ADVANCED TOPICS IN POMP

AARON A. KING

Contents

This document gives some examples of the use of native (C or FORTRAN) codes in `pomp` and introduces the low-level interface to `pomp` objects.

1. Acceleration using native codes: using plug-ins with native code

Since many of the methods we will use require us to simulate the process and/or measurement models many times, it is a good idea to use native (compiled) codes for the computational heavy lifting. This can result in many-fold speedup. `pomp` provides "plug-in" facilities to make it easier to define certain kinds of models. These plug-ins can be used with native codes as well, as we'll see in the following examples. The `pomp` package includes other examples that use C codes: these can be loaded using the `data` command.

In the "intro_to_pomp" vignette, we looked at the SIR model, which we implemented using an Euler-multinomial approximation to the continuous-time Markov process. Here is the same model implemented using native C codes:

```
pomp(
     data=data.frame(
       time=seq(from=1/52,to=4,by=1/52),
       reports=NA
       ),
     times="time",
     t0=0,
     ## native routine for the process simulator:
     rprocess=euler.sim(
       step.fun="_sir_euler_simulator",
       delta.t=1/52/20
       ),
     ## native routine for the skeleton:
     skeleton.type="vectorfield",
```

Date: May 9, 2011, `pomp` version 0.37-1.

```
  skeleton="_sir_ODE",
  ## binomial measurement model:
  rmeasure="_sir_binom_rmeasure",
  dmeasure="_sir_binom_dmeasure",
  ## name of the shared-object library containing the native routines:
  PACKAGE="pomp",
  ## the order of the observable assumed in the native routines:
  obsnames=c("reports"),
  ## the order of the state variables assumed in the native routines:
  statenames=c("S","I","R","cases","W"),
  ## the order of the parameters assumed in the native routines:
  paramnames=c(
    "gamma","mu","iota","beta1","beta.sd",
    "pop","rho","nbasis","degree","period"
    ),
  ## reset cases to zero at each new observation:
  zeronames=c("cases"),
  initializer=function(params,t0,...){
    p <- exp(params)
    with(
         as.list(p),
         {
           fracs <- c(S.0,I.0,R.0)
           x0 <- round(c(pop*fracs/sum(fracs),0,0))
           names(x0) <- c("S","I","R","cases","W")
           x0
         }
         )
  }
  ) -> sir
```

The source code for the native routines `_sir_euler_simulator`, `_sir_ODE`, `_sir_binom_rmeasure`, and `_sir_binom_dmeasure` is provided with the package (in the `examples` directory). To see the source code, do

```
file.show(file=system.file("examples/sir.c",package="pomp"))
```

Also in the `examples` directory is an R script that shows how to compile `sir.c` into a shared-object library and link it with R. Note that the native routines for this model are included in the package, which is why we give the `PACKAGE="pomp"` argument to `pomp`. When you write your own model using native routines, you'll compile them into a dynamically-loadable library. In this case, you'll want to specify the name of that library using the `PACKAGE` argument. Again, refer to the SIR example included in the `examples` directory to see how this is done.

Let's specify some parameters and simulate:

```
params <- c(
            gamma=26,mu=0.02,iota=0.01,
            beta1=1200,beta2=1800,beta3=600,
            beta.sd=1e-3,
            pop=2.1e6,
            rho=0.6,
            S.0=26/1200,I.0=0.001,R.0=1-0.001-26/1200
            )
```

```
sir <- simulate(sir,params=c(log(params),nbasis=3,degree=3,period=1),seed=3493885L)
tic <- Sys.time()
sims <- simulate(sir,nsim=10)
toc <- Sys.time()
print(toc-tic)
```

```
Time difference of 0.206414 secs
```

```
tic <- Sys.time()
traj <- trajectory(sir,hmax=1/52)
toc <- Sys.time()
print(toc-tic)
```

```
Time difference of 0.07029915 secs
```

