ADVANCED TOPICS IN POMP

AARON A. KING

CONTENTS

1. ACCUMULATOR VARIABLES

Recall the SIR example discussed in the "Introduction to pomp" vignette. In this example, the data consist of reported cases, which are modeled as binomial draws from the true number of recoveries having occurred since the last observation. In particular, suppose the zero time for the process is t_0 and let $t_1, t_2, \ldots, t_n$ be the times at which the data $y_1, y_2, \ldots, y_n$ are recorded. Then the k-th observation $y_k = C(t_{k-1}, t_k)$ is the observed number of cases in time interval $[t_{k-1}, t_k)$. If $\Delta_{I\to R}(t_{k-1}, t_k)$ is the accumulated number of recoveries (I to R transitions) in the same interval, then the model assumes

$$y_k = C(t_{k-1}, t_k) \;\sim\; \text{binomial}(\Delta_{I\to R}(t_{k-1}, t_k), \rho)$$

where ρ is the probability a given case is actually recorded.

Now, it is easy to keep track of the cumulative number of recoveries when simulating the continuous-time SIR state process; one simply has to add each recovery to an accumulator variable when it occurs. The SIR simulator codes in the "Introduction to pomp" vignette do this, storing the cumulative number of recoveries in a state variable `cases`, so that at any time t,

$$\texttt{cases}(t) = \text{cumulative number of recoveries having occurred in the interval } [t_0, t).$$

It follows that $\Delta_{I\to R}(t_{k-1}, t_k) = \texttt{cases}(t_k) - \texttt{cases}(t_{k-1})$. Does this not violate the Markov assumption upon which all the algorithms in pomp are based? Not really. Straightforwardly, one could augment the state process, adding $\texttt{cases}(t_{k-1})$ to the state vector at time t_k. The state process would then become a *hybrid* process, with one component (the S, I, R, and `cases` variables) evolving in continuous time, while the retarded `cases` variable would update discretely.

It would, of course, be relatively easy to code up the model in this way, but because the need for accumulator variables is so common, pomp provides an easier work-around. Specifically, in the `pomp`-object constructing call to `pomp`, any variables named in the `zeronames` argument are assumed to be accumulator variables. At present, however, only the `rprocess` plug-ins and the deterministic-skeleton trajectory codes take this into account; setting `zeronames` will have no effect on custom `rprocess` codes.

Date: June 9, 2014, pomp version 0.51-2.

2. INCORPORATING NATIVE CODES USING POMPBUILDER

It's possible to use native codes for `dprocess` and for the measurement model portions of the `pomp` as well. In the "Introduction to pomp" vignette, we looked at the SIR model, which we implemented using an Euler-multinomial approximation to the continuous-time Markov process. Here, we implement a similar model using native C codes. The new, and still experimental, `pompBuilder` function helps us do this.

We'll start by writing snippets of C code to implement each of the important parts of our model. First, we encode a negative binomial measurement model.

```
## negative binomial measurement model
## E[cases|incid] = rho*incid
## Var[cases|incid] = rho*incid*(1+rho*incid/theta)
rmeas <- '
  cases = rnbinom_mu(theta,rho*incid);
'

dmeas <- '
  lik = dnbinom_mu(cases,theta,rho*incid,give_log);
'
```

Next the function that takes one Euler step.

```
## SIR process model with extra-demographic stochasticity
## and seasonal transmission
step.fn <- '
  int nrate = 6;
  double rate[nrate]; // transition rates
  double trans[nrate]; // transition numbers
  double beta; // transmission rate
  double dW; // white noise increment
  int k;

  // seasonality in transmission
  beta = beta1*seas1+beta2*seas2+beta3*seas3;

  // compute the environmental stochasticity
  dW = rgammawn(beta_sd,dt);

  // compute the transition rates
  rate[0] = mu*popsize; // birth into susceptible class
  rate[1] = (iota+beta*I*dW/dt)/popsize; // force of infection
  rate[2] = mu; // death from susceptible class
  rate[3] = gamma; // recovery
  rate[4] = mu; // death from infectious class
  rate[5] = mu;  // death from recovered class

  // compute the transition numbers
  trans[0] = rpois(rate[0]*dt); // births are Poisson
  reulermultinom(2,S,&rate[1],dt,&trans[1]);
  reulermultinom(2,I,&rate[3],dt,&trans[3]);
  reulermultinom(1,R,&rate[5],dt,&trans[5]);
```

```
  // balance the equations
  S += trans[0]-trans[1]-trans[2];
  I += trans[1]-trans[3]-trans[4];
  R += trans[3]-trans[5];
  incid += trans[3]; // incidence is cumulative recoveries
  if (beta_sd > 0.0) W += (dW-dt)/beta_sd; // increment has mean = 0, variance = dt
'
```

