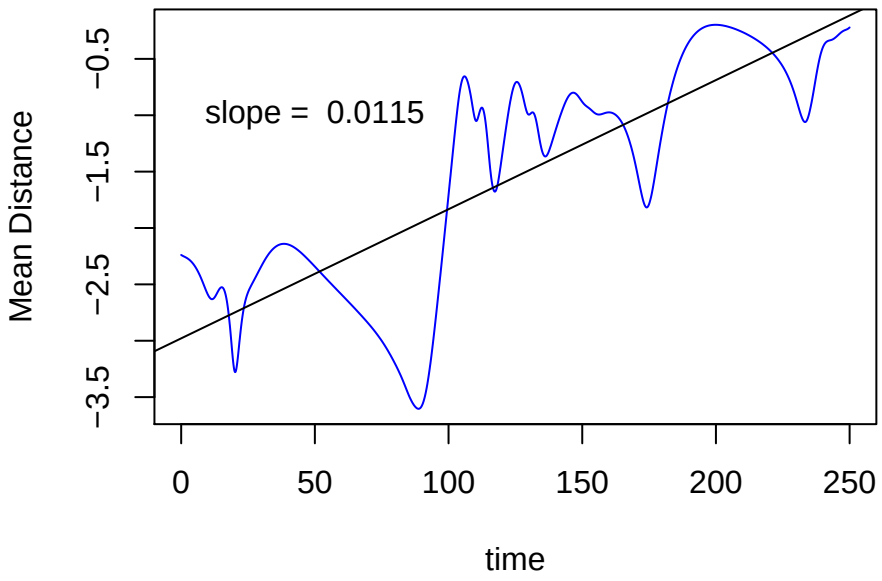

Figure 10: Estimating the divergence rate for the R-M model. The vertical axis is the log of the average distance between points in x,y,z coordinates plotted as a function of time. The line is an OLS regression.

$\mathbf{p} = \{x_0, y_0, z_0\}$ and a nearby point, $\mathbf{q}$. The vector separating $\mathbf{p}$ and $\mathbf{p}$ is $\delta = \mathbf{p} - \mathbf{q}$ and the length of δ is given by $||\delta|| = d_0$. Following our two vectors forward in time, let's denote the solution starting from p as $\mathbf{X}(t, p)$ so that $d(t) = ||\mathbf{X}(t, q) - \mathbf{X}(t, p)||$ denotes the distance between p and q after moving ahead t days. The Lyapunov exponent idea says that the distance grows exponentially in time for a chaotic system and shrinks exponentially when the system is stable. That is $d(t) = ||X(t, q) - X(t, p)|| \approx d_0 e^{\lambda t}$.

Now, obviously this cant be true in general- Since the attractor is a finite size, the maximum distance between trajectories on the attractor is the 'diameter' of the attractor. But, over a short period of time, this exponential model should be a reasonable approximaton. To make it exact, we have to take the limit as $d_0 \to 0$. Re-arranging, we get a definition for the exponent

(2)
$$\lambda = \lim_{t\to\infty, d_0 \to 0} \frac{1}{t} ln \left[\frac{||X(t, q) - X(t, p)||}{d_0} \right]$$

This might look sort familiar, kind of like the definition for a derivative. We'll make more use of this explicitly in our next example.

For most systems it isnt possible to evaluate this analytically - some numerical method is required. In rough outline, this is done by taking the eigenvalues of the matrix obtained by calculating the product of Jacobians for the system evaluated over a large number of time steps. But some additional tricks (QR decomposition) are required for numerical stability. As a consequence, it is alot easier to see how this works for a discrete time system, which we turn to in the next section. But first, a few caveats about interpretting Lyapunov exponents.

A few caveats

First, there are counter-examples in which the Lyapunov exponent for a non-chaotic system is positive and

where an unstable system has a negative exponent. Nevertheless, this is still the most commonly applied way of quantifying chaos in practice.

Second, although we've said that a positive exponent indicates chaos, any **linear** system with a positive real eigenvalue will also have a positive Lyapounov exponent. The difference is that the linear system grows without bound rather than converging to an attractor.

Third, the exponent arrived at can depend on the initial state x_0. For example, if x_0 is an equilibrium, the exponent is 0. So we usually try several starting points, or pick one directly off the attractor.

Fourth, as we said earlier there is one exponent for each dimension of the system. For a generic **q**, the answer will converge to the dominant Lyapunov exponent. But just like in linear systems where the growth along each eigenvector is given by the eigenvalue, there are cardinal directions which correspond to specific exponents. The collection of exponents is referred to as the Lyapunov spectrum. There are some neat things you can do with the complete spectrum, such as characterize the effective dimension of the system / attractor (the Kaplan-Yorke dimension) and the rate at which the system produces entropy (Pesin's formula). But for now, we will have to be content with the maximal Lyapunov exponent.

Lyapunov exponents in discrete time

The Lyapunov exponents for a discrete time system are defined similarly and somewhat easier to handle analytically. Specfically, we can evaluate the Jacobian (matrix of derivatives) directly. For simplicity, let's focus on a 1-d map, which we'll denote by $f(x)$. We can write $X(1,p) = f(p)$. So for $t = 1$ the limit as $d_0 \to 0$ of $\frac{f(q)-f(p)}{d_0} = \frac{f(p+\delta)-f(p)}{d_0}$ is just $f'(p)$. And for $t = 2$, we have $X(2,q) = f(f(p+\delta))$ so that, using the chain rule, the limit of $\frac{f(f(p+\delta))-f(f(p))}{\delta}$ is $f'(f(p))f'(p)$ which is the same as $f'(x_1)f'(x_0)$, since $p = x_0$ and $x_1 = f(x_0)$. From here we can see that at t steps ahead, the derivative will be $\prod_{i=0}^{t-1} f'(x_i)$ which we can use to evaluate the Lyapunov exponent - plugging this in to (2) we get

$$\lambda = \lim_{t\to\infty} \frac{1}{t}\left[\sum_{i=0}^{t-1} ln|f'(x_i)|\right]$$

which now looks alot like taking an average. It is this fact that lets us say that the Lyapunov eponent characterizes the *average divergence rate* of the system, i.e. $\lambda = E\left[ln|f'(x_i)|\right]$ where the expectation is taken with respect to the invariant measure (stationary distribution) of values for x.

