10. Statistical methodology for nonlinear partially observed Markov process models

Edward Ionides

3/12/2018

Contents

Objectives

1. To introduce students to the **pomp** package
2. To explain how the components of a POMP model are encoded in this package
3. To give some experience in the use and manipulation of `pomp` objects

---

10.1 Time series analysis via nonlinear partially observed Markov process (NL-POMP) models

10.1.1 Six problems of Bjornstad and Grenfell (Science, 2001)

- Obstacles for **ecological** modeling and inference via nonlinear mechanistic models:

1. Combining measurement noise and process noise.
2. Including covariates in mechanistically plausible ways.
3. Continuous time models.
4. Modeling and estimating interactions in coupled systems.
5. Dealing with unobserved variables.
6. Modeling spatial-temporal dynamics.

10.1.2 Applications of NL-POMP models

- The same issues arise for any modeling and inference via nonlinear mechanistic models. This arises throughout engineering, the sciences (social, biological and physical) and business.
- For example, in finance, we considered a stochastic volatility example in Chapter 1
- Many applications arise in epidemiology, studying the transmission of infectious diseases.
- Infectious disease dynamics can be highly nonlinear:
  1. Transmission arises when an infected individual contacts a susceptible individual. This leads to a quadratic term in the rate of infections, which should be proportional to

     Fraction of individuals infected
     × Fraction of individuals susceptible to infection

  2. Each new infection depletes the pool of susceptible individuals.
  3. Without depletion of susceptibles, the fraction of individuals susceptible to infection is constant and the epidemic grows exponentially. Then, the system is linear on a log scale.
- Data on infectious diseases are generally limited to diagnosed cases. Much of the transmission dynamics cannot be directly observed.
- Infectious disease epidemiology has motivated developments in statistical methodology and software for NL-POMP models, including a previous course used as a source for these class notes.
- Many other biological populations have similar nonlinearities: the population grows exponentially until limited by some constraint (which could be a food resource or a predator). When the resource is used up, or the predator becomes abundant, the population crashes. Then a new cycle begins.

10.2 An algorithmic approach to inference for POMP models

- Recall our notation for partially observed Markov process models.
- Write $X_n = X(t_n)$ and $X_{0:N} = (X_0, \ldots, X_N)$. Let Y_n be a random variable modeling the observation at time t_n.
- The one-step transition density, $f_{X_n|X_{n-1}}(x_n|x_{n-1};\theta)$, together with the measurement density, $f_{Y_n|X_n}(y_n|x_n;\theta)$ and the initial density, $f_{X_0}(x_0;\theta)$, specify the entire joint density via

$$f_{X_{0:N},Y_{1:N}}(x_{0:N},y_{1:N};\theta) = f_{X_0}(x_0;\theta)\,\prod_{n=1}^{N} f_{X_n|X_{n-1}}(x_n|x_{n-1};\theta)\,f_{Y_n|X_n}(y_n|x_n;\theta).$$

- The marginal density for sequence of measurements, $Y_{1:N}$, evaluated at the data, $y^*_{1:N}$, is

$$f_{Y_{1:N}}(y^*_{1:N};\theta) = \int f_{X_{0:N},Y_{1:N}}(x_{0:N},y^*_{1:N};\theta)\,dx_{0:N}.$$

- To think algorithmically, we define some function calls that provide **basic elements** specifying a POMP model.
  - `rprocess( )`: a draw from the one-step transition distribution, with density $f_{X_n|X_{n-1}}(x_n|x_{n-1};\theta)$.
  - `dprocess( )`: evaluation of the one-step transition density, $f_{X_n|X_{n-1}}(x_n|x_{n-1};\theta)$.

  - `rmeasure( )`: a draw from the measurement distribution with density $f_{Y_n|X_n}(y_n|x_n;\theta)$.
  - `dmeasure( )`: evaluation of the measurement density, $f_{Y_n|X_n}(y_n|x_n;\theta)$.
- This follows the standard R notation, for example we expect `rnorm` to draw from the normal distribution, and `dnorm` to evaluate the normal density.
- A general POMP model is fully specified by defining these basic elements.
- The user will have to say what the basic elements are for their chosen POMP model.
- Algorithms can then use these basic elements to carry out inference for the POMP model.
- We will see that there are algorithms that can carry out likelihood-based inference for this general POMP model specification.

10.2.1 What does it mean for statistical methodology to be simulation-based?

- Oftentimes, simulating random processes is easier than evaluating their transition probabilities.
- In other words, we may be able to write `rprocess()` but not `dprocess()`.
- **Simulation-based** methods require the user to specify `rprocess()` but not `dprocess()`.
- **Plug-and-play**, **likelihood-free** and **equation-free** are alternative terms for simulation-based.
- Much development of simulation-based statistical methodology has occurred in the past decade.

10.3 The pomp R package for POMP models

- **pomp** is an R package for data analysis using partially observed Markov process (POMP) models.
- Note the distinction: lower case **pomp** is a software package; upper case POMP is a class of models.
- **pomp** builds methodology for POMP models in terms of arbitrary user-specified `rprocess()`, `dprocess()`, `rmeasure()`, and `dmeasure()` functions.
- Following modern practice, most methodology in **pomp** is simulation-based, so does not require specification of `dprocess()`.
- **pomp** has facilities to help construct `rprocess()`, `rmeasure()`, and `dmeasure()` functions for model classes of scientific interest.
- **pomp** provides a forum for development, modification and sharing of models, methodology and data analysis workflows.
- **pomp** is available from CRAN

10.4 Example: the Ricker model

- The Ricker model is a basic model in population biology.
- We'll start with a deterministic version and then add process noise and measurement error.

10.4.1 A deterministic version of the Ricker model.

- The **Ricker equation** describes the deterministic dynamics of a simple population, modeling population growth and resource depletion.

$$[R1] \qquad P_{n+1} = r\, P_n\, \exp(-P_n).$$

- Here, P_n is the population density at time $t_n = n$ and r is a fixed value (a parameter), related to the population's intrinsic capacity to increase.
- Notice that $P_n = \log(r)$ is an **equilibrium**. If $P_n = \log(r)$ then $P_{n+1} = P_{n+2} = \cdots = P_n$. Another equilibrium is $P_n = 0$. It is not obvious whether [R1] converges to an equilibrium.
- P is a *state variable*, r is a *parameter*.
- If we know r and the *initial condition* P_0, this deterministic Ricker equation predicts the future population density at all times $n = 1, 2, \ldots$.
- We can view the initial condition, P_0 as a special kind of parameter, an *initial-value parameter*.

10.4.2 Adding stochasticity to the Ricker equation

- We can model process noise in this system by making the growth rate r into a random variable.
- For example, if we assume that the intrinsic growth rate is log-normally distributed, P becomes a stochastic process governed by

$$[R2] \qquad P_{n+1} = r\, P_n\, \exp(-P_n + \varepsilon_n), \qquad \varepsilon_n \sim \text{Normal}(0, \sigma^2),$$

- Here, the new parameter σ is the standard deviation of the noise process ε.

10.4.3 Question: does adding Gaussian noise mean we have a Gaussian latent process model?

- What does it mean to say that the model for $P_{0:N}$ described by equation [R2] is Gaussian?

A **Gaussian process** is one where all joint distributions are multivariate normal. According to this definition, $P_{0:N}$ is *not* Gaussian.

