BDSEM Package: inferring MLE rates of discretely observed birth-death-shift processes

Jason Xu

October 29, 2014

1 Transition probabilities of birth-death-shift process

Here we demonstrate how to use the software to obtain and verify the accuracy of numerically computed transition probabilities of the BDS process. We walk through an illustration that is an analogous, simplified version of the transition probability comparison experiment in the main paper.

First, we specify desired birth, shift, and death rates λ, ν, μ as inputs. Then given an initial population size i, corresponding to an initial state of the process $\mathbf{X}(0) = (i, 0)$, we want to compute the transition probability $p_{(i,0),(k,l)(t)}$ accurately for any time interval length t. We do this for a list of different interval lengths using the function `getTrans.timeList`. This function implements our generating function approach to computing transition probabilities, and in addition to the arguments already mentioned, this requires inputs `s1.seq`, `s2.seq`, vectors of imaginary numbers evenly spaced around the unit circle. Their length determines the maximum k, l values for which transition probabilities are computed; accuracy improves with finer sequences, and their length defined by `gridLength` below should be a power of 2 for best results.

```
library(bdsem)
lam = .02; v = .008; mu = .015
gridLength = 16
s1.seq <- exp(2*pi*1i*seq(from = 0, to = (gridLength-1))/gridLength)
s2.seq <- exp(2*pi*1i*seq(from = 0, to = (gridLength-1))/gridLength)
tList <- c(.5,2,6,10)
transProbs <- getTrans.timeList(tList,
                                lam, v, mu, 10, s1.seq, s2.seq, .5)
```

The results are stored in `transProbs`, a list of matrices, where each entry contains a matrix corresponds to a time interval length from `tList`. For instance, in our example the transition probabilities corresponding to $t = 6$ are contained in the matrix shown below, whose first 10 rows and columns are shown rounded to 3 digits:

```
round(transProbs[[3]][1:10, 1:10], 3)

##        [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8] [,9] [,10]
##  [1,] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000    0     0
##  [2,] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000    0     0
##  [3,] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000    0     0
##  [4,] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000    0     0
##  [5,] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000    0     0
##  [6,] 0.000 0.001 0.001 0.001 0.001 0.000 0.000 0.000    0     0
##  [7,] 0.002 0.006 0.007 0.006 0.003 0.001 0.001 0.000    0     0
##  [8,] 0.011 0.026 0.028 0.019 0.009 0.004 0.001 0.000    0     0
##  [9,] 0.039 0.075 0.067 0.039 0.017 0.006 0.002 0.001    0     0
## [10,] 0.084 0.125 0.092 0.046 0.018 0.006 0.002 0.000    0     0
```

The k, l entry of the matrix corresponds to the transition probability $p_{(i,0),(k-1,l-1)(t)}$. For example, the specific transition probability $p_{(10,0),(8,0)}(6)$ can be looked up

```
transProbs[[3]][9, 1]

## [1] 0.03945
```

Next, we can check these probabilities by simulating from the BDS process. We simulate `nSims` realizations of the process for length t, for all values t in `tList`. Each simulation begins at $\mathbf{X}(0) = (i, 0)$, and transition probabilities $p_{(i,0),(k,l)(t)}$ are computed by empirically computing the proportion of times the simulation ends with $\mathbf{X}(t) = (k, l)$.

These empirical transition probabilities via simulation serve as a ground truth benchmark for comparison, and are computed using `getTrans.MC`. The results are stored in the same format as returned by `getTrans.timeList`.