Now the deterministic skeleton. The "D" prepended to each state variable indicates the derivative of the state variable.

```
skel <- '
  int nrate = 6;
  double rate[nrate]; // transition rates
  double term[nrate]; // transition numbers
  double beta; // transmission rate
  double dW; // white noise increment
  int k;

  beta = beta1*seas1+beta2*seas2+beta3*seas3;

  // compute the transition rates
  rate[0] = mu*popsize; // birth into susceptible class
  rate[1] = (iota+beta*I)/popsize; // force of infection
  rate[2] = mu; // death from susceptible class
  rate[3] = gamma; // recovery
  rate[4] = mu; // death from infectious class
  rate[5] = mu;  // death from recovered class

  // compute the several terms
  term[0] = rate[0];
  term[1] = rate[1]*S;
  term[2] = rate[2]*S;
  term[3] = rate[3]*I;
  term[4] = rate[4]*I;
  term[5] = rate[5]*R;

  // assemble the differential equations
  DS = term[0]-term[1]-term[2];
  DI = term[1]-term[3]-term[4];
  DR = term[3]-term[5];
  Dincid = term[3]; // accumulate the new I->R transitions
  DW = 0;
'
```

Next, we write snippets to perform parameter transformations. Note the convention of prepending "T" to the name to signify the transformed parameter. The log-barycentric transformation is very useful when dealing with parameters (such as our initial conditions) constrained to lie on a unit simplex.

```
## parameter transformations
## note we use barycentric coordinates for the initial conditions
## the success of this depends on S0, I0, R0 being in
## adjacent memory locations, in that order
```

```
partrans <- "
  Tgamma = exp(gamma);
  Tmu = exp(mu);
  Tiota = exp(iota);
  Tbeta1 = exp(beta1);
  Tbeta2 = exp(beta2);
  Tbeta3 = exp(beta3);
  Tbeta_sd = exp(beta_sd);
  Trho = expit(rho);
  Ttheta = exp(theta);
  from_log_barycentric(&TS_0,&S_0,3);
"

paruntrans <- "
  Tgamma = log(gamma);
  Tmu = log(mu);
  Tiota = log(iota);
  Tbeta1 = log(beta1);
  Tbeta2 = log(beta2);
  Tbeta3 = log(beta3);
  Tbeta_sd = log(beta_sd);
  Trho = logit(rho);
  Ttheta = log(theta);
  to_log_barycentric(&TS_0,&S_0,3);
"
```

To model seasonality in transmission, we'll use periodic B-splines. The following constructs a covariate table with three cubic, periodic B-spline basis functions.

```
covartab <- data.frame(
                      time=seq(from=-1/52,to=10+1/52,by=1/26)
                      )

covartab <- cbind(
                  covartab,
                  with(covartab,
                       periodic.bspline.basis(
                                              x=time,
                                              nbasis=3,
                                              degree=3,
                                              period=1,
                                              names="seas%d"
                                              )
                       )
                  )
```

A call to `pompBuilder` assembles the given codes, compiles them, and links them to the current R session.

```
pompBuilder(
            name="SIR",
            data=data.frame(
              cases=NA,
              time=seq(0,10,by=1/52)
```

```
  ),
times="time",
t0=-1/52,
dmeasure=dmeas,
rmeasure=rmeas,
step.fn=step.fn,
step.fn.delta.t=1/52/20,
skeleton.type="vectorfield",
skeleton=skel,
covar=covartab,
tcovar="time",
parameter.transform=partrans,
parameter.inv.transform=paruntrans,
statenames=c("S","I","R","incid","W"),
paramnames=c(
  "gamma","mu","iota",
  "beta1","beta2","beta3","beta.sd",
  "popsize","rho","theta",
  "S.0","I.0","R.0"
  ),
zeronames=c("incid","W"),
initializer=function(params, t0, ...) {
  x0 <- setNames(numeric(5),c("S","I","R","incid","W"))
  fracs <- params[c("S.0","I.0","R.0")]
  x0[1:3] <- round(params['popsize']*fracs/sum(fracs))
  x0
}
) -> sir
```

Let's specify some parameters, simulate, and compute a deterministic trajectory:

```
coef(sir) <- c(
               gamma=26,mu=0.02,iota=0.01,
               beta1=400,beta2=480,beta3=320,
               beta.sd=0.001,
               popsize=2.1e6,
               rho=0.6,theta=10,
               S.0=26/400,I.0=0.001,R.0=1
               )

sir <- simulate(sir,seed=3493885L)
traj <- trajectory(sir,hmax=1/52)
```

The C codes generated by the above are by default written into temporary files in the R session's temporary directory. Setting `pompBuilder`'s `save` option to `TRUE` will cause them to be stored in the current working directory.