10.4.4 Adding measurement error to the Ricker model

- Let's suppose that the Ricker model is our model for the dynamics of a real population.
- For most populations, outside of controlled experiments, we cannot know the exact population density at any time, but only estimate it through sampling.
- Let's model measurement error by treating the measurement y_n^*, conditional on P_n, as a draw from a Poisson distribution with mean $\phi\, P_n$. This corresponds to the model

$$[R3] \qquad Y_n|P_n \sim \text{Poisson}(\phi\, P_n).$$

- The parameter ϕ is proportional to the sampling effort.

10.4.5 Writing the Ricker model as a POMP model

- For our standard definition of a POMP model $(X_{0:N}, Y_{0:N})$, we can check that equations [R2] and [R3] together with the parameter P_0 define a POMP model with

$$X_n = P_n \tag{1}$$
$$Y_n = Y_n \tag{2}$$

.

- Following the usual POMP paradigm, P_n is a true but unknown population density at time t_n.

10.4.6 Working with the Ricker model in pomp.

- The R package **pomp** provides facilities for modeling POMPs, a toolbox of statistical inference methods for analyzing data using POMPs, and a development platform for implmenting new POMP inference methods.
- The basic data-structure provided by **pomp** is the object of class `pomp`, alternatively known as a `pomp` object.
- A `pomp` object is a container that holds real or simulated data and a POMP model, possibly together with other information such as model parameters, that may be needed to do things with the model and data.
- Let's see what can be done with a `pomp` object.
- First, if we haven't already, we must install **pomp**. This can be done from CRAN, by

```
install.packages("pomp")
```

- If you want the latest version, with the source code, you can keep a local clone of the **pomp** repository on Github and install it from there. For example, in a Mac or Linux terminal,

```
git clone git@github.com:kingaa/pomp
R CMD INSTALL pomp
```

- Now we'll load some packages, including **pomp**.

```
set.seed(594709947L)
require(ggplot2)
require(plyr)
require(reshape2)
require(pomp)
stopifnot(packageVersion("pomp")>="0.69-1")
```

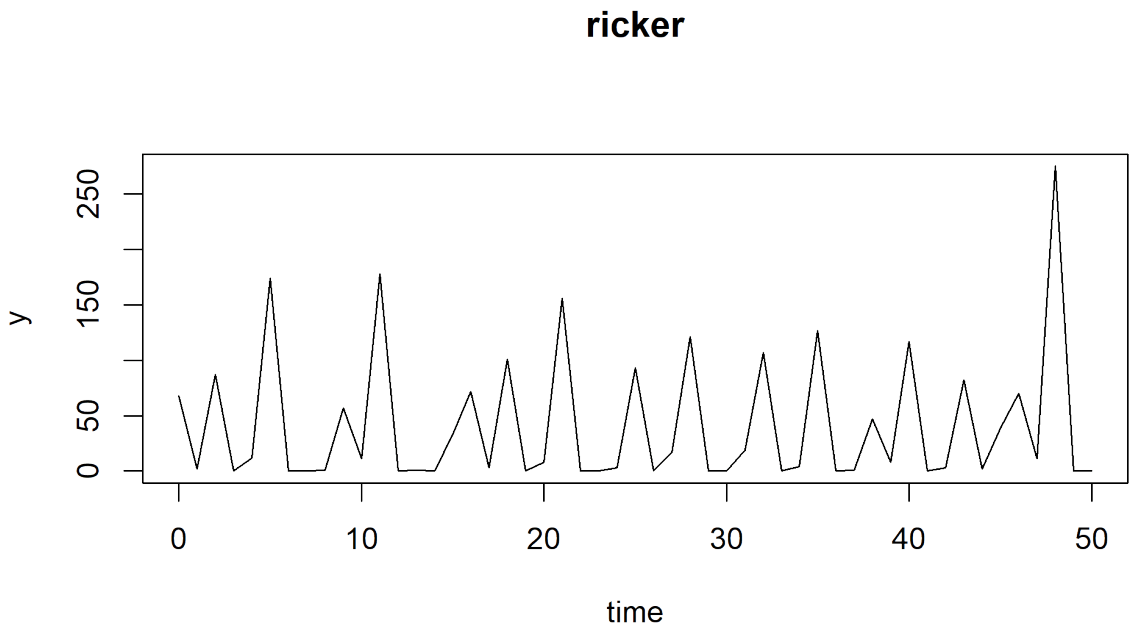
- A pre-built `pomp` object encoding the Ricker model comes included with the package. Load it by

```
pompExample(ricker)
```

```
## newly created object(s):
##  ricker
```

- This has the effect of creating a `pomp` object named `ricker` in your workspace.
- We can plot the data by doing

```
plot(ricker)
```



- Note that this **pomp** representation uses `N` for our variable `P_n`
- We can simulate by doing

```
x <- simulate(ricker)
```

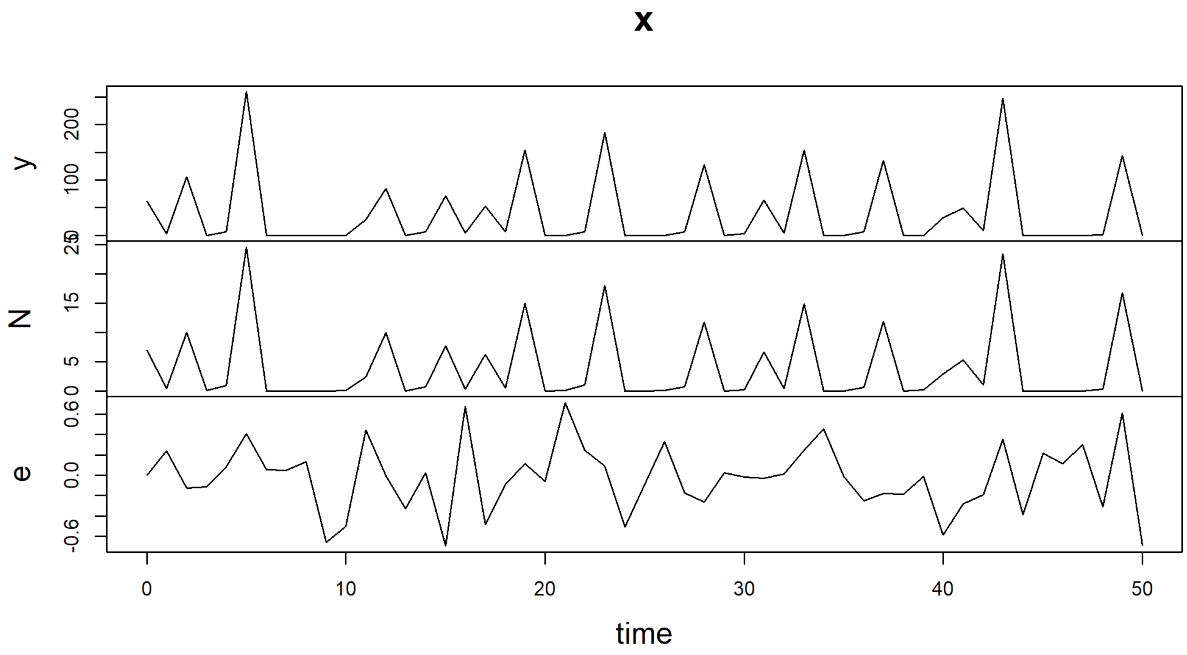
- What kind of object have we created?

```
class(x)
```

```
## [1] "pomp"
## attr(,"package")
## [1] "pomp"
```

```
plot(x)
```

We can use two different POMP models: (1) $X_n = P_n$ and $Y_n = Y_n$ or (2) $X_n = (P_n, \log(\epsilon_n))$ and $Y_n = Y_n$. Thus, we actually chose the higher dimensional model representation. This is so that we can save the simulated draws of $\log(\epsilon_n)$ (to run diagnostics, etc).