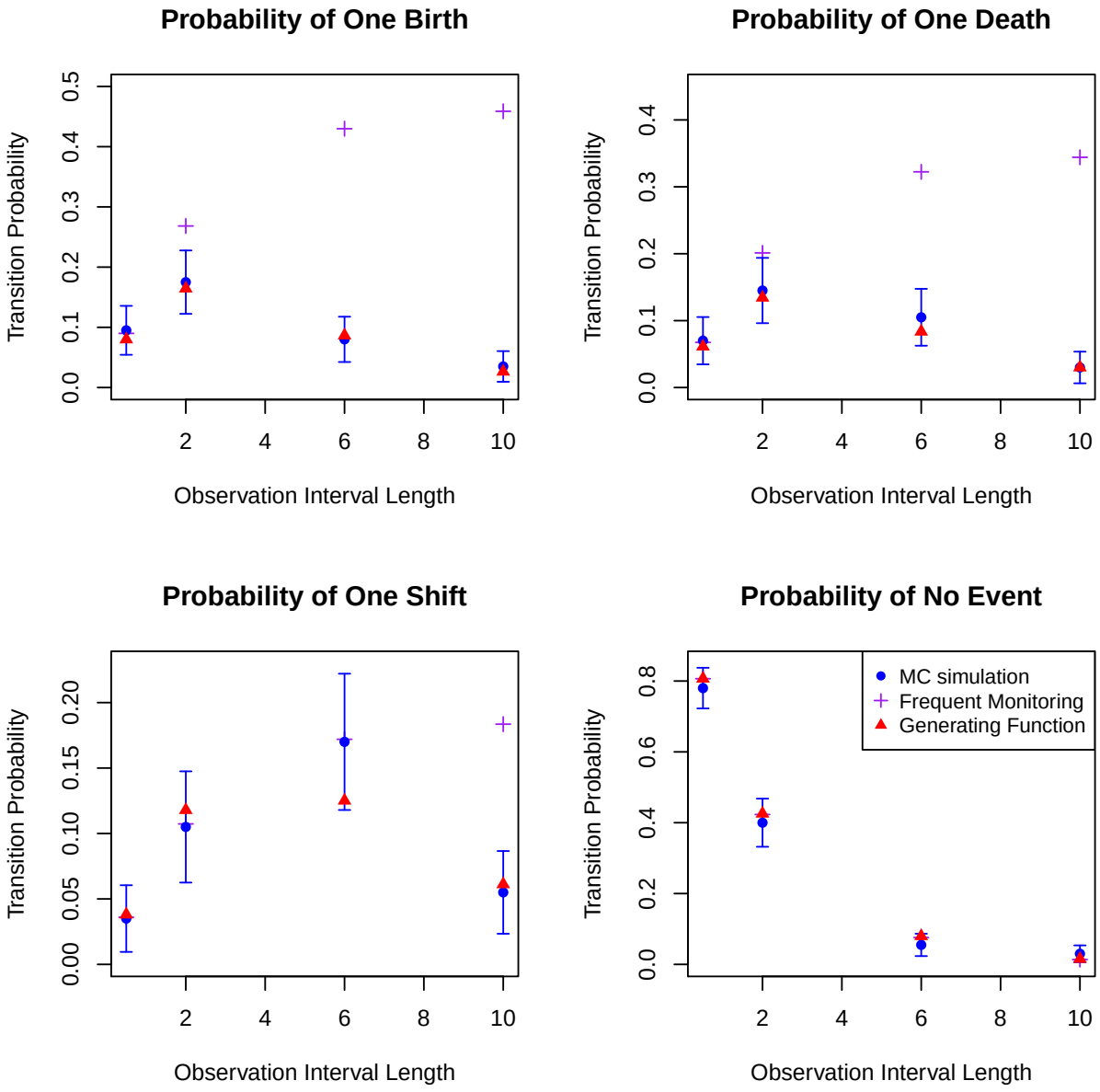
To compare with previous methods, we can also compute the probabilities of one birth, death, or shift, as well as the probability of no event occurring, according to the frequent monitoring method via `getTrans.FreqMon`:

```
nSims = 200  # further increase number of simulations for more precise estimates
# nSims = 10 # for speed of compilation, should increase
tp.MC <- getTrans.MC(nSims, tList, lam, v, mu, 10)
tp.FM <- getTrans.FreqMon(tList, lam, v, mu, 10)
```

