```
> library(knitr)
> # set global chunk options
> opts_chunk$set(fig.path='figure/minimal-', fig.align='center', fig.show='hold')
> options(formatR.arrow=TRUE,width=90,tidy=FALSE)
```

1 SIR model and proposed branching approximation

The general stochastic epidemic model with infection rate β and recovery rate γ has infinitesimal transition probabilities as $h \to 0$

$$\Pr(S(t+h) = x_h, I(t+h) = y_h | S(t) = x_t, I(t) = y_t)$$

$$= \begin{cases} \beta x_t y_t h + o(h) & \text{if } (x_h, y_h) = (x_t - 1, y_t + 1) \\ \gamma y_t h + o(h) & \text{if } (x_h, y_h) = (x_t, y_t - 1) \\ 1 - (\beta x_t y_t + \gamma y_t)h + o(h) & \text{if } (x_h, y_h) = (x_t, y_t) \end{cases}$$

We see the interaction effect between susceptible and infected populations explicitly due to the xy product appearing in the probabilities above. Because new infections occur at rate $\beta x_t y_t$ and recoveries occur with rate γy_t, it suffices to restrict attention to the $S(t), I(t)$ populations. While branching processes fundamentally rely on particle *independence*, we can nonetheless make a very good approximation by mimicking the interaction effect over short time intervals. We propose a two-type branching process, where X_1 will represent the susceptible population and X_2 denotes the infected population. Over any time interval $[t_0, t_1)$, we use the initial population $X_2(0)$ as a constant scaling the instantaneous rates. The only nonzero rates specifying the proposed model are

$$a_1(0,1) = \beta X_2(0), \qquad a_1(1,0) = -\beta X_2(0), \qquad a_2(0,1) = -\gamma, \qquad a_2(0,0) = \gamma. \tag{1}$$

This notation corresponds to what we used for the birth-death-shift process: each type i particle produces k type 1 particles and l type 2 particles with *instantaneous rates* $a_j(k,l)$ upon completion of its lifespan.

This very simple branching process is easy to analyze, and has infection rate $\beta X_2(0) X_1(t)$ and recovery rate $\gamma X_2(t)$ for all $t \in [t_0, t_1)$, closely resembling the true model rates, with the exception of fixing $X_2(0)$ in place of $X_2(t)$ in the rate of infection. This constant initial population fixes a piecewise homogeneous per-particle birth rate to mimic interactions, but notice that *both* populations can change over the interval, and thus by linearity the overall rates change as well. While still an approximation, this offers much more flexibility than models such as TSIR that assume constant populations and rates between discrete observations.

When such a model is appropriate, its simplicity and flexibility provide attractive mathematical properties toward many statistical methods. In particular, repeated derivatives of the probability generating function corresponding to this process surprisingly have closed form solutions that can be evaluated quickly and accurately. This enables us to directly compute transition probabilities given any fixed endpoints, so that probability generating function evaluations and spectral techniques are not necessary.

The transition probabilities of the two-type branching approximation to the SIR model defined by (1) over any time interval of length t are given by

$$\Pr\{\mathbf{X}(t+s) = (k,l) | \mathbf{X}(s) = (m,n)\} := P_{mn,kl}(t) = \sum_{i=0}^{l} A(l-i)B(i), \quad \textit{where}$$

$$\begin{cases} B(i) = 0, & \forall i \leq n; \quad \textit{otherwise,} \\ B(i) = \frac{n!}{(n-i)!}(1-e^{-\gamma t})^{n-i} e^{-i\gamma t}. \\ A(l-i) = 0, & \forall (l-i) \leq (m-k); \quad \textit{otherwise,} \\ A(l-i) = \frac{m!}{(m-k-l-i)!} e^{-k\beta n t} \left[1 - \frac{\beta n}{\beta n - \gamma} e^{-\gamma t} - \left(1 - \frac{\beta n}{\beta n - \gamma}\right) e^{-\beta n t}\right]^{m-k-l-i} \left[\frac{\beta n}{\beta n - \gamma}(e^{-\gamma t} - e^{-\beta n t})\right]^{l-i}. \end{cases}$$

The formula as a sum over products of these expressions above may look unwieldy, but can be computed very quickly in practice with a vectorized implementation, and with high degrees of numerical stability even when very large factorials are necessary by using Loader's algorithm (this is used for `pbinom` in `R`).