10.4.7 Question: What is a generic function?

- How does the concept of a **generic function** fit in with the following related concepts,
  - **object-oriented programming**.
  - assigning a **class** to an object.
  - **overloading** of functions or operators.
  - **inheritance** between classes, when one class extends another.
- How does object-oriented programming work in R? How is this similar or different from any other environment in which you have seen object-oriented programming?
- For current purposes, we don't need to be experts in object-oriented programming in R. However, we should know of the existence of the two main object-oriented systems in R,
  - **S3 classes**
  - **S4 classes**
- We should be able to recognize when code we are using employs S3 and S4 classes.

- We should know where to turn to for help if we find ourselves needing to know more details about how these work.
- **pomp** uses the S4 class system, so that system is of more immediate relevance. Many older R packages use S3 classes.

- Why do we see more time series in the simulated `pomp` object?
- We can turn a `pomp` object into a data frame:

```
y <- as.data.frame(ricker)
head(y)
```

```
##   time   y
## 1    0  68
## 2    1   2
## 3    2  87
## 4    3   0
## 5    4  12
## 6    5 174
```

```
head(simulate(ricker,as.data.frame=TRUE))
```

```
##   time  y          N            e sim
## 1    0 68 7.00000000  0.000000000   1
## 2    1  3 0.26614785 -0.069613447   1
## 3    2 96 9.10502650 -0.001322229   1
## 4    3  0 0.06857102  0.416314640   1
## 5    4 22 3.20815215  0.114151375   1
## 6    5 86 8.95670223  0.434859137   1
```

- We can also run multiple simulations simultaneously:

```
x <- simulate(ricker,nsim=10)
class(x)
```

```
## [1] "list"
```

```
sapply(x,class)
```

```
##  [1] "pomp" "pomp" "pomp" "pomp" "pomp" "pomp" "pomp" "pomp" "pomp" "pomp"
```

```
x <- simulate(ricker,nsim=10,as.data.frame=TRUE)
head(x)
```

```
##   time   y            N          e sim
## 1    0  61 7.000000e+00  0.0000000   1
## 2    1   3 2.400265e-01 -0.1729162   1
## 3    2  53 5.293097e+00 -0.4665640   1
## 4    3  17 1.443727e+00  0.1939209   1
## 5    4 155 1.868361e+01  0.2041458   1
## 6    5   0 8.847808e-06  0.3206253   1
```

```
str(x)
```

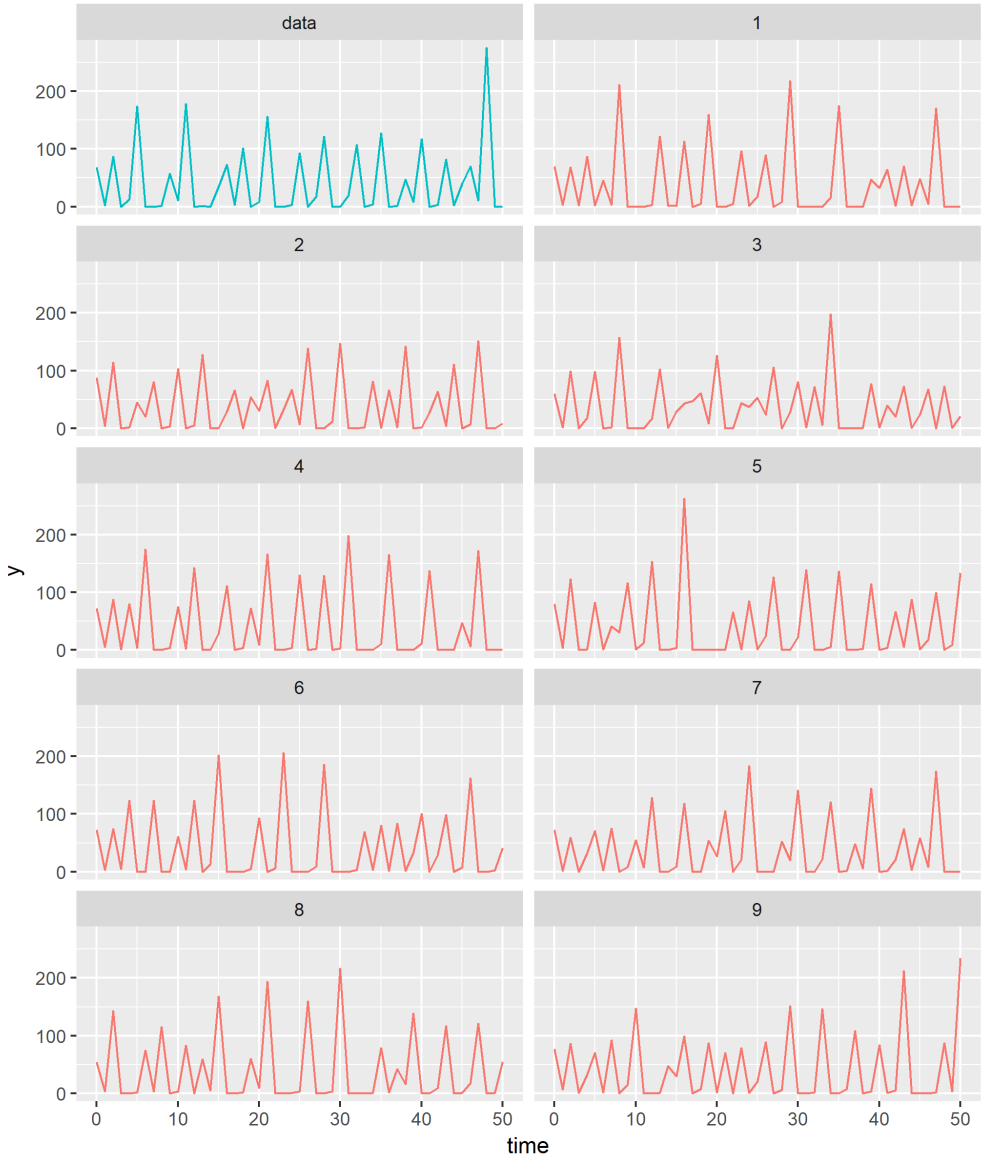
```
## 'data.frame':    510 obs. of  5 variables:
##  $ time: num  0 1 2 3 4 5 6 7 8 9 ...
##  $ y   : num  61 3 53 17 155 0 0 0 13 95 ...
```

```
##  $ N   : num  7 0.24 5.29 1.44 18.68 ...
##  $ e   : num  0 -0.173 -0.467 0.194 0.204 ...
##  $ sim : Ord.factor w/ 10 levels "1"<"2"<"3"<"4"<..: 1 1 1 1 1 1 1 1 1 1 ...
```

In above df, gives you observation process and state (latent) process. In pomp terminology, N is the latent process, Y is the observed process

Also,

```
x <- simulate(ricker,nsim=9,as.data.frame=TRUE,include.data=TRUE)
ggplot(data=x,aes(x=time,y=y,group=sim,color=(sim=="data")))+
  geom_line()+guides(color=FALSE)+
  facet_wrap(~sim,ncol=2)
```