It is easy to compare corresponding entries of `transProbs` and `tp.MC` to verify that our method calculates accurate values, although you must increase `nSims` for accurate Monte Carlo estimates. We can also plot the transition probabilities defined under frequent monitoring as computed in all three cases, similar to the more detailed Figure 2 in the main paper. Again, the Monte Carlo results may be inaccurate without increasing `nSims` in our example to an adequately large number; you are encouraged to re-run the above code with large `nSims` to recreate the results in the main paper.

```
#store results in more intuitive formats
noEvent <- birthProbs <- shiftProbs <- deathProbs <- c()
sd.nothing <- sd.birth <- sd.shift <- sd.death <- c()

for(i in 1:length(tList)){
  death <- rbind( tp.MC[[i]][10,1], transProbs[[i]][10,1], tp.FM[[i]][1,1])
  shift <- rbind( tp.MC[[i]][10,2], transProbs[[i]][10,2], tp.FM[[i]][1,2])
  birth <- rbind( tp.MC[[i]][11,2], transProbs[[i]][11,2], tp.FM[[i]][2,2])
  nothing <- rbind( tp.MC[[i]][11,1], transProbs[[i]][11,1], tp.FM[[i]][2,1])

  sd.death <- c(sd.death, sqrt(death[1]*(1 - death[1])/nSims) )
  sd.shift <- c(sd.shift, sqrt(shift[1]*(1 - shift[1])/nSims) )
  sd.birth <- c(sd.birth, sqrt(birth[1]*(1 - birth[1])/nSims) )
  sd.nothing <- c(sd.nothing, sqrt(nothing[1]*(1 - nothing[1])/nSims) )

  deathProbs <- cbind(deathProbs, death)
  shiftProbs <- cbind(shiftProbs, shift)
  birthProbs <- cbind(birthProbs, birth)
  noEvent <- cbind(noEvent, nothing)
}
```

```
## Warning:  package 'plotrix' was built under R version 2.15.3
```



2 Inference in simple BDS model

This section demonstrates how to infer birth, shift, and death rates that do not depend on covariates using our method, and can be used to recreate the second simulation study (top panel, Figure 3 of main paper) comparing inference when the likelihood is built up using transition probabilities computed using our method to the setting where frequent monitoring transition probabilities are used.

As discussed in the paper, we generate a partially observed dataset and infer rates using both methods. We simulate from the BSD process with parameters $\lambda = .07, \mu = .12, \nu = .2$ and record 200 discretely observed states of the process evenly spaced dt

time units apart. Each simulated interval begins with an initial population size drawn uniformly between 1 and 15, and this data generating process is repeated three times, producing three datasets corresponding to spacings $dt = (.2, .4, .6)$. The code below initializes these settings:

```
# list of time interval lengths and list of possible initial sizes
tList <- c(0.2, 0.4, 0.6)
initList <- c(1:15)
lam = 0.07
v = 0.02
mu = 0.11
trueParam <- c(lam, v, mu)  #simple rates, no covariates
N = 200  #number of intervals to generate per dataset

# similarly to previous example, we need to specify the grid of inputs
gridLength = 16
s1.seq <- exp(2 * pi * (0+1i) * seq(from = 0, to = (gridLength - 1))/gridLength)
s2.seq <- exp(2 * pi * (0+1i) * seq(from = 0, to = (gridLength - 1))/gridLength)
```

The entire process of generating data and MLE inference on the synthetic dataset using our method is then completed using the function `FFT.replicate`. This function calls `makedata.simple` to generate the three synthetic datasets, and optimizes the likelihood using `optim`. Because we want to compare directly to the frequent monitoring method which does not have a corresponding EM algorithm, here we perform *generic optimization*. An example of inference using the EM algorithm follows in the next sections. We specify necessary parameters for the algorithm described in the transition probability example above below, as well as the number of repetitions below: the code is not evaluated here because it takes some time to run.

```
FFT.rep = 200  #number of replications
FFTruns <- FFT.replicate(FFT.rep, N, tList,
                         lam, v, mu, initList, trueParam, s1.seq, s2.seq)
```