2 R code

Below, I'll walk through some example code that implements the above formula. I've included R code that solves the ODEs using the same method as the birth-death-shift transposon paper, since we've seen that method works, for purposes of comparing accuracy — in practice, the point of closed form solutions is that we *won't* have to evaluate a whole grid of solutions.

First, we have the relatively simple ODE solutions, and code to evaluate them over a grid of arguments (naively with nested for loops) and functions to take the Fourier transforms of these grids, yielding transition probabilities.

```
> library(matrixStats)
> # solutions to phi1 and phi2, the generating function ODES
> # phi1 is generating function starting with 1 suscep, phi2 is starting with 1 infecte
> phi1 <- function(t, s1, s2, beta, gam, I0){
+   return( 1 + exp(-gam*t)*beta*I0*(s2-1)/(beta*I0-gam) +
+             exp(-beta*I0*t)*(s1 - 1 - beta*I0*(s2-1)/(beta*I0-gam) ) )
+ }
> phi2 <- function(t, s2, gam){
+   return(1 + (s2-1)*exp(-gam*t))
+ }
> #solve over a grid: s1.seq and s2.seq will be vectors of complex numbers along unit c
> phi1.grid <- function(t, s1.seq, s2.seq, beta, gam, I0){
+   grid <- matrix(nrow = length(s1.seq), ncol = length(s2.seq)) #this will store the g
+   for(i in 1:length(s1.seq)){
+     for(j in 1:length(s2.seq)){
+       grid[i,j] <- phi1(t, s1.seq[i], s2.seq[j], beta, gam, I0)
+     }
+   }
```

```
+   return(grid)
+ }
> phi2.grid <- function(t, s2.seq, gam){
+   grid <- matrix(nrow = length(s2.seq), ncol = length(s2.seq)) #this will store the g
+   for(j in 1:length(s2.seq)){
+     grid[,j] <- phi2(t, s2.seq[j], gam) #columns will be constant
+   }
+   return(grid)
+ }
> #exponentiate grid solutions appropriately and fast fourier transform to get trans pr
> getTransProbsODE <- function(t, gridLength, beta, gam, S0, I0){
+   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)
+
+   jointGrid <- phi1.grid(t, s1.seq, s2.seq, beta, gam, I0)^S0 *
+     phi2.grid(t, s2.seq, gam)^I0
+   fourierGrid <- fft(jointGrid)/length(jointGrid)
+   return(Re(fourierGrid))
+ }
```