Let's put this into practice for the logistic map. With $f(x) = rx(1-x)$ we can easily evaluate the derivative - $f'(x) = r(1-2x)$. To calculate the Lyapunov exponent, we will average this over a large number of iterations of the map.

```
#logistic map
r<-3.95
f<-function(x) {r*x*(1-x)}
df<-function(x) r*(1-2*x) #derivative
T<-60

x[1]<-runif(1) #random starting point
for (i in 1:(T-1)) {x[i+1]<-f(x[i])} #iterate to generate a trajectory

lam=mean(log(abs(df(x)))) #lyapunov exponent
```

Let's compare the direct estimate from this code chunk, $\lambda = 0.512$ to the divergence rate we get doing the regression approach. To start we'll use a sample of 50 time steps. Is that enough?

The points are the observed mean distance between trajectories starting from initial values that were separated by 0.01 or less. Note that the distance asymptotes fairly quickly - remember the growth is only exponential when the distance is much less than the size of the attractor. It is pretty flat from about 9 steps onward. So

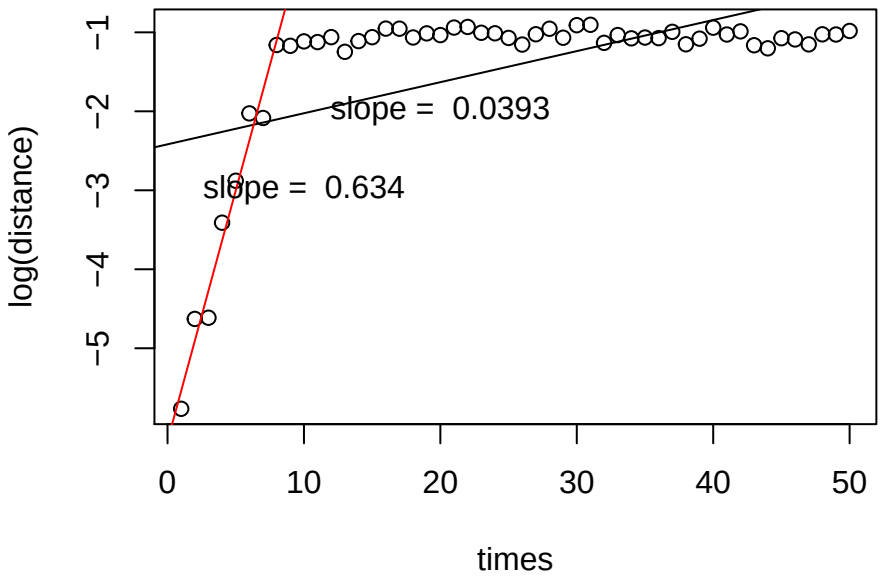

Figure 11: Estimating the divergence rate for the Logistic map. This is the log of the average distance between points plotted as a function of time.

if we naively fit the regression to all 50 (black line), we get a gross underestimate of the Lyapunov exponent. On the other hand, if we get our estimate from only the first 9 points, it is much closer to the analytical answer.

Exercises:

1. Change r (a little) what happens? Try this again for r=1.1, 2, 3. What do you get?
2. Calculate the invariant measure for r=4. Evaluate it for a few other values.
3. Write a shell to compute the Lyapunov exponent for 100 values of r ranging from 3 to 4
4. Do 1-4 again for a different map - e.g. $xe^r(1 + (e^{r/N} - 1)x)^(- N)$ for N=1, 4, and 8. What similarities / differences do you see?

Prediction

One of the more exciting and practical aspects of this stuff is the possibility that we might use it to improve our ability to make predictions for ecosystem dynamics. The fact that the attractor for many systems tends to be a relatively smooth shape (though it may be kind of porous) suggests that we might make use of this for making predictions, even when we dont know the dynamics.

The trick is to leverage the empirically observed attractor for making predictions. We can do this even when dont have any equations for the dynamics, provided that we have enough observations to construct a reasonable picture of the attractor shape (manifold).

Let's imagine that we have the attractor, know the current state, and would like to make a prediction. The easiest way to do this is to find where the current state is on the attractor and look at what nearby trajectories

do as we move into the future. The smoothness suggests that if we have a decent set of neighbors that we ought to be able to do a good job, at least for a while. As we've seen the k-step ahead map can get pretty wiggly, pretty fast, so we shouldnt expect to get too far out into the future.

Let's try to operationalize this. Before we do, I should say that there are r packages to do this sort of thing, rEDM and others. But using them doesnt do a lot to illuminate what's going on. They are definitely good ways to save time on implementation later - but for now, let's do it ourselves so we can see exactly how it works.