2. Acceleration using native codes: writing rprocess and dprocess from scratch.

In the preceding example, we used "plug-ins" provided by the package, in conjunction with native C routines, to specify our model. One can also write simulators and density functions "from scratch". Here, we'll have a look at how the discrete-time bivariate AR(1) process with normal measurement error is implemented. You can load a pomp object for this model and have a look at its structure with the commands

```
require(pomp)
data(ou2)
show(ou2)
```

Here we'll examine how this object is put together. The process model simulator and density functions are as follows:

```
ou2.rprocess <- function (xstart, times, params, paramnames, ...) {
  nvar <- nrow(xstart)
  npar <- nrow(params)
  nrep <- ncol(xstart)
  ntimes <- length(times)
  ## get indices of the various parameters in the 'params' matrix
  ## C uses zero-based indexing!
  parindex <- match(paramnames,rownames(params))-1
  array(
        .C("_ou2_adv",
           X = double(nvar*nrep*ntimes),
           xstart = as.double(xstart),
           par = as.double(params),
           times = as.double(times),
           n = as.integer(c(nvar,npar,nrep,ntimes)),
           parindex = as.integer(parindex),
           DUP = FALSE,
           NAOK = TRUE,
           PACKAGE = "pomp"
           )$X,
        dim=c(nvar,nrep,ntimes),
        dimnames=list(rownames(xstart),NULL,NULL)
```

```
        )
}

ou2.dprocess <- function (x, times, params, log, paramnames, ...) {
  nvar <- nrow(x)
  npar <- nrow(params)
  nrep <- ncol(x)
  ntimes <- length(times)
  parindex <- match(paramnames,rownames(params))-1
  array(
        .C("_ou2_pdf",
           d = double(nrep*(ntimes-1)),
           X = as.double(x),
           par = as.double(params),
           times = as.double(times),
           n = as.integer(c(nvar,npar,nrep,ntimes)),
           parindex = as.integer(parindex),
           give_log=as.integer(log),
           DUP = FALSE,
           NAOK = TRUE,
           PACKAGE = "pomp"
           )$d,
        dim=c(nrep,ntimes-1)
        )
}
```

The call that constructs the `pomp` object is:

```
ou2 <- pomp(
            data=data.frame(
              time=seq(1,100),
              y1=NA,
              y2=NA
              ),
            times="time",
            t0=0,
            rprocess = ou2.rprocess,
            dprocess = ou2.dprocess,
            dmeasure = "_ou2_normal_dmeasure",
            rmeasure = "_ou2_normal_rmeasure",
            paramnames=c(
              "alpha.1","alpha.2","alpha.3","alpha.4",
              "sigma.1","sigma.2","sigma.3",
              "tau"
              ),
            statenames = c("x1","x2"),
            obsnames = c("y1","y2"),
            PACKAGE="pomp"
            )
```

Notice that the process model is implemented using using `.C`, while the measurement model is specified by giving the names of native C routines. Read the source to see the definitions of these functions. For convenience, the source codes are provided with the package in the `examples` directory. Do

```
file.show(file=system.file("examples/ou2.c",package="pomp"))
```

to view the source code.

There is an important issue that arises when using native codes. This has to do with the order in which parameters, states, and observables are passed into the native codes. `pomp` relies on the names (also row-names and column-names) attributes to identify variables in vectors and arrays. When you write a C or FORTRAN version of `rprocess` or `dmeasure` for example, you write a routine that takes parameters, state variables, and/or observables in the form of a vector. However, you have no control over the order in which these are given to you. Without some means of knowing which element of each vector corresponds to which variable, you cannot write the codes correctly. This is where the `paramnames`, `statenames`, and `obsnames` arguments to `pomp` come in. When you specifying the names of parameters, state variables, and observables (data variables) here, `pomp` matches these names against the corresponding names attributes of vectors and passes to your native routine integer vectors which you can use to identify the correct variables. See the source code to see how this is done.