The object returned by FFTruns is a rather unintuitive array, and the following code simply stores parameter estimates and 95% confidence interval coverage in a more intuitive list and matrix, respectively. The columns of the coverage matrix correspond to the birth, shift, and death rate respectively, and the rows correspond to the times in `tList`.

```
#the object returned by FFTruns is not intuitive, and the following code stores
#parameter estimates and 95% confidence interval coverage in matrices a more
#readable format:
FFT.params <- vector("list",length(tList))
FFT.coverage <- mat.or.vec(3,3)
for(i in 1:length(tList)){
  FFT.params[[i]] <- sapply(FFTruns[i,], getEst)
```

```
  coverMatrix <- sapply(FFTruns[i,], getCover, trueParam = trueParam)
  FFT.coverage[i,] <- apply(coverMatrix, 1, mean)
}
```

Similarly, we may repeat the same steps using the frequent monitoring likelihood by then running the code below:

```
FM.rep = 200
FMruns <- FM.replicate(FM.rep, N, tList, lam, v, mu, initList, trueParam)

FM.params <- vector("list", length(tList))
FM.coverage <- mat.or.vec(3, 3)  #each row corresponds to a dt,
# each column corresponds to a parameter coverage
for (i in 1:length(tList)) {
    FM.params[[i]] <- sapply(FMruns[i, ], getEst)
    coverMatrix <- sapply(FMruns[i, ], getCover, trueParam = trueParam)
    FM.coverage[i, ] <- apply(coverMatrix, 1, mean)
}
```

Finally, we include some example plotting code to produce boxplots that clearly illustrate the advantage in using our method to compute transition probabilities. This code is not evaluated here, but should be used after running the previous experiment.

```
pdf('boxPlots.pdf', width = 11, height = 4 )
par(mfrow=c(1,3))
birthplot = boxplot(xaxt = 'n', cex.lab = 1.3, ylab = "Estimate",
                    ylim = c(.02,.127), FFT.params[[1]][1,],
                    FM.params[[1]][1,], FFT.params[[2]][1,],
                    FM.params[[2]][1,], FFT.params[[3]][1,],
                    FM.params[[3]][1,], col = c(0,"grey",0,"grey",0,"grey"),
                    main = "Birth rate estimate",cex.axis = 1.5,
                    cex.main = 2.0, boxwex = .9)
abline(h = lam, lty = 2)
axis(1, at=c(1.5,3.5,5.5), labels=tList, cex.axis = 1.5)
text(c(1,3,5),.123, FFT.coverage[,1], cex = 1.5)
text(c(2,4,6),.123, FM.coverage[,1], cex = 1.5)

deathplot = boxplot(xaxt = "n", cex.lab = 2, xlab = "Time Interval Length",
                    ylim = c(.04,.202), FFT.params[[1]][3,],
                    FM.params[[1]][3,], FFT.params[[2]][3,],
                    FM.params[[2]][3,],  FFT.params[[3]][3,],
                    FM.params[[3]][3,], col = c(0,"grey",0,"grey",0,"grey"),
                    main = "Death rate estimate", cex.main = 2.0,
                    boxwex = .9, cex.axis = 1.5)
axis(1, at=c(1.5,3.5,5.5), labels=tList, cex.axis = 1.5)
```

```
abline(h = mu, lty = 2)
text(c(1,3,5),.197, FFT.coverage[,3], cex = 1.5)
text(c(2,4,6),.197, FM.coverage[,3], cex = 1.5)

shiftplot = boxplot(xaxt = 'n', ylim = c(0,.08), FFT.params[[1]][2,],
                    FM.params[[1]][2,], FFT.params[[2]][2,],
                    FM.params[[2]][2,], FFT.params[[3]][2,],
                    FM.params[[3]][2,], col = c(0,"grey",0,"grey",0,"grey"),
                    main = "Shift rate estimate",
                    cex.main = 2.0, cex.axis = 1.5, boxwex = .9)
abline(h = v, lty = 2)
axis(1, at=c(1.5,3.5,5.5), labels=tList, cex.axis = 1.5)
text(c(1,3,5),.076, FFT.coverage[,2], cex = 1.5)
text(c(2,4,6),.076, FM.coverage[,2], cex = 1.5)
legend("bottomright",legend=c("FFT", "FM"), pch = c(0,15),
       pt.cex = 1.7, cex = 1.2, col=c(1, "grey"))

dev.off()
```