The code chunk below simulates the RM model, measures distances between points, and uses the nearest neighbors to guess the future path for the system. One tricky bit when we are trying to forecast from an observed point is that the nearest neighbors are often the points just before or just after the thing we want to forecast. So we need to add in a criterion to exclude those. Of course this isnt a problem when making a prediction from a random point near the attractor (i.e. one that wasnt part of the observed series)

```
#rosenzweig-macarthur - take 2
T=10000; dt=.1; # set the time horizon and step size

#set up for deSolve
times <- seq(0, T, by = dt)
parameters<-c(c=5.0, h=3.0, d=.1, j=2.0, m=0.4,n=.008);# nice chaotic jughandle
state <- c(X = 0.8,
           Y = 0.1,
           Z = 9)
RM_ode<-function(t, state, parameters) {
  with(as.list(c(state, parameters)),{
    # rate of change
    dX <- X*(1-X)-c*X*Y/(1+h*X)
    dY <- c*X*Y/(1+h*X)-d*Y*Z/(1+j*Y)-m*Y
    dZ <- d*Y*Z/(1+j*Y) - n*Z

    # return the rate of change
    list(c(dX, dY, dZ))
  })   # end with(as.list ...
}

#run deSolve
out <- ode(y = state, times = times, func = RM_ode, parms = parameters)

#relabel x,y,z
xx<-out[,2]
yy<-out[,3]
zz<-out[,4]

ts=4025 #a good looking place to start

#point from which to start a prediction
xs<-xx[ts]
ys<-yy[ts]
zs<-zz[ts]

#find some neighbors
nn=6 #number of nearest neighbors
timepenalty<-100*(abs(times-ts*dt)<200)#penalty to exclude points too close in time
distance<-(xs-xx[-ts])^2/var(xx)+(ys-yy[-ts])^2/var(yy)+(zs-zz[-ts])^2/var(zz)+timepenalty
```

```
## Warning in (xs - xx[-ts])^2/var(xx) + (ys - yy[-ts])^2/var(yy) + (zs - zz[-
## ts])^2/var(zz) + : longer object length is not a multiple of shorter object
## length
```

```
dsort<-sort(distance,index.return=TRUE)
nbr<-dsort$ix[1:nn] #indices of nearest neighbors

#ask where the neighbors are for the next 50 steps into the future
xp<-array(data=NA,dim=c(51,nn))
yp<-array(data=NA,dim=c(51,nn))
zp<-array(data=NA,dim=c(51,nn))
for (i in 1:nn){
  xp[,i]=xx[nbr[i]:(nbr[i]+as.vector(50))]
  yp[,i]=yy[nbr[i]:(nbr[i]+as.vector(50))]
  zp[,i]=zz[nbr[i]:(nbr[i]+as.vector(50))]
}


#plot results
plot3D <- scatterplot3d(xx,-yy,zz, type = "l", xlab = "X", ylab = "Y", zlab = "Z", main="Rosenzweig-Mac
plot3D$points(xs,-ys,zs,type="p",col="black",pch=15)
for (i in 1:nn){plot3D$points(xp[,i],-yp[,i],zp[,i],type="l",lwd=2,col="red")}
plot3D$points(xs,-ys,zs,type="p",col="black",pch=15)
```

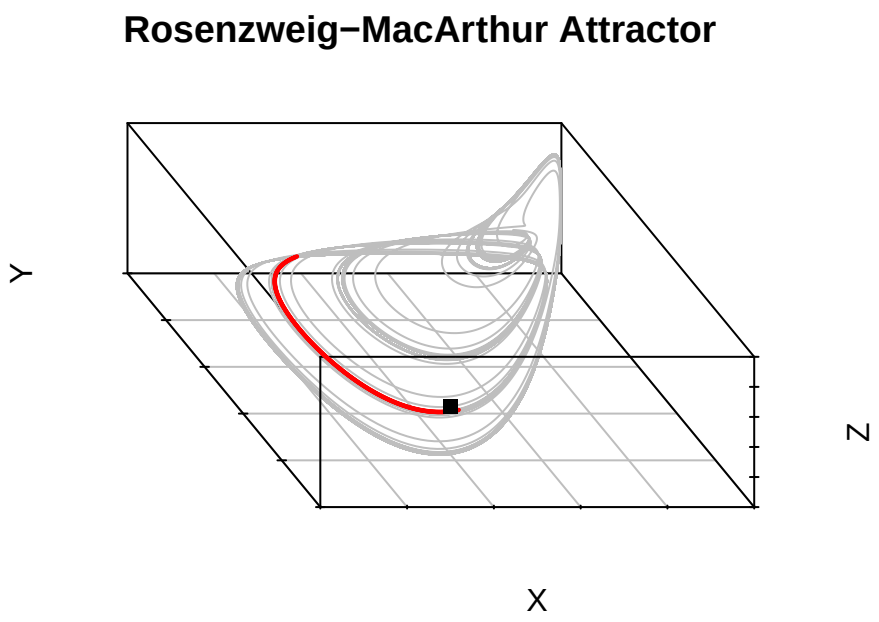

Figure 12: Using the attractor to make a prediction. The box indicates the target point from which we'd like a forecast and the red lines are neighboring trajectories.

The red trajectories emanate from the nearest neighbors of the target point. We can combine these any way we like to obtain predictions into the future. E.g. we could average them, take the variance, etc. Over this

short a horizon, the trajectories are all very similar, as we'd expect, so the variance will be small and the mean a very accurate prediction of the future values of the target. If we went alot farther into the future, the trajectories would diverge, just like they did when we started thinking about Lyapunov exponents.

In this example, we have tons of perfect data and aked for a prediction for a point with close neighbors. One obvious question is how will we do when there are fewer data points and the neighbors are farther apart. Here's an example from the same thing, just with less data and a point that is a little off the attractor

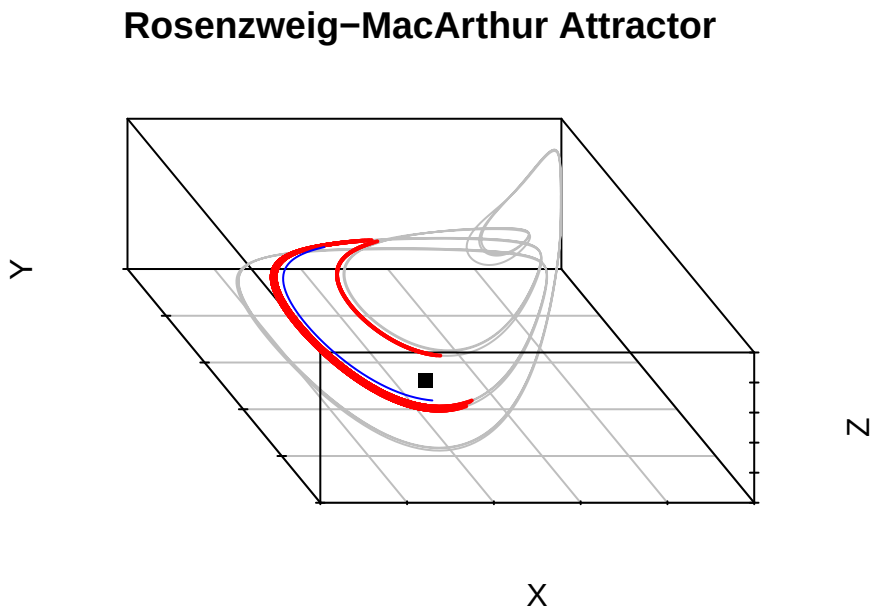

Figure 13: The red lines are the nearest neighbors and the blue line is the prediction based on taking their mean.