- We can compute a trajectory of the deterministic skeleton

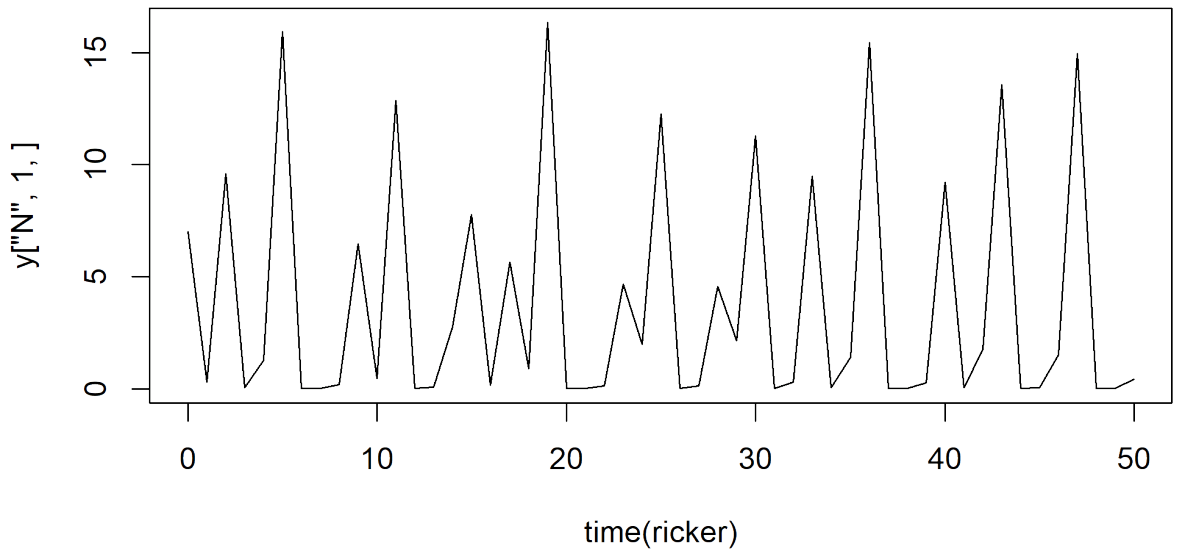
```
y <- trajectory(ricker)
dim(y)
```

```
## [1]  2  1 51
```

```
dimnames(y)
```

```
## $variable
## [1] "N" "e"
##
## $rep
## NULL
##
## $time
## NULL
```

```
plot(time(ricker),y["N",1,],type="l")
```



- Notice that `ricker` has parameters associated with it:

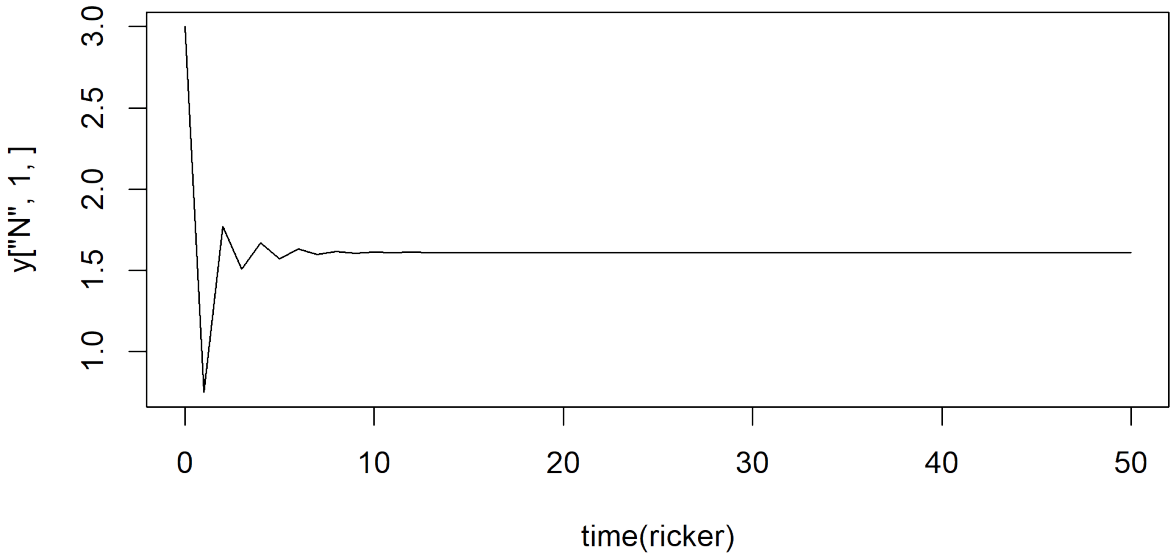
```
coef(ricker)
```

```
##        r    sigma      phi        c      N.0      e.0
## 44.70118  0.30000 10.00000  1.00000  7.00000  0.00000
```

In output above, N.0 is the initial value of the state (latent) process

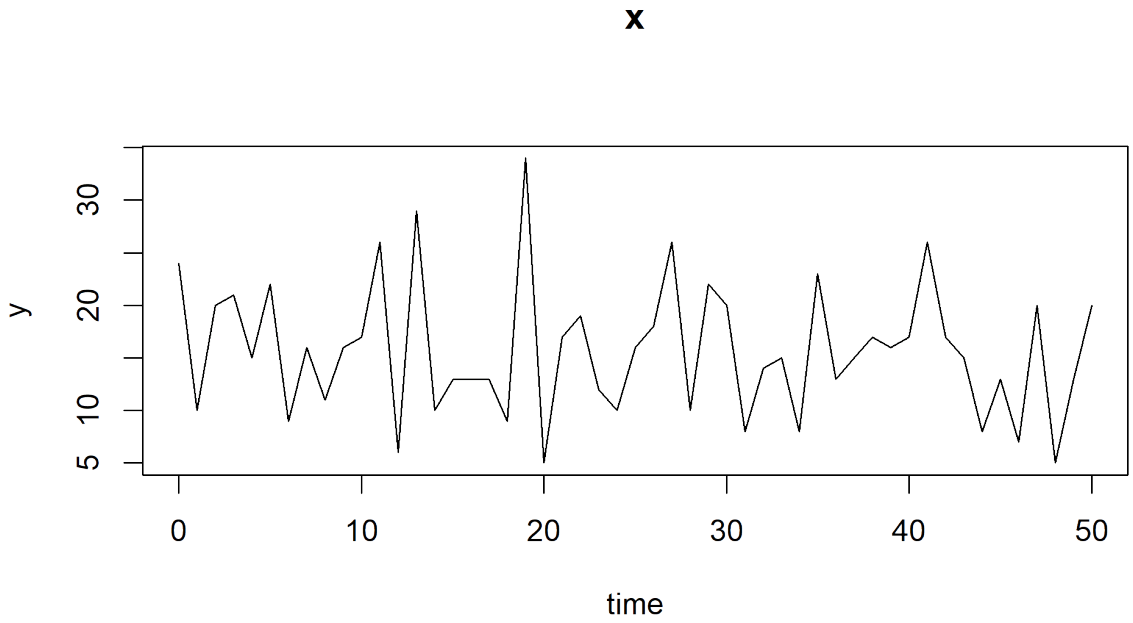
- These are the parameters at which the simulations and deterministic trajectory computations above were done.
- We can run these at different parameters:

```
theta <- coef(ricker)
theta[c("r","N.0")] <- c(5,3)
y <- trajectory(ricker,params=theta)
plot(time(ricker),y["N",1,],type="l")
```



New set of parameters creates state process that converges to an equilibrium population density.

```
x <- simulate(ricker,params=theta)
plot(x,var="y")
```



Here, the dynamics are pushing this to a stable process.

Adding noise to a stable process doesn't have long-term effects (as in above). However, for the many gg-plots above, we have unstable processes.