We can use these as a "ground truth" for transition probabilities of the *approximate branching model* to test our closed form probabilities, implemented below. For the paper, we will probably only include the more relevant comparison to Monte Carlo estimates from simulating the true SIR process.

```
> # expression for the phi1 cross-partials
> A <- function(t, m, n, k, j, beta, gam){
+   if(j > m-k){return(0)} else{
+     return( factorial(m) * exp(-k*beta*n*t) *
+               ( 1 - beta*n*exp(-gam*t)/(beta*n-gam) - exp(-beta*n*t) *
+                   (1 - beta*n/(beta*n-gam)  ) )^(m-k-j) *
+               ( beta*n*(exp(-gam*t) - exp(-beta*n*t)) /
+                   (beta*n - gam) )^(j) / factorial(m-k-j) )
+   }
+ }
> # expression for the phi2 partials
> B <- function(t, n, j, gam){
+   if(j>n){return(0)} else{
+     return( factorial(n)* (1-exp(-gam*t))^(n-j) * exp(-gam*j*t)/factorial(n-j) )
+   }
+ }
> # remove if statements from A(), B(), so that vector arguments work
> # input j must be greater than m-k
> A.vectorized <- function(t, m, n, k, j, beta, gam){
+   return( factorial(m) * exp(-k*beta*n*t) *
```

```
+             ( 1 - beta*n*exp(-gam*t)/(beta*n-gam) - exp(-beta*n*t) *
+                 (1 - beta*n/(beta*n-gam)  ) )^(m-k-j) *
+             ( beta*n*(exp(-gam*t) - exp(-beta*n*t)) /
+                 (beta*n - gam) )^(j) / factorial(m-k-j) )
+ }
> #log version of previous, to handle larger numbers
> A.vec.log <- function(t, m, n, k, j, beta, gam){
+   return( lgamma(m+1) - lgamma(m-k-j+1) - (k*beta*n*t) +
+             (m-k-j)* log( ( 1 - beta*n*exp(-gam*t)/(beta*n-gam) -
+                               exp(-beta*n*t)*(1 - beta*n/(beta*n-gam)  ) ) )
+           + j * log( ( beta*n*(exp(-gam*t) - exp(-beta*n*t))/(beta*n - gam) ) )
+   )
+ }
> B.vectorized <- function(t, n, j, gam){
+   return( factorial(n)* (1-exp(-gam*t))^(n-j) * exp(-gam*j*t)/factorial(n-j) )
+ }
> B.vec.log <- function(t, n, j, gam){
+   return( lgamma(n+1) - lgamma(n-j+1) + (n-j)*log(1 - exp(-gam*t)) - gam*j*t  )
+ }
> # loops over appropriate indices in a Leibniz-rule sum for partial derivative solutio
> # uses expressions A(), B(), and returns P_{(m,n),(k,l)}(t) given beta and gamma
> # slow version with for loop: included here since it's more transparent to read
> TransProb_mnkl_old <- function(t, m, n, k, l, beta, gam){
+   AA <- BB <- rep(0,l+1)
+   c <- choose(l,seq(0,l))
+   for(i in 1:(l+1)){
+     AA[i] <- A(t, m, n, k, l-i+1, beta, gam)
+     BB[i] <- B(t, n, i-1, gam)
+   }
+ #  print(AA)
+ #  print(BB)
+   return( sum(AA*BB* c) / (factorial(k)*factorial(l) ) )
+ }
> # note: of course k can never be greater than m
> # uses log versions to handle large populations
> # put k and l first for use with outer()
> TransProb_mnkl <- function(k,l, t, m, n, beta, gam){
+   if(k>m){return(0)}
+   logAA <- logBB <- rep(0,l+1)
+   aj <- rev(seq(0,min(m-k,l)))
+   bj <- seq(0, min(n,l) )
+   c <- lgamma(l+1) - lgamma(seq(0,l) + 1) - lgamma( l+1 - seq(0,l))
+   logAA[( l+1-min(m-k,l) ):(l+1)] <- A.vec.log(t, m, n, k, aj, beta, gam)
+   logBB[ 1: (min(n,l)+1) ] <- B.vec.log(t, n, bj, gam)
+   #  print(AA)
```

```
+   #  print(BB)
+   #  print(c)
+   term <- c + logAA + logBB
+   return( exp( logSumExp(term) - lgamma(k+1) - lgamma(l+1) )  )
+ }
> #vectorize the previous for use with outer
> vectorized <- Vectorize(TransProb_mnkl, vectorize.args = c('k','l'))
> # S_0, I_0 are m,n: fixes those and returns a grid of transitions
> # over different end states for comparison with getTransProbs()
> #vectorized version
> getTransProbsClosed <- function(t, gridLength, beta, gam, S0, I0){
+   return( outer(0:gridLength, 0:gridLength, vectorized,
+                 t = t, beta = beta, gam = gam, m = S0, n = I0))
+ }
> # simple simulation of true SIR model over a time interval
> simulateSIR <- function(t.end, S, I, beta, gam, maxEvents = 99999999){
+   t.cur <- 0
+   for(i in 1:maxEvents){
+     if( S<0 || I <0 ){
+       print("Negative population? Error")
+       return(-99) #error code
+     }
+     if( S == 0 || I ==0){
+       #print("S or I is zero, end epidemic")
+       return( c(S,I) )   #end epidemic
+     }
+     infectRate <- S*I*beta
+     recovRate <- I*gam
+     rates <- c(infectRate, recovRate)
+
+     t.next <- rexp(1, sum(rates)) #time until next event
+     t.cur <- t.cur+t.next
+     if(t.cur > t.end){            #end of simulation period
+       return( c(S,I) )
+     }
+     decision <- rbinom(1, 1, infectRate/sum(rates))  #sample the type of next event
+     #print(decision)
+     if(decision == 1){   #infection
+       S <- S-1; I <- I+1
+     } else {             #recovery
+       I <- I-1
+     }
+   }
+   return(-99) #error code for testing
+ }
```

```
> #run simulation once with error catch
> sim.once <- function(t.end, S, I, beta, gam, maxEvents = 99999999){
+   res = -99 # error catch
+   while(res[1] == -99){
+     res <- simulateSIR(t.end, S, I, beta, gam, maxEvents)  }
+   return(res)
+ }
> getTrans.MC <- function(N, t.end, S, I, beta, gam){
+   result <- replicate(N, sim.once(t.end, S, I, beta, gam))
+   trans.count <- matrix(0, S+I,S+I)  #make big enough to account for all events: cou
+   for(i in 1:N){
+     id <- result[,i]+1
+     #indices in the resulting transition count: ie if you end at (1,1),
+     # you add a count to the (2,2) entry of the count matrix, etc
+     trans.count[id[1], id[2]] = trans.count[id[1], id[2]] + 1
+   }
+   tpm <- trans.count/sum(trans.count)
+   return(tpm)
+ }
```