Because most of the neighbors are on the lower band, the predicted trajectory obtained by averaging is a bit biased. This suggests that we might want to try a little harder in how we go about using the information from the attractor. Before we get into this, let's think about what's going on a bit more, from a different point of view.

Function Approximation

For any homogeneous deterministic dynamical system, given by a set of ODE's, $\dot{x}_i = f_i(x_1, ...x_S)$ the solution at a particular time in the future can be written (at least theoretically) as a function of the initial conditions, i.e. $x_i(t+\tau) = F_i[x_1(t), ...x_s(t), \tau]$. In what follows, we assume that the lead time, τ is fixed and drop it from the notation.

Our previous prediction algorithm of looking for neighbors and averaging the results τ steps ahead can be thought of as way of approximating $F_i[x_1, ...x_s]$ in the neighborhood of our target point. Note that using the nearest neighbor method, any pair of target points that have the same set of nearest neighbors will have the same prediction. This leads us (implicitly) to a piece-wise constant approximation for F. That is, our estimate of F using the nearest-neighbor averaging approach will be locally flat, and exhibit discrete jumps

as the neighborhoods change. The jumps will be smaller if we use a bigger neighborhood, but the potetial for bias is also bigger, particularly in regions where F is changing rapidly.

With this in mind, we might try a more sophisticated approach to approximating F. There are many options for flexibly inferring functions, ranging from basis function expansions, to local linear models, to neural networks. Probably the easiest of these to implement in r is the local linear regression approach, which is just a special case of *lm*.

To start out, let's do this for the logistic map, because it is easier to see when / how it is working.

```
#logistic map with local linear approximation
r=3.95
F<-function(x) r*x*(1-x)

#simulate data
T=50
x[1]=.1
for (i in 1:(T-1)){x[i+1]=F(x[i])}

#over a grid of values, use a local linear model to approximate F
s=75 #inverse length scale - big number means few neighbors, small number includes more points

#set up a grid for making predicitons
ng=100
xg<-seq(0,1,length=ng)

# use weighted option in lm to do a local-linear regression for each grid point
Y=x[2:(T+1)]
fpred<-0; nnpred<-0
fslope<-0
for (i in 1:ng){
  dist=abs(xg[i]-x[1:T]) #distance to current grid point
  wts=exp(-s*dist^2) #there a lots of choices for weight functions
  X=x[1:T]-xg[i] # we want the regression centered in xg[i]
  locreg<-lm(Y~X,weights=wts) # this line does the local linear model
  fpred[i]<-locreg$coefficients[1] #extract coefficients
  fslope[i]<-locreg$coefficients[2]
  #note that when X=0, x=xg, so all we need for the prediction is the intercept

  #nearest neighbor for comparison
  nn=6 #number of neighbors
  dsort<-sort(dist,index.return=TRUE)
  nbr<-dsort$ix[1:nn]
  nnpred[i]<-mean(Y[nbr]) #average of future values for points near current grid point

}

#plot stuff
plot(x[1:T],x[2:(T+1)],type='p',pch=10,col="blue")
lines(xg,fpred,col="red")
lines(xg,F(xg),col="black")
lines(xg,nnpred,col="green")
```

The results are shown in Figure 14. The red line is the locally weighted linear regression. We could tweak the distance kernel / length scale to make the local regression work better, but this isnt too bad. For comparison, the nearest-neighbor averaging approach is shown in green. This is pretty good where there are plenty of

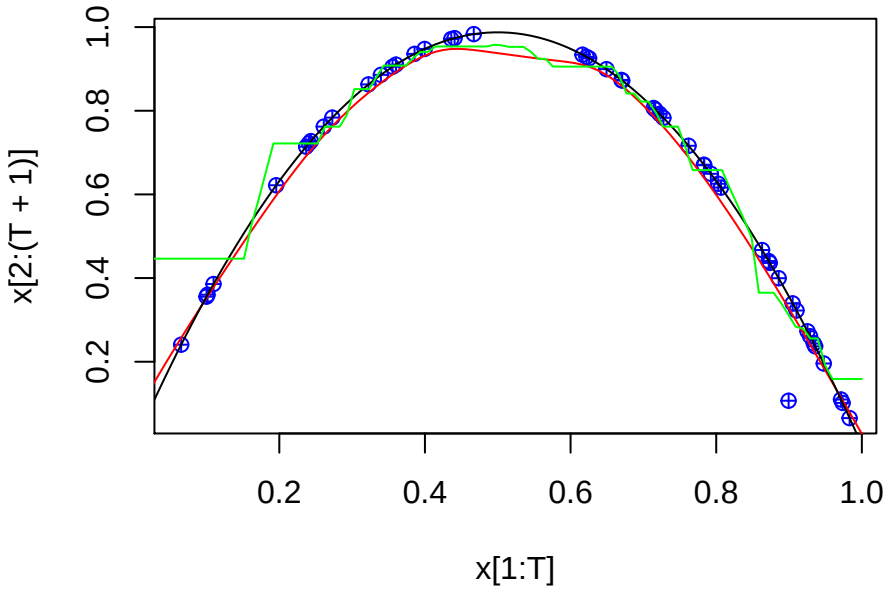

Figure 14: Local linear regression and NN averaging for the logistic map. The blue points are the data generated by iterating the Logistic map (black line). The red line is the locally weighted linear regression. The nearest-neighbor averaging approach is shown in green.

data, but not so great where there are gaps. And the flat spots are decidely at odds with the fact that the target is smooth.