We'll specify some parameters:

```
theta <- c(
           alpha.1=0.8, alpha.2=-0.5, alpha.3=0.3, alpha.4=0.9,
           sigma.1=3, sigma.2=-0.5, sigma.3=2,
           tau=1,
           x1.0=-3, x2.0=4
           )

tic <- Sys.time()
x <- simulate(ou2,params=theta,nsim=500,seed=80073088L)
toc <- Sys.time()
print(toc-tic)
```

```
Time difference of 0.04453111 secs
```

3. The low-level interface

There is a low-level interface to `pomp` objects, primarily designed for package developers. Ordinary users should have little reason to use this interface. In this section, each of the methods that make up this interface will be introduced.

Getting initial states. The `init.state` method is called to initialize the state (unobserved) process. It takes a vector or matrix of parameters and returns a matrix of initial states.

```
data(ou2)
true.p <- coef(ou2)
x0 <- init.state(ou2)
x0
```

```
   [,1]
x1   -3
x2    4
```

```
new.p <- cbind(true.p,true.p,true.p)
new.p["x1.0",] <- 1:3
init.state(ou2,params=new.p)
```

```
   [,1] [,2] [,3]
x1    1    2    3
x2    4    4    4
```

Simulating the process model. The `rprocess` method gives access to the process model simulator. It takes initial conditions (which need not correspond to the zero-time `t0` specified when the `pomp` object was constructed), a set of times, and a set of parameters. The initial states and parameters must be matrices, and they are checked for commensurability. The method returns a rank-3 array containing simulated state trajectories, sampled at the times specified.

```
x <- rprocess(ou2,xstart=x0,times=time(ou2,t0=T),params=as.matrix(true.p))
dim(x)
```

```
[1]   2   1 101
```

```
x[,,1:5]
```

```
   [,1]      [,2]     [,3]     [,4]     [,5]
x1   -3 0.4818917 1.414307 2.430473 4.298055
x2    4 3.1569381 2.435648 1.652435 2.384673
```

Note that the dimensions of `x` are `nvars x nreps x ntimes`, where `nvars` is the number of state variables, `nreps` is the number of simulated trajectories (which is the number of columns in the `params` and `xstart` matrices), and `ntimes` is the length of the `times` argument. Note also that `x[,,1]` is identical to `xstart`.

Simulating the measurement model. The `rmeasure` method gives access to the measurement model simulator:

```
x <- x[,,-1,drop=F]
y <- rmeasure(ou2,x=x,times=time(ou2),params=as.matrix(true.p))
dim(y)
```

```
[1]   2   1 100
```

```
y[,,1:5]
```

```
        [,1]       [,2]      [,3]     [,4]     [,5]
y1 0.7813381 -0.6377216 2.6241978 4.002483 3.061009
y2 2.8435438  1.8923927 0.9671583 1.523087 2.151975
```

Process and measurement model densities. The `dmeasure` and `dprocess` methods give access to the measurement and process model densities, respectively.

```
fp <- dprocess(ou2,x=x,times=time(ou2),params=as.matrix(true.p))
dim(fp)
```

```
[1]  1 99
```

```
fp[,36:40]
```

```
[1] 0.003988565 0.023216103 0.008546941 0.001208048
[5] 0.004726450
```

```
fm <- dmeasure(ou2,y=y[,1,],x=x,times=time(ou2),params=as.matrix(true.p))
dim(fm)
```

```
[1]   1 100
```

```
fm[,36:40]
```

```
[1] 0.08650342 0.07017731 0.08825893 0.01986403 0.03125561
```

All of these are to be preferred to direct access to the slots of the pomp object, because they do error checking on the inputs and outputs.

4. Other examples

There are a number of example pomp objects included with the package. These can be found by running

```
data(package="pomp")
```

The R scripts that generated these are included in the data-R directory of the installed package. The majority of these use compiled code, which can be found in the package source.

A. A. King, Departments of Ecology & Evolutionary Biology and Mathematics, University of Michigan, Ann Arbor, Michigan 48109-1048 USA

E-mail address: kingaa at umich dot edu

URL: http:pomp.r-forge.r-project.org