Given the functions above for different ways to compute transition probabilities as well as Monte Carlo simulation, we are ready to run the code and look at comparisons. Below, we set $S_0 = 140, I_0 = 10$ and look at a time interval of length .5, but these are parameters we can explore when deciding what kind of results/comparisons to report.

```
> #Choose initial S and I population here
> S <- 140
> I <- 10
> beta = .5/(S+I)
> gamma = .1
> N <- 1000 #number of MC realizations: increase N in practice
> t.end <- .5 #time interval length
> #monte carlo estimate and standard error
> tpm.MC <- getTrans.MC(N, t.end, S, I, beta, gamma)
> sd.MC <- sqrt( (tpm.MC)*(1 - tpm.MC)/N )

> # now, calculate probabilities using generating functions
> gridLength = 256
> system.time( tpm1 <- getTransProbsODE(t.end, gridLength,
+                                       beta, gamma, S, I)[1:(S+I),1:(S+I)] )

   user  system elapsed
  0.366   0.007   0.424

> system.time( tpm2 <- getTransProbsClosed(t.end, gridLength,
+                                          beta, gamma, S, I)[1:(S+I),1:(S+I)]  )
```

```
   user  system elapsed
  3.131   0.051   3.206

>
```

According to timing results, it looks like just using the Fourier transforms on a grid of ODE solutions is faster, but recall in practice we only ever need to evaluate one closed form entry at a time rather than ever used the function `getTransProbClosed`. We will only need to use the function `TransProb_mnkl`.

Next, we can take a look at accuracy:

```
> # total errors:
> sum(abs(tpm1-tpm2)) # compare the two methods

[1] 2.731309e-08

> sum(abs(tpm1-tpm.MC)) #compare to true model Monte Carlo probabilities

[1] 0.1966959
```

In terms of *total* absolute error, we see that the closed form probabilities check out with the ODE method which we know should work properly. In terms of assessing how well we capture the true SIR model we are approximating, there is clearly error accrued: let's marginalize over susceptibles so that it's easier to compare than looking at every individual transition probability. We can also explore other ways to quantify or visualize the errors, and of course will eventually also compare with the continued fraction method.

Below, the purple crosses are probabilities computed using our method, and the blue points with confidence intervals are from Monte Carlo. We can try different settings to see when these are close and when they break down. When incorporated within a M-H sampler, it will be interesting to see how any discrepancies actually affect the final posterior estimates over parameters.

```
> #marginalize over susceptibles: c
> infectiveProbs.MC <- colSums(tpm.MC)
> infectiveSD.MC <- sqrt( (infectiveProbs.MC)*(1 - infectiveProbs.MC)/(N) )
> #check that this SD is correct...
> lower <- infectiveProbs.MC - infectiveSD.MC*1.96
> upper <- infectiveProbs.MC + infectiveSD.MC*1.96
> infectiveProbs.FFT <-round(colSums(tpm2),4)
> require(plotrix)
> plot(seq(S+I), xlim = c(0,50), infectiveProbs.FFT, pch = 3,
+      col = 'purple', main = "Probabilities of ending with x infectives")
> plotCI(seq(S+I), xlim = c(0,50), infectiveProbs.MC, pch = 16,
+        col = 4, ui = upper, li = lower, add=TRUE)
> #marginalize other way:
> suscepProbs.MC <- rowSums(tpm.MC)
```

```
> suscepSD.MC <- sqrt( (suscepProbs.MC)*(1 - suscepProbs.MC)/(N) )
> lower <- suscepProbs.MC - suscepSD.MC*1.96
> upper <- suscepProbs.MC + suscepSD.MC*1.96
> suscepProbs.FFT <-round(rowSums(tpm2),4)
> plot(seq(S+I), xlim = c(110,160), suscepProbs.FFT, pch = 3,
+      col = 'purple', main = "Probabilities of ending with x susceptibles")
> plotCI(seq(S+I), xlim = c(110,160), suscepProbs.MC, pch = 16,
+        col = 4, ui = upper, li = lower, add=TRUE)
```