When we want to go multiple steps into the future, there are two (maybe more) choices open to us. The obvious thing to do is to iterate the estimate of f that we just obtained. This is not a bad idea, but approximation errors can blow up pretty fast. The other way to go is to try to estimate the k-step map directly. This can be better when k is relatively small. Obviously, based on the earlier figures, we arent going to pull this off for k=10!

Here's an example of how to do the 2-step ahead map

```
#logistic map with local linear approximation
r=3.95
F<-function(x) r*x*(1-x)

#simulate data
T=50
x[1]=.1
for (i in 1:(T-1)){x[i+1]=F(x[i])}

#over a grid of values, use a local linear model to approximate F
s=500
ng=100
xg<-seq(0,1,length=ng)
Y2=x[3:(T+1)]
Y1=x[2:T]
fpred<-0; nnpred<-0;fslope<-0; f2pred<-0;ffpred<-0 #initialize a bunch of stuff
for (i in 1:ng){
  dist=abs(xg[i]-x[1:T-1])
  wts=exp(-s*dist^2) #weight function
  X=x[1:T-1]-xg[i] # we want the regression centered in xg[i]

  locreg<-lm(Y2~X,weights=wts) # this line does the local linear model for the 2-step map
  f2pred[i]<-locreg$coefficients[1] #extract coefficients

  locreg<-lm(Y1~X,weights=wts) # this line does the local linear model for the 1-step map
  fpred[i]<-locreg$coefficients[1] #extract coefficients
  fslope[i]<-locreg$coefficients[2] #extract slope for iterating
}

# iterate the 1-step ahead map
for (i in 1:ng){
  dist=abs(fpred[i]-xg)
  ind=which(dist==min(dist)) #find grid point closest to 1-step prediction
  ffpred[i]<-fpred[ind]+fslope[i]*(fpred[i]-xg[ind]) #second iteration for 1-step map
}


#plot stuff
plot(x[1:T-1],x[3:(T+1)],type='p',pch=10,col="blue", main="Two step ahead map", xlab="x(t)",ylab="x(t+2)
lines(xg,f2pred,col="purple")
lines(xg,F(F(xg)),col="black")
lines(xg,ffpred,col="red")
```

In Figure 15, the purple line is the 2-step map estimated directly from $x(t+2)$ and the red line is the two-step map obtained by iterating the 1-step map a second time. The red line is distinctly wigglier, but both are

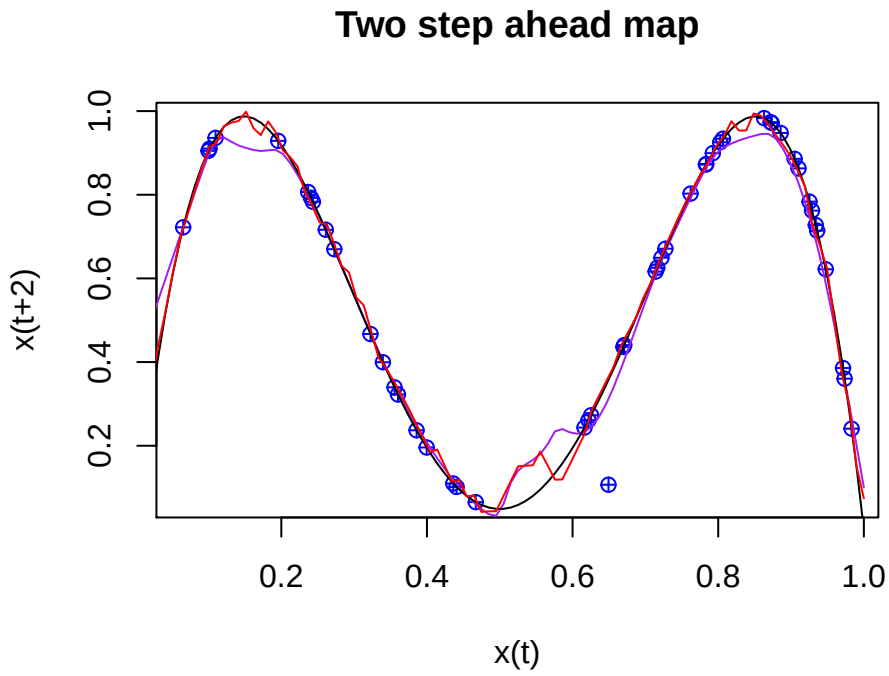

Figure 15: Two ways to get ahead: iterated 1-step map v. direct estimate of the 2-step map. The purple line is the locally weighted linear regression estimate of the 2-step map. The red line is the second iteration of the 1-step map.

pretty close.

Note that we can only use a direct estimate for the k-step ahead map when we have enough data k-steps out. This puts a hard upper bound on how far into the future we can take this approach. Conversely, we can iterate the 1-step map forever, though it will end up pretty bad pretty fast. On the other hand, if we naively tried to estimate the 10-step map directly, we'd most likely get a flat line; any algorithm that allows for process uncertainty will treat the 10-step ahead data as noise. But an iterated approach would not become flat. It seems reasonable to imagine that a hybrid appraoch would be an improvement, though I have not seen this...

When we have more than one input dimension, as in the Rosenzweig-MacArthur example, the local linear model becomes a local multiple regression. Since this is called with the same *lm* in r, not much changes. The only real difference is that we need predictions for each state variable. Typically these are constructed independently, though it is certainly possible that it might be more efficient to have some joint estimation approach.

It is also worth pointing out that there are lots of ways to improve on the function approximation scheme used and many, many papers have done so. My own work is based on using Bayesian Gaussian process regression to infer the map. But the local regression approach (which is directly comparable to Sugihara's s-map) is probably the easiest thing to implement and has several other nice features. Importantly all of the function approximation approaches them make use of the same fundamental idea: that we are trying to estimate a smooth map.