3 Verifying Restricted Moments

Next, we compare our restricted moment calculations to their corresponding Monte Carlo estimates. We check that the expected birth calculation is accurate, for instance, by verifying the equality

$$E(N_t^+|X_0 = i,j) = \sum_{k,l} E(N_t^+, 1_{x_t=kl}|x_0 = i,j) : \tag{1}$$

The left hand side is verified via Monte Carlo simulation, simply counting the number of births per simulation of the process for t time units, and averaging over realizations. The right hand side entries are those quantities computed using our generating functions in the main theorem of the paper: we compute these over a range of i,j values that comprises most of the support, and sum them to compare with the Monte Carlo averages.

The code below recreates a similar, smaller scale version of Figure 2 in the appendix: again, Monte Carlo results are improved by increasing `numSims` below. Begin by initializing parameters and computing Monte Carlo empirical estimates via the function `sim.N.eventcount`:

```
lam = 3*.0188; v = 3*.0026;  mu = 3*.0147 #increase rates here so there are not
#too many realizations with zero events, which thus requires more MC simulations
initNum = 10; param <- c(lam,v,mu)
numSims = 200 #in practice, increase number of MC simulations
#numSims = 10    # to compile quickly; use at least 100
```

```
tList <- c(.5,1,3,5)

#compute Monte Carlo averages and empirical confidence intervals here
eventsMC <- sdsMC <- eventsFFT <- lMC <- uMC <- mat.or.vec(4, length(tList))
for(i in 1:length(tList)){
  results = sim.N.eventcount(numSims, tList[i], lam, v, mu, initNum)
  eventsMC[,i] <- rowMeans(results)
  sdsMC[,i] <- apply(results,1,sd)
  lMC[,i] <- apply(results,1,quantile, probs = .025 )  #empirical Monte Carlo 95% CI
  uMC[,i] <- apply(results, 1, quantile, probs = .975)
}
```