Note that in dynamics we have (1) chaotic cycles (i.e. unstable) and (2) stable cycles. In (1), a small difference in initial conditions leads to larger differences later. Here, sample paths are divergent, viewed as a function of initial conditions. In (2), there exists an equilibrium or stable limit cycle.

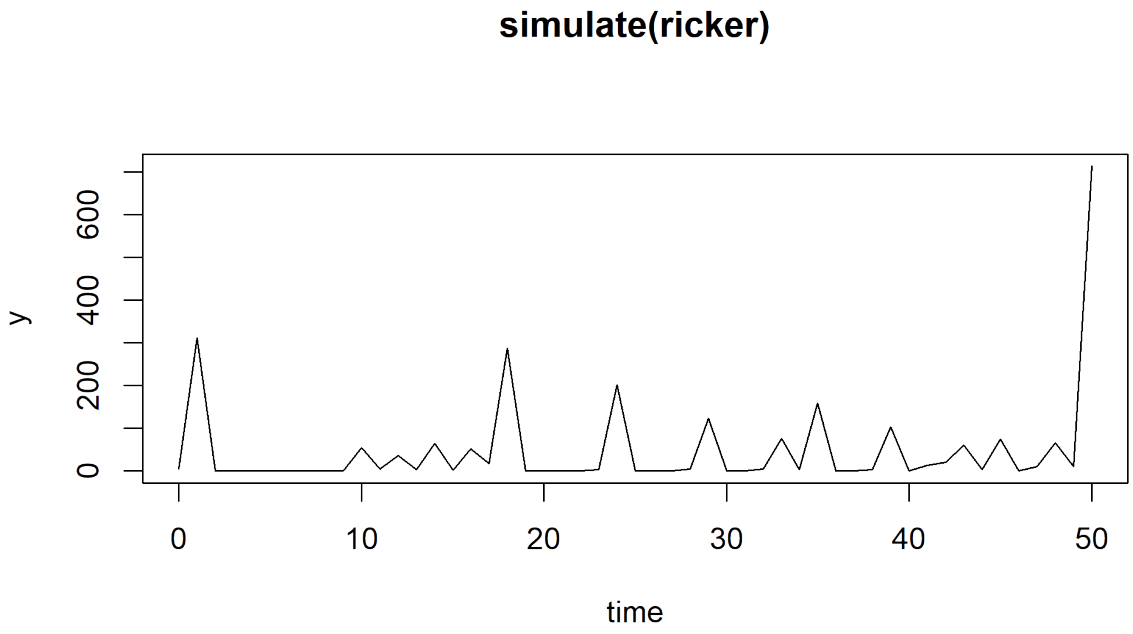
- We can also change the parameters stored inside of `ricker`:

```
coef(ricker,c("r","N.0","sigma")) <- c(39,0.5,1)
coef(ricker)
```

```
##     r sigma   phi     c   N.0   e.0
##  39.0   1.0  10.0   1.0   0.5   0.0
```

The c parameter above is in front of $-P_n$ in the Ricker equation in R2. Here, it's just 1.

```
plot(simulate(ricker),var="y")
```



- In all of the above, it's possible to work with more than one set of parameters at a time. For example:

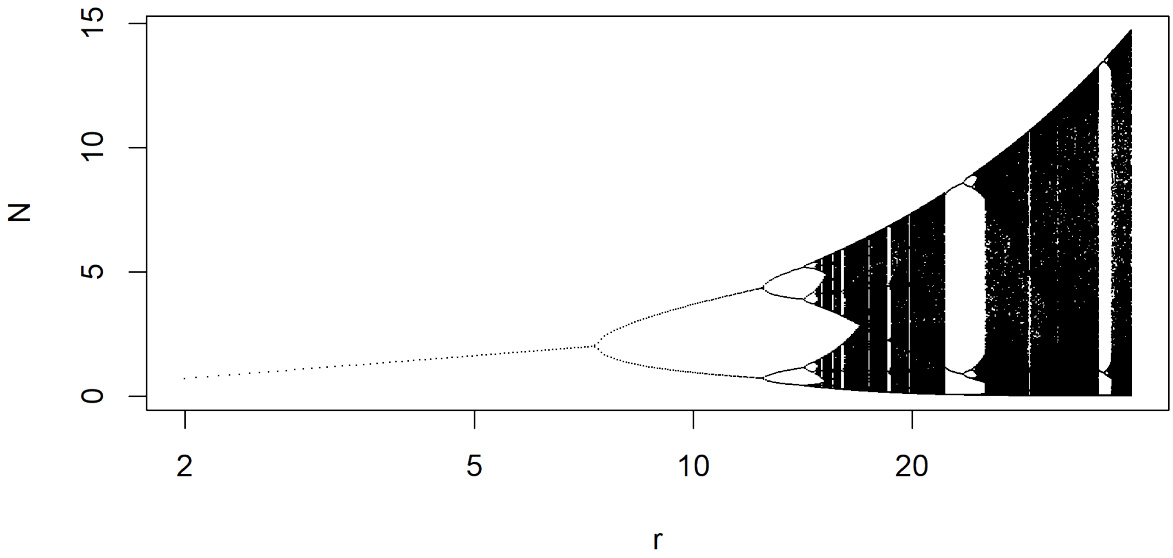
```
p <- parmat(coef(ricker),500)
dim(p); dimnames(p)
```

```
## [1]   6 500
```

```
## $variable
## [1] "r"     "sigma" "phi"   "c"     "N.0"   "e.0"
##
## $rep
```

```
## NULL
```

```
p["r",] <- seq(from=2,to=40,length=500)
y <- trajectory(ricker,params=p,times=200:1000)
matplot(p["r",],y["N",,],pch=".",col='black',xlab='r',ylab='N',log='x')
```



- This figure is called a bifurcation diagram for the Ricker map. The Ricker map is another name for the Ricker equation: the Ricker equation defines a recursion, and these recursions are often called maps when mathematically studying their behavior.
  - How do you interpret this bifurcation diagram?
  - What does it mean when the single line for small values of r splits into a double line, around $r = 0.8$?
  - What does it mean when solid vertical lines appear, around $r = 18$?
  - A bifurcation diagram like this can only be computed for a deterministic map. Why? However, the bifurcation diagram for the deterministic skeleton can be useful to help understand a stochastic process. We'll see an example later in this chapter.
  - Look at the R code for the bifurcation diagram. Notice that the first 200 iterations of the Ricker map are discarded, by setting `times=200:1000`. Why? This is a technique called **burn-in**, by analogy with an industrial technique by the same name. Burn-in is a standard technique in Markov chain Monte Carlo, as described in the Wikipedia article on the Metropolis-Hastings algorithm: "The Markov chain is started from an arbitrary initial value and the algorithm is run for many iterations until this initial state is forgotten. These samples, which are discarded, are known as burn-in." The concept of burn-in applies to any simulation study where one is interested in simulating the steady state of a dynamic system, ignoring the transient behavior due to the choice of starting values for the simulation.
- More information on manipulating and extracting information from `pomp` objects can be viewed in the help pages (`methods?pomp`).

- There are a number of other examples included with the package. Do `pompExamples()` to see a list of these.