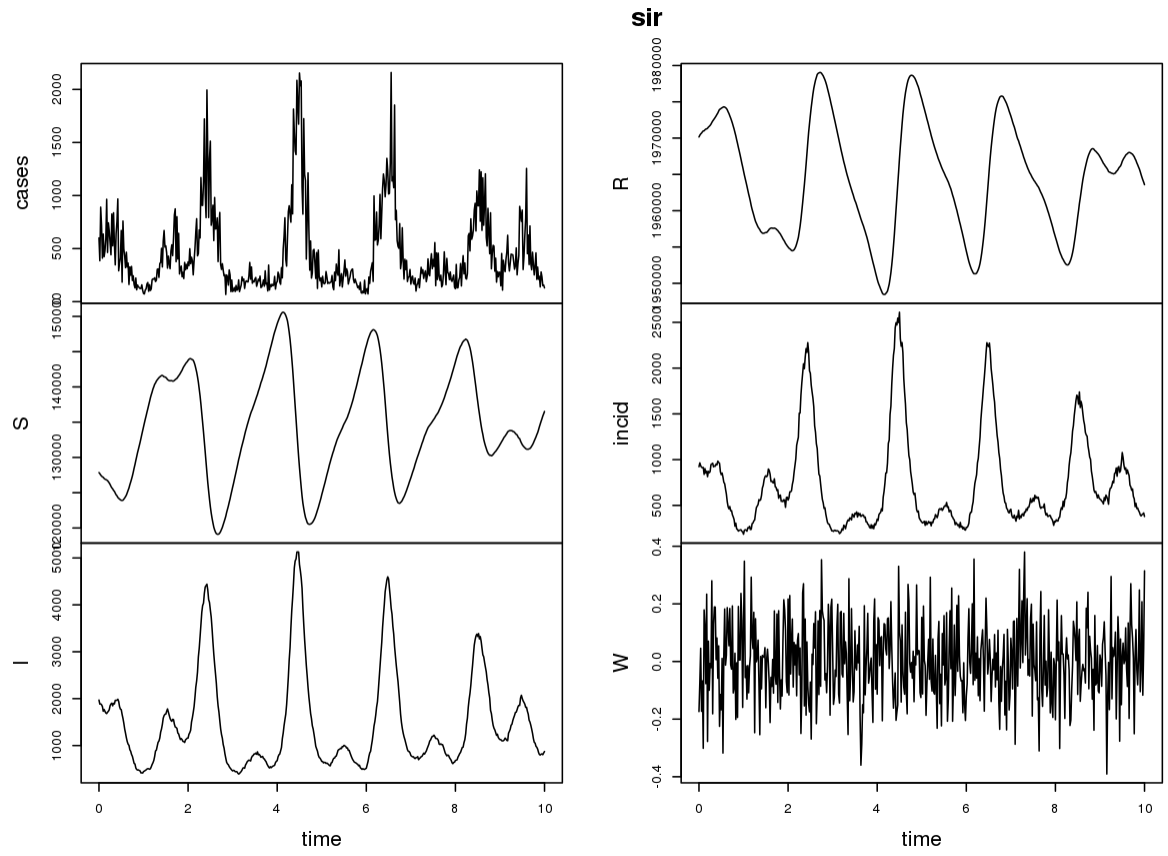

FIGURE 1. Results of `plot(sir)`.

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.

```
pompExample(ou2)
## newly created pomp object(s):
##  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=true.p)
dim(x)
## [1]   2   1 101
x[,,1:5]
##    [,1]   [,2]    [,3]   [,4]  [,5]
## x1   -3 -3.283 -0.6275  6.035 9.479
## x2    4  7.630  7.6132 10.855 2.598
```

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=true.p)
dim(y)
## [1]   2   1 100
y[,,1:5]
##      [,1]    [,2]  [,3]  [,4]   [,5]
## y1 -1.549 -0.5512  6.02 9.618  9.242
```

```
## y2  8.309  9.2891 12.27 1.584 -1.991
```

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=true.p)
dim(fp)
## [1]  1 99
fp[,36:40]
## [1] 0.006357 0.008231 0.011166 0.018466 0.015811

fm <- dmeasure(ou2,y=y[,1,],x=x,times=time(ou2),params=true.p)
dim(fm)
## [1]   1 100
fm[,36:40]
## [1] 0.118827 0.065906 0.016613 0.118275 0.008519
```

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

```
pompExample()
```

The R scripts that generated these are included in the `examples` directory of the installed package. Do

```
list.files(system.file("examples",package="pomp"))
```

The majority of these use compiled code, which can be found in the package source.