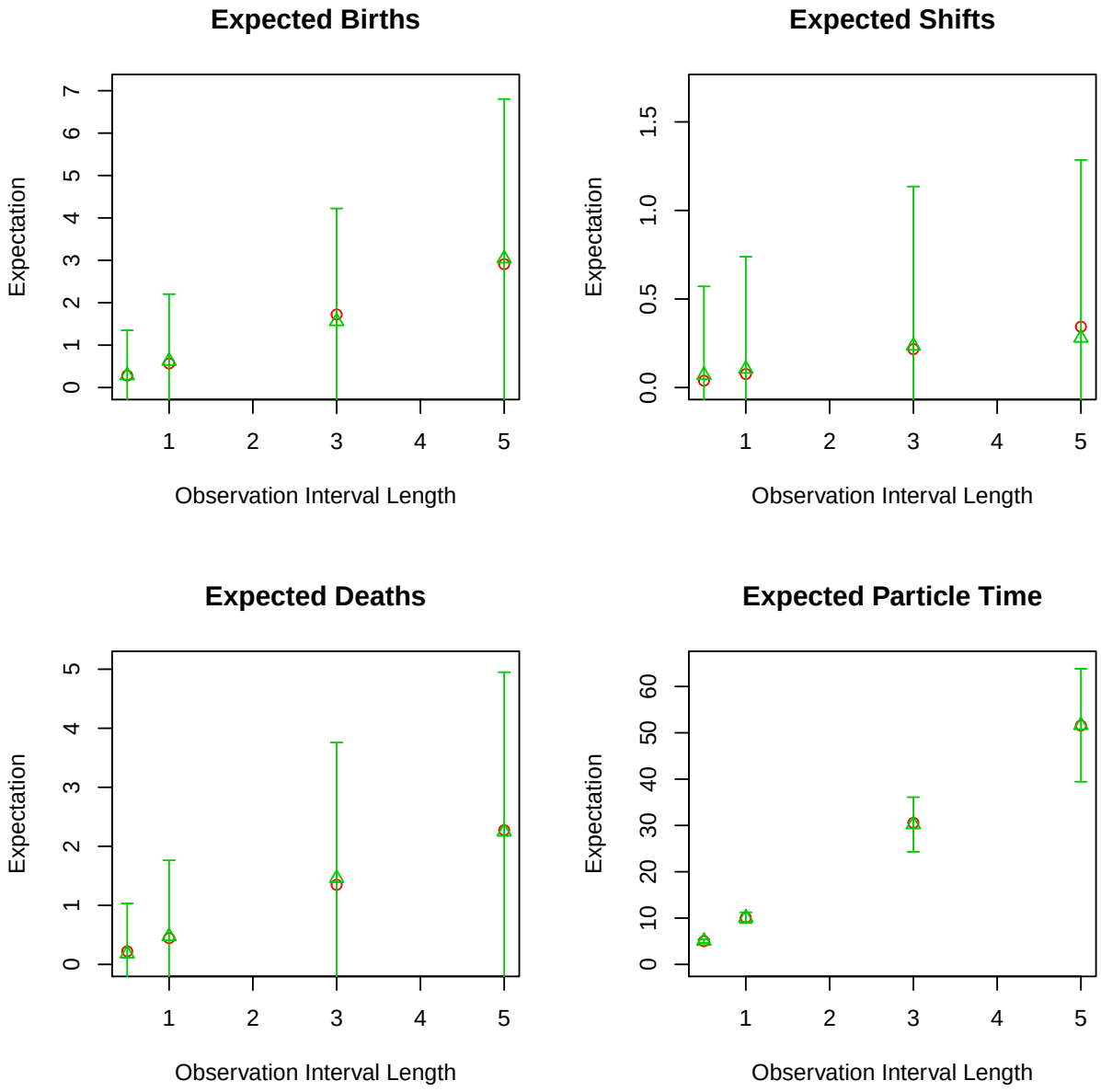
Now we compute the restricted moments using our method: this step involves solving an ordinary differential equation numerically for each (i, j) pair, where i, j are entries in `s1.seq, s2.seq`, and then taking the fast Fourier transform of the result:

```
#these are lists of matrices of restricted means given ending count,
#list index corresponds to entries of tlist
bmeans <- getBirthMeans.timeList(tList, lam, v, mu, initNum, s1.seq, s2.seq, .5)
smeans <- getShiftMeans.timeList(tList, lam, v, mu, initNum, s1.seq, s2.seq, .5)
dmeans <- getDeathMeans.timeList(tList, lam, v, mu, initNum, s1.seq, s2.seq, .5)
ptimes <- getParticleT.timeList (tList, lam, v, mu, initNum, s1.seq, s2.seq, .5)
```