Excercises:

1. Simulate some data from a noisy Ricker map: $x_{t+1} = x_t e^{r - x_t + \epsilon_t}$ where $\epsilon_t \sim N(0, \sigma^2)$
2. Use a local linear model to approximate the 1-step ahead map
3. Use this to make predictions for 2, 3, 4 steps ahead.

Missing state variables

We started this session by remarking that some of what looks like stochasticity in ecological dynamics likely results from missing state variables. And so far, we havent done much to fix that problem. In a static regression context, missing variables are pretty tricky to deal with. But in a dynamical systems context, we can actually make a fair bit of progress because the missing variables and the observed variables are all part of the same attractor. The crucial idea here is Takens(1981) theorem of time-delay embedding which shows that we can, generally speaking, compensate for missing state variables by using lags of the observed variables.

There are lots of descriptions of time delay embedding, most of which are totally impenetrable to readers who - like me - never had a course in Topology. It is usually stated something like this:

Suppose that we are considering an m dimensional dynamical system, $\boldsymbol{x} = \{x_1, ..., x_m\}$ that converges to an attractor, A, whose dimension is $d < m$. And from that system we have a time series of scalar observations, $\{h(\boldsymbol{x}_t)\}$ for $t = 0...T$. Then the collection of E dimensional lagged coordinate vectors of h are an ***embedding*** *of A provided that $E > 2d$.*

It is probably worthwhile trying to define these terms and build up some intuition for why this might work. We have already met the 'attractor' for a system, but haven't really shown that the dimension of the attractor is less than the dimension of the system. If you think about the attractor for the Rosenzweig-MacArthur model, it kind of looks like an upside down teacup. As such, it is an object that clearly lives in a 3 dimensional space. But the walls of the cup and the handle are infinitessimally thin, so that the attractor is alot closer to 2 dimensions. (If you think about a hollow ball, any point on the surface of the ball can be described by 2 coordinates, even though the ball is a 3-d object).

The second mysterious bit is the 'observation function', h. The idea here is that although the dynamics are described in terms of $\mathbf{x}$, we dont need to know exactly what the true 'state' variables of the system are, just that our observations are some function of them. Alot of the time we end up thinking about h as representing one coordinate axis, though there isnt a need to do so. For instance, when we have data on species abundances, we tend to think of abundance as one of the state variables, even though total abundance is really an aggregation over many different kinds of individuals (e.g. old / young, big / small, sick / well, etc).

The third mystery bit, if you've never seen it before, is what an 'E-dimensional lagged coordinate vector' is. Since we dont have data on all of the the original axes, we make synthetic axes out of values of h from distinct time points. So rather than the state being represented by the vector $\{x_t, y_t, z_t\}$ instead we have the vector $\mathbf{h}_t = \{h(\mathbf{x}_{t-\tau}), ..., h(\mathbf{x}_{t-(E+1)\tau})\}$. Here, τ is the time lag, which needs to be chosen big enough to 'unfold' the attractor, but not too big that things become noisy. To make the need for unfolding a bit clearer, recognize that if $\tau = 0$ then all points lie on the $1:1$ line. If τ is just a little bigger than 0, the map we get will still be pretty close to a straight line. The usual prescription for setting τ is to set it equal to the first zero of the autocorrelation function for the data, i.e. make τ big enough that the components of the vector are nearly staistically independent. In ecological practice, where many data sets are short and consist of annual observations, it is pretty typical to just set $\tau = 1$.

The other term, E, is referred to as the embedding dimension - the number of lags needed to get to a 1:1 mapping between the original attractor and its delay coordinate reconstruction. To see what this means, imagine taking a wire and wrapping it around your hand without overlapping itself. Now imagine taking it off your hand and looking at its shadow. The shadow would (most likely) include several places where the wire seems to cross. But you know that when you look at it in 3-d it never does. This is particularly relevant for making predictions; any point where trajectories cross means you dont know which one to follow and leads to apparent indeterminism. The idea with E, like τ is to make it just big enough. Big enough to break up the apparent crossings without being so big that things become noisy again.

Remember that we are going to use this to estimate a function - the bigger E is the more inputs we have, which means the more data we need to get an estimate. This is an important point - the embedding dimension is, as a practical matter, set by the length of the time series. One estimate is that the maximum recoverable E scales as $\sqrt{T}$. So for an ecological time series - which are typically less than 50 years, we should expect to see effective values for E less than 7 - which is more or less exactly what appears in the literature.

The last mystery bit is what it means for the collection of lagged coordinate vectors to 'embed' the attractor. That's where the topology comes in - approximately, it means that the shape traced out in the synthetic coordinate space is a one-to-one and invertible transformation of the original attractor. The practical upshot of this is that anything we could do with the original attractor we can also do with the attractor reconstructed from lags.

Let's take a few slides to make this more intuitive. We will start by visualizing the Rosenzweig-MacArthur attractor reconstruction and then do some delay-coordinate forecasting with a 2-d discrete time model - one where it is easy to see what the map ought to look like.

```
#rosenzweig-macarthur - embedding
times <- seq(0, 2500, by = 0.1)
lt=length(times)
parameters<-c(c1=5.0, h1=3.0, c2=.1, h2=2.0, m2=0.4,m3=.008);# nice chaotic jughandle
state <- c(X = 0.8,
           Y = 0.1,
           Z = 9)
RM_ode<-function(t, state, parameters) {
  with(as.list(c(state, parameters)),{
    # rate of change
    dX <- X*(1-X)-c1*X*Y/(1+h1*X)
    dY <- c1*X*Y/(1+h1*X)-c2*Y*Z/(1+h2*Y)-m2*Y
    dZ <- c2*Y*Z/(1+h2*Y) - m3*Z
```

```
    # return the rate of change
    list(c(dX, dY, dZ))
  })    # end with(as.list ...
}
out <- ode(y = state, times = times, func = RM_ode, parms = parameters)

#relabel data
xx<-out[,2]
yy<-out[,3]
zz<-out[,4]

#different choises for the lag
tau=c(2, 15, 30,90,150)

par(mfrow=c(2,2))
an=130;ys=2;

#original attractor
plot3D <- scatterplot3d(xx,-yy,zz, type = "l", xlab = "X", ylab = "Y", zlab = "Z", main="Original Attra

#delay-corrdinate reconstructions
for (i in 1:3){
plot3Dx <- scatterplot3d(xx[1:(lt-2*tau[i])],xx[(tau[i]+1):(lt-tau[i])],xx[(1+2*tau[i]):lt], type = "l"
}
```