In addition, there are several interactive demos that can be instructive.

```
demo(package='pomp')
```

displays a list.

5. Different methods of implementing `rprocess`: relative performance

Here, we'll investigate various ways in which one might try to accelerate `rprocess`. We'll write a vectorized version in R code, then we'll see what the same thing looks like coded in C. We'll compare these different versions in terms of their speed at simulation. We'll also compare their performance against that of the plugins.

We'll use a discrete-time bivariate AR(1) process with normal measurement error as our example. In this model, the state process $X_t \in \mathbb{R}^2$ satisfies

$$X_t = \alpha\, X_{t-1} + \sigma\, \varepsilon_t. \tag{1}$$

The measurement process is

$$Y_t = \beta\, X_t + \tau\, \xi_t. \tag{2}$$

In these equations, α and and β are 2×2 constant matrices. ξ_t and ε_t are mutually-independent families of i.i.d. bivariate standard normal random variables. σ is a lower-triangular matrix such that $\sigma\sigma^T$ is the variance-covariance matrix of $X_{t+1}|X_t$. We'll assume that each component of X is measured independently and with the same error, τ, so that the variance-covariance matrix of $Y_t|X_t$ has τ^2 on the diagonal and zeros elsewhere.

An implementation of this model is included in the package as a `pomp` object; load it by executing `pompExample(ou2)`.

An unvectorized implementation using R code only. Before we set about vectorizing the codes, let's have a look at what a plug-in based implementation written entirely in R might look like.

```
pompExample(ou2)
## newly created pomp object(s):
##  ou2
ou2.dat <- as.data.frame(ou2)

pomp(
     data=ou2.dat[c("time","y1","y2")],
     times="time",
     t0=0,
     rprocess=discrete.time.sim(
       step.fun=function (x, t, params, ...) {
         eps <- rnorm(n=2,mean=0,sd=1) # noise terms
         xnew <- c(
                   x1=params["alpha.1"]*x["x1"]+params["alpha.3"]*x["x2"]+
                       params["sigma.1"]*eps[1],
                   x2=params["alpha.2"]*x["x1"]+params["alpha.4"]*x["x2"]+
                       params["sigma.2"]*eps[1]+params["sigma.3"]*eps[2]
                   )
         names(xnew) <- c("x1","x2")
         xnew
       }
       )
     ) -> ou2.Rplug

simdat.Rplug <- simulate(ou2.Rplug,params=coef(ou2),nsim=5000,states=T)
```

Notice how we specify the process model simulator using the `rprocess` plug-in `discrete.time.sim`. The latter function's `step.fun` argument is itself a function that simulates one realization of the process for one timestep and one set of parameters. When we vectorize the code, we'll do many realizations at once.

Vectorizing the process simulator using R code only. Now, to write a vectorized `rprocess` in R, we must write a function that simulates `nrep` realizations of the unobserved process. Each of these realizations may start at a different point in state space and each may have a different set of parameters. Moreover, this function must be capable of simulating the process over an arbitrary time interval and must be capable of reporting the unobserved states at arbitrary times in that interval. We'll accomplish this by writing an R function with arguments `xstart`, `params`, and `times`. About these inputs, we must assume:

(1) `xstart` will be a matrix, each column of which is a vector of initial values of the state process. Each state variable (matrix row) will be named.
(2) `params` will be a matrix, the columns of which are parameter vectors. The parameter names will be in the matrix column-names.
(3) `times` will be a vector of times at which realizations of the state process are required. We will have `times[k]` $\leq$ `times[k+1]` for all indices `k`, but we cannot assume that the entries of `times` will be unique.
(4) The initial states `xstart` are assumed to obtain at time `times[1]`.