The result stored in each of the four entries above is in a similar format as `transProbs` from the first example, containing restricted means instead of probabilities. Next, we sum the entries in each of these restricted mean matrices (the right hand side of equation 1), and plot results alongside MC estimates:

```
for(i in 1:length(tList)){
  eventsFFT[,i] <- c( sum(bmeans[[i]]), sum(smeans[[i]]),
                      sum(dmeans[[i]]), -sum(ptimes[[i]]) )
}

#pdf('restrictedMoments.pdf')
par(mfrow=c(2,2))
plot(tList, eventsFFT[1,], pch = 1, col = 2, ylim = c(0,7.1),
     ylab = "Expectation", xlab = "Observation Interval Length",
     main = "Expected Births")
plotCI(tList, eventsMC[1,], pch = 2, col = 3,
       ui=eventsMC[1,]+1.96*sdsMC[1,], li=eventsMC[1,]-1.96*sdsMC[1,], add = TRUE)

plot(tList, eventsFFT[2,], pch = 1, col = 2,ylim = c(0,1.7),
     ylab = "Expectation", xlab = "Observation Interval Length",
     main = "Expected Shifts")
plotCI(tList, eventsMC[2,], pch = 2, col = 3,
```

```
       ui=eventsMC[2,]+1.96*sdsMC[2,], li=eventsMC[2,]-1.96*sdsMC[2,], add = TRUE)

plot(tList, eventsFFT[3,], pch = 1, col = 2, ylim = c(0,5.1),
     ylab = "Expectation", xlab = "Observation Interval Length",
     main = "Expected Deaths")
plotCI(tList, eventsMC[3,], pch = 2, col = 3,
       ui=eventsMC[3,]+1.96*sdsMC[3,], li=eventsMC[3,]-1.96*sdsMC[3,], add = TRUE)

plot(tList, eventsFFT[4,], pch = 1, col = 2, ylim = c(0,65),
     ylab = "Expectation", xlab = "Observation Interval Length",
     main = "Expected Particle Time")
plotCI(tList, eventsMC[4,], pch = 2, col = 3,
       ui=eventsMC[4,]+1.96*sdsMC[4,], li=eventsMC[4,]-1.96*sdsMC[4,], add = TRUE)
#dev.off()
```



After increasing the number of Monte Carlo simulations, you should verify that these computations under the hood of the EM algorithm are indeed accurate.

4 EM algorithm for inference in covariate-dependent BDS process

We conclude this vignette with a tutorial on using our software for recovering the coefficient vector in the setting where rates depend on multiple covariates using the EM algorithm we developed. In this example, we create multiple independent “patients" whose characteristics determine the patient-specific rates of their corresponding BDS processes. Recall from the main paper that rates of each process are determined by a vector of c

covariates $\mathbf{z}_p = (z_{p,1}, z_{p,2}, \ldots, z_{p,c}) \in \mathbb{R}^c$ through a log-linear model:

$$\log(\lambda_p) = \boldsymbol{\beta}^{\lambda} \cdot \mathbf{z_p}, \log(\nu_p) = \boldsymbol{\beta}^{\nu} \cdot \mathbf{z_p}, \log(\mu_p) = \boldsymbol{\beta}^{\mu} \cdot \mathbf{z_p}, \tag{2}$$

where $\boldsymbol{\beta} := (\boldsymbol{\beta}^{\lambda}, \boldsymbol{\beta}^{\nu}, \boldsymbol{\beta}^{\mu})$ are the regression coefficients and $\cdot$ represents a vector product. Note that inference for the simple BDS process that we studied using general optimization is possible by setting $\mathbf{z}_p$ to be only a vector of 1's, and $\boldsymbol{\beta} = \lambda, \nu, \mu$.

Following the simulation study in section 3.2 of the paper, we begin by specifying the true coefficient vector with 4 coefficients per rate:

```
beta.lam <- c(log(7.5), log(0.5), log(0.3), log(3))
beta.v <- c(log(0.5), log(8), log(0.5), log(0.9))
beta.mu <- c(log(4), log(0.3), log(0.8), log(0.9))
betas <- c(beta.lam, beta.v, beta.mu)  #true parameters
```

Next, we generate vectors of patient covariates uniformly according to the code below, and create a matrix containing the individual rates using `MakePatientRates`: rows correspond to the birth, shift, and death rates, with columns corresponding to individual patients.

```
num.patients <- 20  #again, recommend at least 100
x1 <- runif(num.patients, 0, 2)
x2 <- runif(num.patients, 6, 10)
x3 <- runif(num.patients, 4, 6)
patients.design <- rbind(1, x1, x2, x3)
patients.rates <- MakePatientRates(patients.design, betas)
patients.rates[, 1:5]  #show for first 5 patients

##         [,1]    [,2]    [,3]    [,4]     [,5]
## [1,] 0.01722 0.02546 0.02722 0.49647 0.058731
## [2,] 0.06699 0.06113 0.01699 0.02875 0.001861
## [3,] 0.04677 0.03394 0.11390 0.15807 0.288085

apply(patients.rates, 1, mean)

## [1] 0.23254 0.02191 0.18101
```

