ADVANCED TOPICS IN POMP

AARON A. KING

Contents

1. Accelerating your codes: vectorizing `rprocess` and using native codes 1
2. Accumulator variables 8
3. The low-level interface 9
4. Other examples 10

This document discusses accelerating pomp by vectorizing your codes and/or using native (C or FORTRAN) codes. It also introduces pomp's low-level interface for code developers.

1. Accelerating your codes: vectorizing `rprocess` and using native codes

In the "Introduction to pomp" vignette, we used *plug-ins* provided by the package to specify the `rprocess` component of partially-observed Markov process models. The `rprocess` plug-ins require you to write a simulator for a single realization of the process, for a single set of parameters, from one time to another. pomp then calls this code many times—using potentially many different parameter values, states, and times—whenever it simulates the process, computes likelihood via Monte Carlo integration, etc. The inference methods implemented in pomp are quite computationally intensive, which puts a premium on the speed of your codes. Sometimes, you can realize substantial speed-up of your code by vectorizing it. This necessitates foregoing the relative simplicity of the plug-in-based implementation and writing `rprocess` "from scratch". Here, we'll develop a vectorized version of `rprocess` 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 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 `data(ou2)`.

Date: April 14, 2012, pomp version 0.41-5.

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.

```
data(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
```

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
           )

tic <- Sys.time()
simdat.Rvect <- simulate(ou2.Rvect,params=theta,states=T,nsim=1000)
toc <- Sys.time()
etime.Rvect <- toc-tic
```

Doing 1000 simulations of `ou2.Rvect` took 0.1 secs. Compared to the 2.32 secs it took to run 1000 simulations of `ou2.Rplug`, this is a 23-fold speed-up.

Using R's byte compiler. From version 2.13, R has provided byte-compilation facilities, in the base package `compiler`. Let's see to what extent we can speed up our codes by byte-compiling the components of our `pomp` object.

```
require(compiler)
pomp(
     ou2.Rplug,
     rprocess=discrete.time.sim(
       step.fun=cmpfun(
         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
         },
         options=list(optimize=3)
         )
       )
     ) -> ou2.Bplug
```

Doing these 1000 simulations of `ou2.Bplug` took 1.72 secs. This is a 1.4-fold speed-up relative to the plug-in code written in R.

We can byte-compile the vectorized R code, too, and compare its performance:

```
ou2.Bvect <- pomp(ou2.Rplug,rprocess=cmpfun(ou2.Rvect.rprocess,options=list(optimize=3)))

tic <- Sys.time()
simdat.Bvect <- simulate(ou2.Bvect,params=theta,states=T,nsim=1000)
toc <- Sys.time()
etime.Bvect <- toc-tic
```

This code shows a 28-fold speed-up relative to the plug-in code written in R.

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")
                )

tic <- Sys.time()
simdat.Cplug <- simulate(ou2.Cplug,params=theta,states=T,nsim=100000)
toc <- Sys.time()
etime.Cplug <- toc-tic
```

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 1.87 secs. This is a 124-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` convention.

```
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
                  )

tic <- Sys.time()
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)
toc <- Sys.time()
etime.Cvect <- toc-tic
```

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 2.22 secs, a 105-fold speed-up relative to `ou2.Rplug`.

More on native codes and plug-ins. 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 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,
       PACKAGE="pomp"
       ),
     ## native routine for the skeleton:
     skeleton.type="vectorfield",
     skeleton="_sir_ODE",
     ## binomial measurement model:
     rmeasure="_sir_binom_rmeasure",
     dmeasure="_sir_binom_dmeasure",
     ## name of the shared-object library containing the
     ## native measurement-model 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","logbeta1","beta.sd",
       "pop","rho","nbasis","degree","period"
       ),
     ## designate 'cases' as an accumulator variable
     ## i.e., set it to zero after each observation
     zeronames=c("cases"),
     comp.names=c("S","I","R"),
     parameter.transform="_sir_par_trans",
     parameter.inv.transform="_sir_par_untrans",
     initializer=function(params, t0, comp.names, ...) {
       ic.names <- paste(comp.names,".0",sep="")
       snames <- c("S","I","R","cases","W")
```

```
  fracs <- params[ic.names]
  x0 <- numeric(length(snames))
  names(x0) <- snames
  x0[comp.names] <- round(params['pop']*fracs/sum(fracs))
  x0["cases"] <- 0
  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"))
```

In the `demo` directory is an R script that shows how to compile `sir.c` into a shared-object library and link it with R. Do `demo(sir)` to run and view this script. 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.

You can also use the R package inline to put C or FORTRAN codes directly into your R functions.

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 to these 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`, `covarnames`, and `obsnames` arguments to `pomp` come in: use these arguments to specify the order in which your C code expects to see the parameters, state variables, covariates, and observables (data variables). `pomp` will match these names against the corresponding names attributes of vectors. It will then pass to your native routines index vectors you can use to locate the correct variables. See the source code to see how this is done.

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

```
params <- c(
            gamma=26,mu=0.02,iota=0.01,
            logbeta1=log(1200),logbeta2=log(1800),logbeta3=log(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(params,nbasis=3,degree=3,period=1),seed=3493885L)
sims <- simulate(sir,nsim=10,obs=T)
traj <- trajectory(sir,hmax=1/52)
```

2. 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.

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=true.p)
 dim(x)
```

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

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

```
   [,1]      [,2]    [,3]     [,4]     [,5]
x1   -3 -1.306265 5.48259 4.582667 2.419212
x2    4  3.382165 5.56973 3.992289 4.956622
```

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 -2.923222 4.994905 5.790538 1.778335 0.844486
y2  2.463450 3.556510 3.594183 5.468858 5.177477
```

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] 0.020375721 0.024636584 0.009769575 0.022778334
[5] 0.012506434
```

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

```
[1]   1 100
```

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

```
[1] 0.008567435 0.106605182 0.042264109 0.103815061
[5] 0.118806689
```

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`