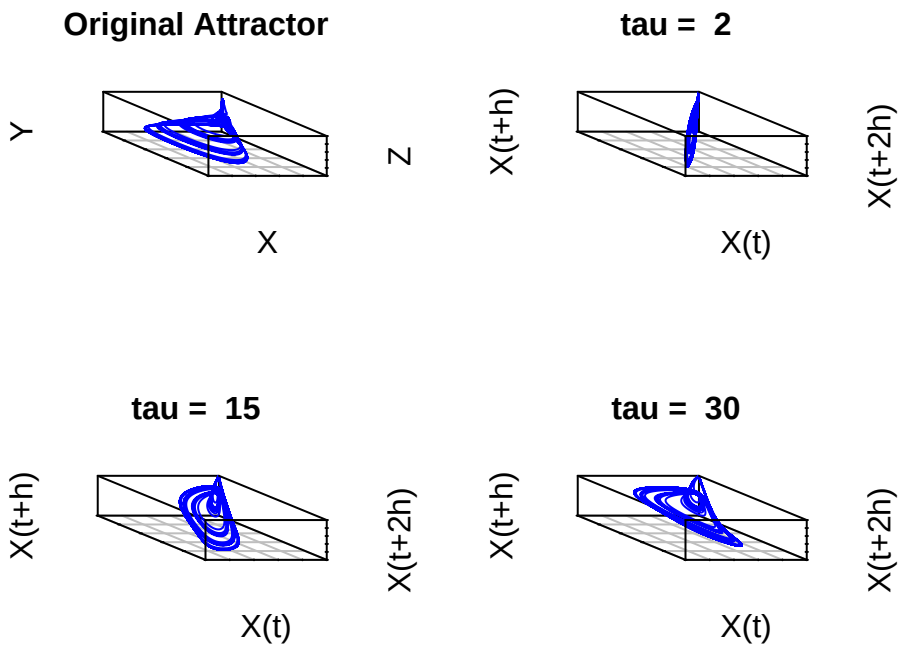

Figure 16: Attractor for the RM model and several representations with lag-coordinates. In each lagged coordinate panel, lags of the producer, x, were used to reconstruct the attractor.

From the plots in Figure 16 we can see that when τ is too small the reconstructed attractor is squished along the diagonal. As we increase τ the paths become more distinct and we can more easily reconstruct relevant neighborhoods. You might want to try this with different values for τ and maybe try reconstructions with other state variables.

Now lets try to use this in a 2-d discrete time model, where we can more easily see what it's doing and whether it is working.

```
x=1;y=1;
r=2.75; a=0.5000;b=0.07
T=1000;

for (t in 1:(T-1)){
  x[t+1]=x[t]*exp(r-a*x[t]-b*y[t])
  y[t+1]=y[t]*exp(r-a*y[t]+b*y[t])
}

tau=c(1,2,3)
an=60;ys=3/4;
par(mfrow=c(1,2))
plot3D <- scatterplot3d(x[1:(T-1)],y[1:(T-1)],x[2:T], type = "p", xlab = "X(t)", ylab = "Y(t)", zlab =
i=1
plot3Dx <- scatterplot3d(x[(tau[i]+1):(lt-tau[i])],-x[1:(lt-2*tau[i])],x[(1+2*tau[i]):lt], type = "p",
```

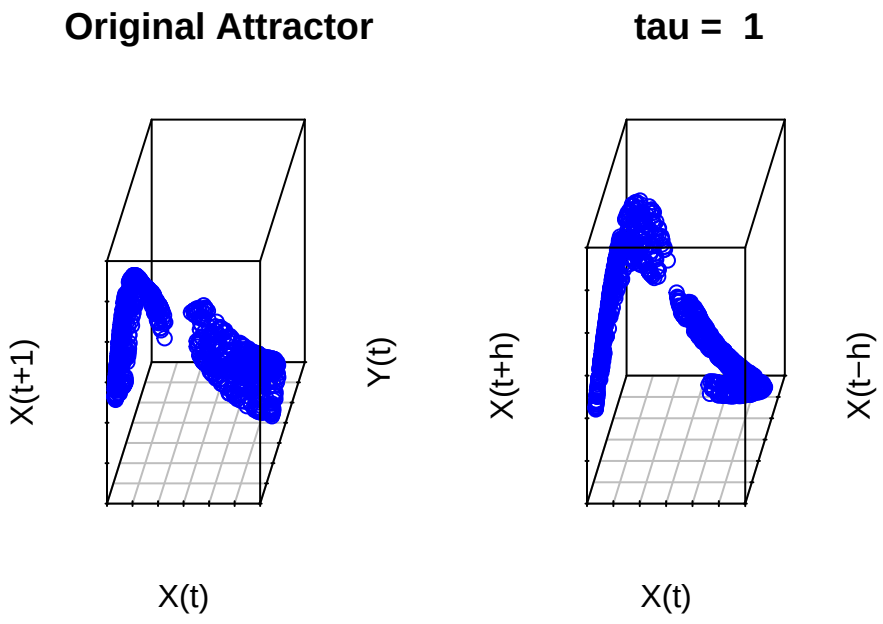

Figure 17: Attractor and reconstruction for the 2-d Ricker model. In the lagged coordinate panel, lags of the x were used to reconstruct the attractor.