Now we are able to generate the discretely observed dataset using `MakePatientData`. Details are also available in the help file, but briefly, the simulated observation intervals are spaced `t.pat` units apart, and between 2 and 7 observations per patient are created, with initial population uniformly drawn between 2 and 14.

```
t.pat <- 0.2
PATIENTDATA <- MakePatientData(t.pat, num.patients, patients.rates)
PATIENTDATA[, 1:8]
```

```
##              [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## i               1    1    1    2    2    2    3    3
## init.patient    2    2    2    6    6    6   10   10
##                 1    3    3    7    6    5   11    9
##                 1    3    3    7    6    5   10    9
##                 0    0    0    0    0    0    0    0
```

The first row corresponds to a patient ID, and the last three rows contain information about the state of the process $\mathbf{X}(t)$ at the endpoints of the observation interval. This is detailed in the help file, but note that the second row contains information about the average initial number of particles used to simulate data for that patient. This information is not used at any stage in the EM algorithm. When running the algorithm on a real dataset, we first convert the data to this format, but we may replace the second row with the observation interval lengths since they will generally not be evenly spaced and are necessary information to the E-step. Then, we simply replace the function `ESTEP` with `ESTEP.realdata`, which replaces the constant interval length supplied by argument `t.pat` by the information supplied in the second row of `PATIENTDATA`.

Finally, running the EM algorithm to recover the coefficients given data generated in this format is straightforward after specifying a few more settings and using the function `EM.run`. The following code takes some time to run, especially as we increase the number of simulated patients, and is not evaluated here. You are encouraged to run the code and verify the output over many simulations from different initial guesses.

We note that the algorithm is numerically sensitive in the sense that initial guesses which are ill-suited for the corresponding dataset may lead to overflow/underflow in numerical computations. Similarly, intermediate steps may generate warning messages from methods in `deSolve`. In our simulation studies included in paper, roughly half of the initial guesses fail, but our algorithm is robust in the sense that those that do converge result at the same solution. Similar issues may arise in black box optimization, but as we show in the paper, even using relatively robust standard optimization methods such as Nelder-Mead lead to a disparate range of results at convergence, which is highly undesirable.

```
gridLength = 32 #can set to 32 also
s1.seq <- exp(2*pi*1i*seq(from = 0, to = (gridLength-1))/gridLength)
s2.seq <- exp(2*pi*1i*seq(from = 0, to = (gridLength-1))/gridLength)

#perturb around true values for initial guess:
betaInit <- as.vector(rmvnorm(1, mean = betas, sigma = diag(.1*abs(betas) )))

relTol = .0000001 #set relative convergence criteria

#Run EM algorithm
EMrun <- EM.run(betaInit, t.pat, num.patients,
                PATIENTDATA, patients.design, s1.seq, s2.seq, relTol)
#post-hoc numerical confidence interval computation, a slow function
```

```
#sigma <- getStandardErrors(EMrun$betaEstimate, t.pat,
#                            num.patients, PATIENTDATA, patients.design, s1.seq, s2.seq)
#upper95 <-EMrun$betaEstimate +1.96*sigma
#lower95 <-EMrun$betaEstimate -1.96*sigma
```

A comparison to standard optimization using `optim` can be run as shown below, with convergence criterion also equal `relTol`:

```
###Repeat using optim beginning from same initial guess, maxiter set to 2000
#add hessian=TRUE if standard errors needed
beta.optim <- FFT.pat.optim(betaInit, t.pat, num.patients, PATIENTDATA,
                            patients.design, s1.seq, s2.seq, relTol, 2000)
```