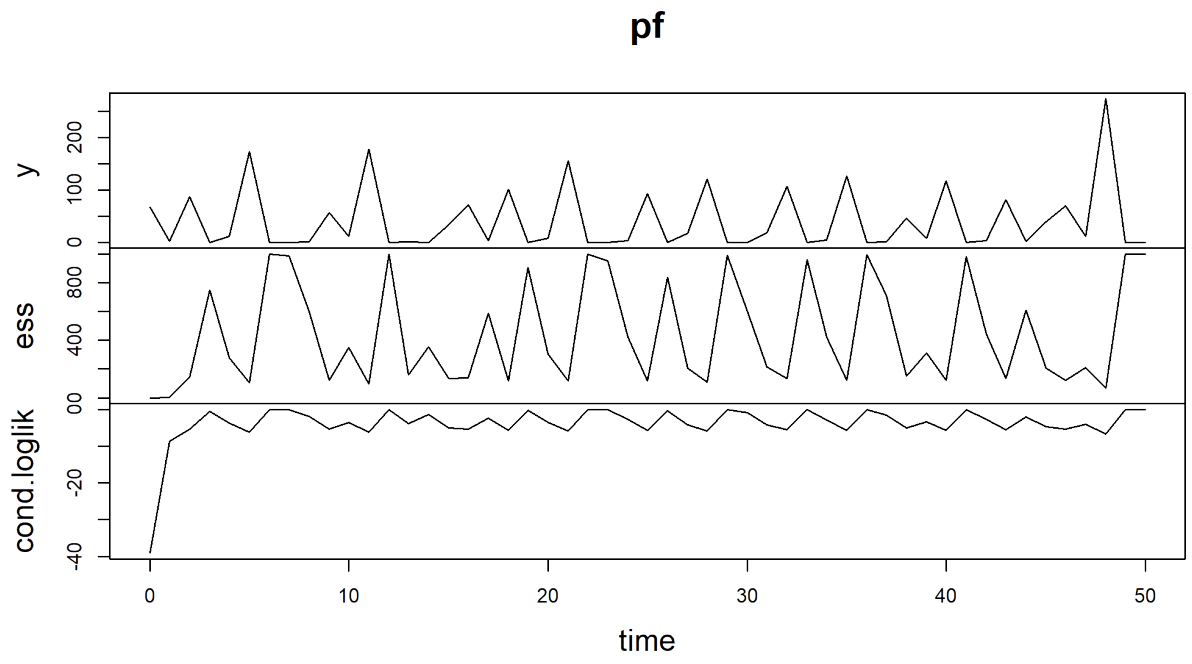
10.5 Inference algorithms in pomp

- **pomp** provides a wide range of inference algorithms. We'll learn about these in detail soon, but for now, let's just look at some of their general features.
- The `pfilter` function runs a simple **particle filter**, which is a Monte Carlo algorithm that can be used to evaluate the likelihood at a particular set of parameters. One uses the `Np` argument to specify the number of particles to use:

```
pf <- pfilter(ricker,Np=1000)
class(pf)
```

```
## [1] "pfilterd.pomp"
## attr(,"package")
## [1] "pomp"
```

```
plot(pf)
```



ess – effective sample size (from MCMC). The CLL is the conditional log-likelihood: $f_{Y_n|Y_{1:n-1}}$ for $n = 1, \cdots, N$

```
logLik(pf)
```

```
## [1] -200.5362
```

- Note that `pfilter` returns an object of class `pfilterd.pomp`. This is the general rule: inference algorithms return objects that are `pomp` objects with additional information. The package provides tools to extract this information.
- We can run the particle filter again by doing

```
pf <- pfilter(pf)
logLik(pf)
```

```
## [1] -200.7444
```

which has the result of running the same computation again.

- Note that, because the particle filter is a Monte Carlo algorithm, we get a slightly different estimate of the log likelihood.
- Note that, by default, running `pfilter` on a `pfilterd.pomp` object causes the computation to be re-run with the same parameters as before. Any additional arguments we add override these defaults. This is the general rule in **pomp**. For example,

```
pf <- pfilter(pf,Np=100)
logLik(pf)
```

```
## [1] -201.9509
```

Here, the particle filtering has been performed with only 100 particles.

10.6 Building a custom `pomp` object

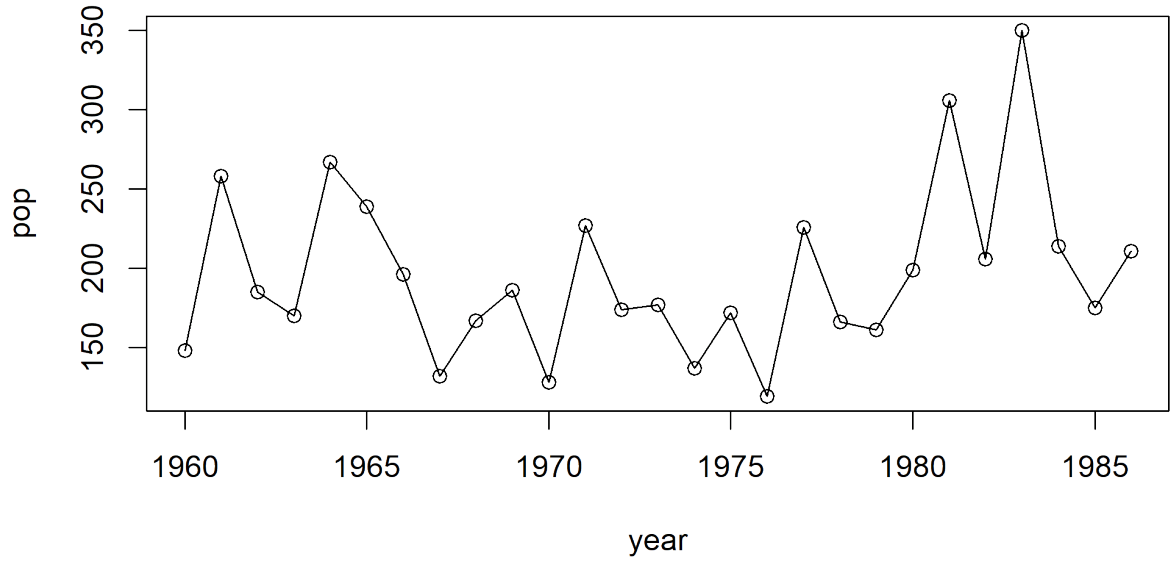
A real **pomp** data analysis begins with constructing one or more `pomp` objects to hold the data and the model or models under consideration. We'll illustrate this process a dataset of the abundance of the great tit (*Parus major*) in Wytham Wood, near Oxford (McCleery and Perrins 1991).

Download and plot the data:

```
dat <- read.csv("parus.csv")
head(dat)
```

```
##   year pop
## 1 1960 148
## 2 1961 258
## 3 1962 185
## 4 1963 170
## 5 1964 267
## 6 1965 239
```

```
plot(pop~year,data=dat,type='o')
```