The main reason to think about a 2-d discrete time case is that it is often possible to work out an exact expression for the delay-coordinate model. Here, the original system is given by

(3)

$$x_{t+1} = x_t \exp\left[r - ax_t - by_t\right]$$

(4)

$$y_{t+1} = y_t \exp\left[r - ay_t + bx_t\right]$$

To obtain the exact expression for the delay coordinate model, we will shift y back in time and solve the resulting set of equations for x_{t+1} in terms of x_t and x_{t-1}. Breaking this down, we first shift (4) back a time step and plug into (3) to get

(5)

$$x_{t+1} = x_t \exp\left[r - ax_t - b\{y_{t-1} \exp\left[r - ay_{t-1} + bx_{t-1}\right]\}\right]$$

which probably doesnt seem like it did much good. But now, we can shift (3) back a step and solve for y_{t-1}. Doing so, we get

(6)

$$y_{t-1} = \left[r - ax_{t-1} - \ln\left[x_t/x_{t-1}\right]\right]/b$$

And finally, we can plug (6) into (5) to get

(7)

$$x_{t+1} = x_t \exp\left[r - ax_t - \{\left[r - ax_{t-1} - \ln\left[x_t/x_{t-1}\right]\right] \exp\left[r - a\left[r - ax_{t-1} - \ln\left[x_t/x_{t-1}\right]\right]/b + bx_{t-1}\right]\}\right]$$

In all honesty, this isnt pretty. But the point is that the fact that we can predict x_{t+1} using x_t and x_{t-1} doesnt require any magic, or even topology - in this case all that's required is algebra. The topological proof just demonstrates that this should work even when an exact algebraic expression isnt available.

Another take on Takens

Based on this little exercise with the 2-d model, we can stretch our imaginations a bit and relax our mathematical rigor a lot to think of Taken's theorem as saying something along these lines:

Imagine we have an m dimensional system composed of n observable $\mathbf{x} = \{x_1, ...x_n\}$ state variables and s unobservable $\mathbf{y} = \{y_1, ...y_s\}$ state variables (so that $m = n + s$) where the dynamics are governed by the maps

(8)

$$\mathbf{x}_{t+1} = F\left[\mathbf{x}_t, \mathbf{y}_t\right]$$

and

(9)

$$\mathbf{y}_{t+1} = G\left[\mathbf{x}_t, \mathbf{y}_t\right]$$

Next, assume that (8) is invertible such that

(10)

$$\mathbf{y}_{t-1} = \Phi\left[\mathbf{x}_t, \mathbf{x}_{t-1}\right]$$

then the dynamics may be equivalently re-written using only lags of the observables as

(11)

$$\mathbf{x}_{t+1} = F\left[\mathbf{x}_t, G\left[\mathbf{x}_{t-1}, \Phi\left[\mathbf{x}_t, \mathbf{x}_{t-1}\right]\right]\right]$$

Of course, when n isnt equal to s more lags may be required. It should be said that this expression is not a proof or a rigourous re-statement of Takens' theorem. It is just a heuristic for building intuition about why Takens' should work.

It also highlights another quirk of dynamical systems. There isnt any algebraic reason that we had to stop at shifting back one step. We could have equivalently shifted back two steps before solving to get rid of y_{t-2},

(12)

$$\mathbf{x}_{t+1} = F\left[\mathbf{x}_t, G\left[\mathbf{x}_{t-1}, G\left[\mathbf{x}_{t-2}, \Phi\left[\mathbf{x}_{t-1}, \mathbf{x}_{t-2}\right]\right]\right]\right]$$

Or we could go crazy and go back three steps, four steps, etc. All of these are *algebraically* equivalent to the original 2-d system. In practice, going back further tends to make things worse - for much the same reason that the 10-step ahead map in Figure 5 was so wiggly.

In addition, although Takens' theorem was proved using lags of only a single state variable, this line of thinking indicates that it is possible- and sometimes highly beneficial - to use combinations of observables when more than one is available. That is, if the complete system is 3-d but we only have data for 2 state variables, we can either use lags of x_1, lags of x_2 or lags of both to make predictions. Which approach is best depends on the availability of data in regions where the reconstructed attractor is changing rapidly. Obviously, if we have data on all 3, we should probably just use them directly!

Finally, thinking about things this way also helps us see what other tricks we might do with Takens'. For example, when y is an exogenous driver of x, so that (9) doesnt depend on $\mathbf{x}$, we can a) write y_{t+1} in terms of (x_t, x_{t+1}) and b) write x_{t+1} in terms of (x_t, x_{t-1}) but - *and this is the important part* - we can't write x_{t+1} solely in terms of y! This is the core idea underlying the most counter-intuitive claim of Convergent Cross Mapping (Sugihara et al 2012): If lags of x predict y but lags of y dont predict x then y is a unidirectional driver for x. The algebraic manipulations described here indicate how this works without recourse to any topological jargon.

Excercises:

1. Using the notation above, write out a generic time-lagged expression for x_t in terms of y_t and y_{t-1} in the case where G is independent of x
2. Adapt the local-linear regression code to using either (x_t, y_t) or (x_t, x_{t-1}) to predict x_{t+1}.

Summary

This set of notes was alot longer than I thought it would be. And there are still lots of neat things we didnt cover! But it is probably worthwhile to recap what we did cover and what I was hoping to get across.

The first point I was trying to make is that alot of what we think of as noise in ecology isnt really random *per se*. Rather it is the result of looking at complex, high dimensional dynamics through a low dimensional lens. This has important implications for applied ecology - those things we are treating as noise probably respond, eigher directly or indirectly, to our impacts on eosystems in ways that undermine our efforts at conservation and management.

The second point is that complex dynamics set a firm upper bound on how well we can predict ecological dynamics- Even if we knew exactly how things work, which we rarely if ever do. The practical implication of this is that conservation and management actions need to be based on short-term predictions, rather than equilibria.

The third point is corrollary to this - if the time intervals between samples are too long, we are likely to conclude that the system is unpredictable.

The last point is that we can mitigate both the incompleteness of our data and our uncertainty in how ecosystems work by making sure we sample often enough and using empirical dynamics and delay embedding to compensate for missing state variables.