This function must return a rank-3 array, which has the realized values of the state process at the requested times. This array must have rownames. Here is one implementation of such a simulator.

```
ou2.Rvect.rprocess <- function (xstart, times, params, ...) {
  nrep <- ncol(xstart)                  # number of realizations
  ntimes <- length(times)               # number of timepoints
  ## unpack the parameters (for legibility only)
  alpha.1 <- params["alpha.1",]
  alpha.2 <- params["alpha.2",]
  alpha.3 <- params["alpha.3",]
  alpha.4 <- params["alpha.4",]
  sigma.1 <- params["sigma.1",]
  sigma.2 <- params["sigma.2",]
  sigma.3 <- params["sigma.3",]
  ## x is the array of states to be returned: it must have rownames
  x <- array(0,dim=c(2,nrep,ntimes))
  rownames(x) <- rownames(xstart)
  ## xnow holds the current state values
  x[,,1] <- xnow <- xstart
  tnow <- times[1]
  for (k in seq.int(from=2,to=ntimes,by=1)) {
    tgoal <- times[k]
    while (tnow < tgoal) {              # take one step at a time
      eps <- array(rnorm(n=2*nrep,mean=0,sd=1),dim=c(2,nrep))
      tmp <- alpha.1*xnow['x1',]+alpha.3*xnow['x2',]+
        sigma.1*eps[1,]
      xnow['x2',] <- alpha.2*xnow['x1',]+alpha.4*xnow['x2',]+
        sigma.2*eps[1,]+sigma.3*eps[2,]
      xnow['x1',] <- tmp
      tnow <- tnow+1
    }
    x[,,k] <- xnow
  }
  x
}
```

We can put this into a pomp object that is the same as `ou2.Rplug` in every way except in its `rprocess` slot by doing

```
ou2.Rvect <- pomp(ou2.Rplug,rprocess=ou2.Rvect.rprocess)
```

Let's pick some parameters and simulate some data to see how long it takes this code to run.

```
theta <- c(
           x1.0=-3, x2.0=4,
           tau=1,
           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
           )


simdat.Rvect <- simulate(ou2.Rvect,params=theta,states=T,nsim=100000)
```

Doing 100000 simulations of `ou2.Rvect` took 3.45 secs. Compared to the 9.16 secs it took to run 5000 simulations of `ou2.Rplug`, this is a 53-fold speed-up.

Accelerating the code using C: a plug-in based implementation. As we've seen, we can usually acheive big accelerations using compiled native code. A one-step simulator written in C for use with the `discrete.time.sim` plug-in is included with the package and can be viewed by doing

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

The one-step simulator is in function `ou2_step`. Prototypes for the one-step simulator and other functions are in the `pomp.h` header file; view it by doing

```
file.show(file=system.file("include/pomp.h",package="pomp"))
```

We can put the one-step simulator into the `pomp` object and simulate as before by doing

```
ou2.Cplug <- pomp(
                  ou2.Rplug,
                  rprocess=discrete.time.sim("ou2_step"),
                  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")
                  )

simdat.Cplug <- simulate(ou2.Cplug,params=theta,states=T,nsim=100000)
```

Note that `ou2_step` is written in such a way that we must specify `paramnames`, `statenames`, and `obsnames`. These 100000 simulations of `ou2.Cplug` took 1.84 secs. This is a 100-fold speed-up relative to `ou2.Rplug`.

A vectorized C implementation. The function `ou2_adv` is a fully vectorized version of the simulator written in C. View this code by doing

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

This function is called in the following `rprocess` function. Notice that the call to `ou2_adv` uses the `.C` interface.

```
ou2.Cvect.rprocess <- function (xstart, times, params, ...) {
  nvar <- nrow(xstart)
  npar <- nrow(params)
  nrep <- ncol(xstart)
  ntimes <- length(times)
  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))
           )$X,
        dim=c(nvar,nrep,ntimes),
        dimnames=list(rownames(xstart),NULL,NULL)
        )
```

```
}
```

The call that constructs the `pomp` object is:

```
ou2.Cvect <- pomp(
                  ou2.Rplug,
                  rprocess=ou2.Cvect.rprocess
                  )
paramnames <- c(
                "alpha.1","alpha.2","alpha.3","alpha.4",
                "sigma.1","sigma.2","sigma.3",
                "tau",
                "x1.0","x2.0"
                )

simdat.Cvect <- simulate(ou2.Cvect,params=theta[paramnames],
                         nsim=100000,states=T)
```

Note that we've had to rearrange the order of parameters here to ensure that they arrive at the native codes in the right order. Doing 100000 simulations of `ou2.Cvect` took 2.15 secs, a 85-fold speed-up relative to `ou2.Rplug`.

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