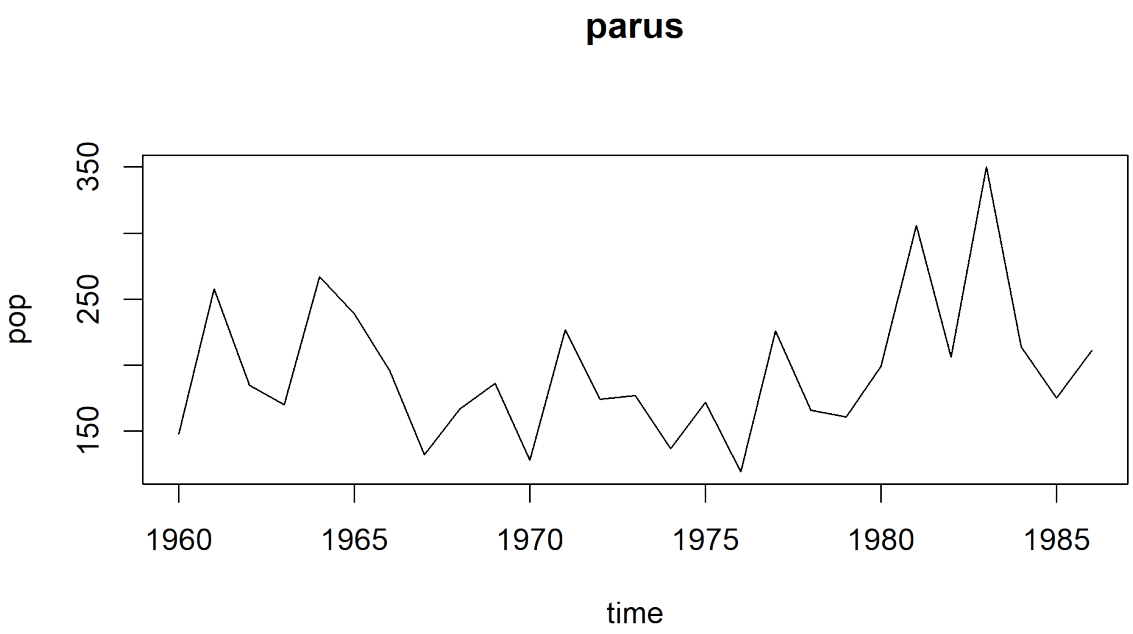
Let's suppose that we want to fit the stochastic Ricker model discussed above to these data.

The call to construct a `pomp` object is, naturally enough, `pomp`. Documentation on this function can be had by doing `?pomp`. Do `class?pomp` to get documentation on the `pomp` class. Learn about the various things you can do once you have a `pomp` object by doing `methods?pomp` and following the links therein. Read an overview of the package as a whole with links to its main features by doing `package?pomp`. A complete index of the functions in **pomp** is returned by the command `library(help=pomp)`. Finally, the home page for the `pomp` project is (http://kingaa.github.io/pomp); there you have access to the complete source code, manuals, mailing lists, etc.

```
require(pomp)
parus <- pomp(dat,times="year",t0=1959)
```

The `times` argument specifies that the column labelled "year" gives the measurement times; `t0` is the "zero-time", the time at which the state process will be initialized. We've set it to one year prior to the beginning of the data. Plot it:

```
plot(parus)
```



10.6.1 Adding in the deterministic skeleton

We can add the Ricker model deterministic skeleton to the `parus` `pomp` object. Since the Ricker model is a discrete-time model, its skeleton is a map that takes P_n to P_{n+1} according to the Ricker model equation

$$P_{n+1} = r\,P_n\,\exp(-P_n).$$

We provide this to `pomp` in the form of a `Csnippet`, a little snippet of C code that performs the computation.

```
skel <- Csnippet("DN = r*N*exp(-N);")
```

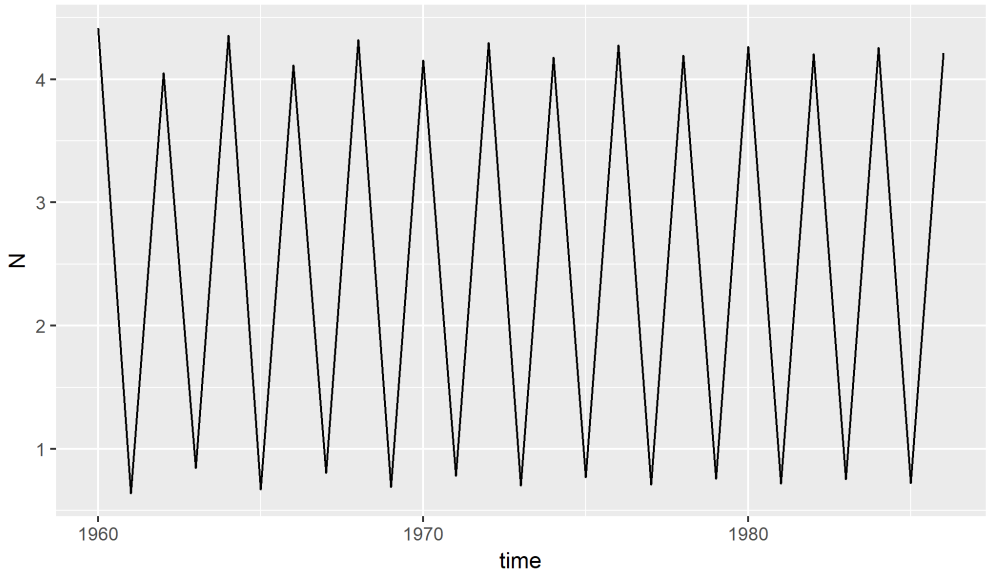
We then add this to the `pomp` object:

```
parus <- pomp(parus,skeleton=map(skel),statenames="N",paramnames="r")
```

Note that we have to inform **pomp** as to which of the variables we've referred to in `skel` is a state variable (`statenames`) and which is a parameter (`paramnames`).

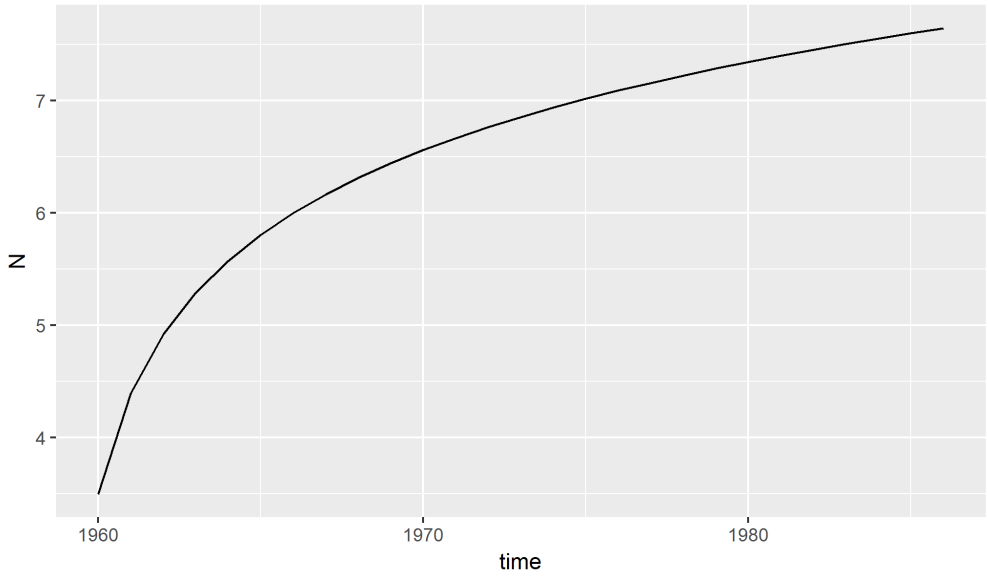
With just the skeleton defined, we are in a position to compute the trajectories of the deterministic skeleton at any point in parameter space. For example,

```
traj <- trajectory(parus,params=c(N.0=1,r=12), as.data.frame=TRUE)
ggplot(data=traj,aes(x=time,y=N))+geom_line()
```



It may be interesting to note how different the dynamics become if the skeleton is considered as the derivative of a differential equation rather than as a discrete time map. It is harder to get chaotic dynamics in a continuous time system.

```
parus2 <- pomp(parus,skeleton=vectorfield(skel),statenames="N",paramnames="r")
traj2 <- trajectory(parus2,params=c(N.0=1,r=12),as.data.frame=TRUE)
ggplot(data=traj2,aes(x=time,y=N))+geom_line()
```



10.6.2 A note on terminology

- If we know the state, $x(t_0)$, of the system at time t_0, it makes sense to speak about the entire trajectory of the system for all $t > t_0$.

- This is true whether we are thinking of the system as deterministic or stochastic.
- Of course, in the former case, the trajectory is uniquely determined by $x(t_0)$, while in the stochastic case, only the probability distribution of $x(t)$, $t > t_0$ is determined.
- In **pomp**, to avoid confusion, we use the term "trajectory" exclusively to refer to *trajectories of a deterministic process*. Thus, the `trajectory` command iterates or integrates the deterministic skeleton forward in time, returning the unique trajectory determined by the specified parameters. When we want to speak about sample paths of a stochastic process, we use the term *simulation*.

  For a stochastic latent process, we "simulate a sample path" whereas for a deterministic latent process, we "solve for the trajectory."

- Accordingly, the `simulate` command always returns individual sample paths from the POMP. In particular, we avoid "simulating a set of differential equations", preferring instead to speak of "integrating" the equations, or "computing trajectories".

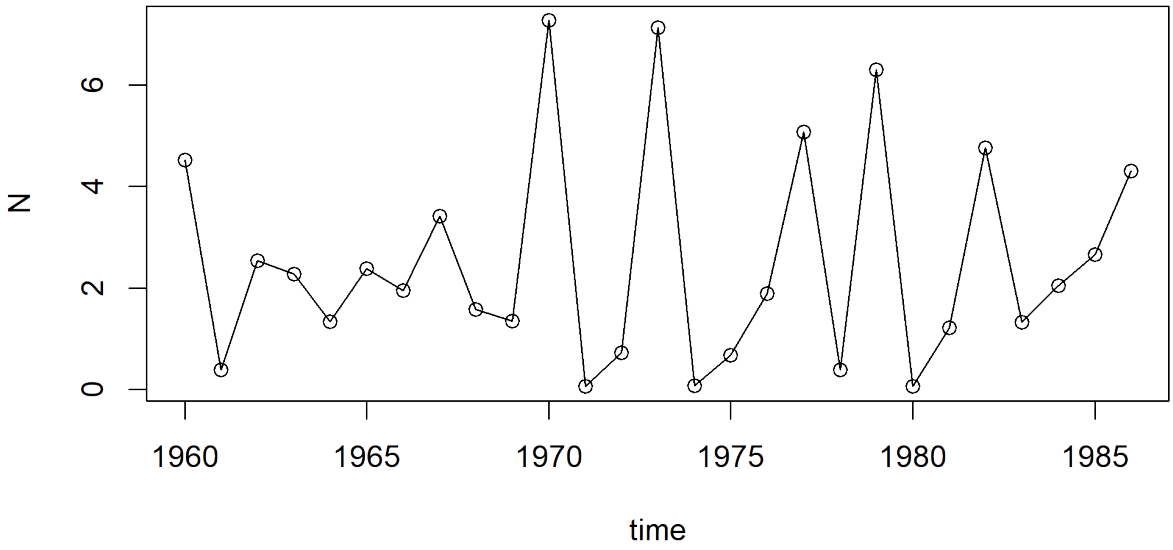
10.6.3 Adding in the process model simulator

- We can add the stochastic Ricker model to `parus` by writing a Csnippet that simulates one realization of the stochastic process, from an arbitary time t to $t+1$, given arbitrary states and parameters.
- The following does this.

```
stochStep <- Csnippet("
  e = rnorm(0,sigma);
  N = r*N*exp(-N+e);
")
pomp(parus,rprocess=discrete.time.sim(step.fun=stochStep,delta.t=1),
     paramnames=c("r","sigma"),statenames=c("N","e")) -> parus
```

- Note that in the above, we use the `exp` and `rnorm` functions from the **R** API.
- In general any C function provided by **R** is available to you. **pomp** also provides a number of C functions that are documented in the header file, `pomp.h`, that is installed with the package.
- See the `Csnippet` documentation (`?Csnippet`) to read more about how to write them.
- Note too that we use `discrete.time.sim` here because the model is a stochastic map.
- We specify that the time step of the discrete-time process is `delta.t`, here, 1 yr.
- At this point, we have what we need to simulate the stochastic Ricker model.

```
sim <- simulate(parus,params=c(N.0=1,e.0=0,r=12,sigma=0.5),
                as.data.frame=TRUE,states=TRUE)
plot(N~time,data=sim,type='o')
```




```
# lines(N~time,data=traj,type='l',col='red')
```

10.6.4 Adding in the measurement model and parameters

- We complete the specification of the POMP by specifying the measurement model.
- To obtain the Poisson measurement model described above, we write two Csnippets. The first simulates:

```
rmeas <- Csnippet("pop = rpois(phi*N);")
```

and the second computes the likelihood of observing `pop` birds given a true density of `N`:

```
dmeas <- Csnippet("lik = dpois(pop,phi*N,give_log);")
```

- Note the `give_log` argument. When this code is evaluated, `give_log` will be set to 1 if the log likelihood is desired, and 0 else.
- We add these specifications of `rmeasure` and `dmeasure` into the `pomp` object:

```
pomp(parus,rmeasure=rmeas,dmeasure=dmeas,statenames=c("N"),paramnames=c("phi")) -> parus
```

- Now we can simulate the whole POMP. First, let's add some parameters to the `pomp` object:

```
coef(parus) <- c(N.0=1,e.0=0,r=20,sigma=0.1,phi=200)
```

```
sims <- simulate(parus,nsim=3,as.data.frame=TRUE,include.data=TRUE)
ggplot(data=sims,mapping=aes(x=time,y=pop))+geom_line()+
  facet_wrap(~sim)
```

10.7 Exercises

10.7.1 Ricker model parameters

- Fiddle with the parameters to try and make the simulations look more like the data.
- This will help you build some intuition for what the various parameters do.

10.7.2 Reformulating the Ricker model

- Reparameterize the Ricker model so that the scaling of P is explicit:

$$P_{n+1} = r\,P_n\,\exp\left(-\frac{P_n}{K}\right).$$

- Modify the `pomp` object we created above to reflect this reparameterization.
- Modify the measurement model so that

$$\mathrm{pop}_n \sim \mathrm{Negbin}(\phi\,P_n, k),$$

  i.e., pop_n is negative-binomially distributed with mean $\phi\,P_t$ and clumping parameter k.
- See `?NegBinomial` for documentation on the negative binomial distribution and the **R** Extensions Manual section on distribution functions for information on how to access these in C.

10.7.3 Beverton-Holt

- Construct a `pomp` object for the *Parus major* data and the **stochastic Beverton-Holt** model,

$$P_{n+1} = \frac{a\,P_n}{1+b\,P_n}\,\varepsilon_n,$$

  where a and b are parameters and

$$\varepsilon_t \sim \text{Lognormal}(-\tfrac{1}{2}\sigma^2, \sigma^2).$$

- Assume the same measurement model as we used for the Ricker model.

Acknowledgment

These notes draw on material developed for a short course on Simulation-based Inference for Epidemiological Dynamics by Aaron King and Edward Ionides, taught at the University of Washington Summer Institute in Statistics and Modeling in Infectious Diseases, 2015, 2016 & 2017.

10.8 References

McCleery, R. H., and C. M. Perrins. 1991. Effects of predation on the numbers of great tits *Parus major*. Pages 129–147 *in* Bird population studies. relevence to conservation and management. Oxford University Press, Oxford.