Finally, let's begin comparisons with the continued fraction method. We begin by importing MultiBD and defining parameters of the SIR model. We choose rates similar to the Eyam example for now, and begin with a small population of 100 susceptibles and 15 infectives. There seems to be some computational bug when testing for larger cases... will look into this soon. We compute a matrix containing transition probabilities using all three methods below:

```
> library(MultiBD)
> tList  <- c(.1, .2, .25, .3 ,.35, .4, .5, .6, .7, .8, .9, 1)
> gridLength = 128
> a0 = 110 # S_0
> b0 = 15 # I_0
> A = 0
> B = gridLength - 1
> alpha = 3.2 #3.2 #this is death rate
> beta = .025 #.019 #this is transition or infection rates
> nSim = 4000 #number of MC simulations
> brates1=function(a,b){0}
> drates1=function(a,b){0}
> brates2=function(a,b){0}
> drates2=function(a,b){alpha*b}
> trans=function(a,b){beta*a*b}
```

Having specified the parameters, list of times, and rates, let's compute probabilities using each method (the following takes a while to run because of the Monte Carlo simulations):

```
> #indexed by time, type of computation, and dimensions of the tpm
> tpmArray <- array(NA, dim= c(length(tList),3, 52, 25 )) #store a subset of transitio
> for(i in 1:length(tList)){
+   t.end <- tList[i]
+   system.time( tpm.Closed <- getTransProbsClosed(t.end, gridLength,
+                                                  beta, alpha, a0, b0) )
+   tpm1 = tpm.Closed[1:(a0+1),] #using 2-type branching approximation
+
+   #using continued fractions via MultiBD
+   system.time( tpm2 <- dbd_prob(t.end, a0, b0, drates1, brates2, drates2, trans,
+                                                  a=A, B))#, computeMode=2))
```

```
+   #MC simulation "ground truth"
+   tpm.MC <- getTrans.MC(nSim, t.end, a0, b0, beta, alpha)
+   tpm3 <- tpm.MC[1:(a0+1), ]
+
+   #store subset of matrices containing about 99 percent of the mass:
+   tpmArray[i,1,,] <- tpm1[60:(a0+1),1:25]
+   tpmArray[i,2,,] <- tpm2[60:(a0+1),1:25]
+   tpmArray[i,3,,] <- tpm3[60:(a0+1),1:25]
+ }
```

Next, we can look at measures of similarity: first, we can look at where the largest entries are located, and then plot the probability mass over regions where most of the support lies.

```
> #for example, look at the ones with t.end = .5
> small1 <- tpmArray[5,1,,]
> small2 <- tpmArray[5,2,,]
> small3 <- tpmArray[5,3,,]
> #they comprise most of transition probability mass:
> sum(small1); sum(small2); sum(small3)

[1] 0.9993855

[1] 0.9930228

[1] 0.99325

> # mean errors
> mean(abs(small1- small3 ) ) #2-type vs MC

[1] 0.0004120551

> mean(abs(small2 - small3) ) #Continued Frac vs MC

[1] 0.0001502793

> # scaled heatmap images to compare tpm visually
> par(mfrow=(c(3,1)))
> image(small1, main = "Two-type branching approximation")
> image(small2, main = "Continued Fraction expansion")
> image(small3, main = "Monte Carlo estimates")
```

From the heatmap plots above, we see that indeed the continued fraction method looks like it assigns probability mass to the correct areas, very similar to the contours of the MC estimates in the bottom panel. The two-type approximation is close but centered slightly off, as is reflected by the slightly higher mean error and indices of largest values above the plot as well.

However, for larger time intervals, the two-type approximation is more noticeably far from the true MC estimates: below shows the heatmaps for $t = 1$:

```
> par(mfrow=(c(3,1)))
> image(tpmArray[12,1,,], main = "Two-type branching approximation")
> image(tpmArray[12,2,,], main = "Continued Fraction expansion")
> image(tpmArray[12,3,,], main = "Monte Carlo estimates")
```

Next, we would like to vary the populations, time intervals, etc, and make some plots of a few given transition probabilities as these parameters vary. We can compare the 2-type branching approximation with the continued fraction method and see how closely each aligns with MC simulation.

```
> library(plotrix)
> inds <- t(which(tpmArray[7,2,,] >= sort(tpmArray[7,2,,], decreasing=T)[16], arr.ind=T
> #ind1 <- sample(52,25, replace=T); ind2 <- sample(25,25,replace=T)
> par(mfrow = c(4,4), oma = c(5,4,0,0) + 0.1, mar = c(0,0,1,1) + 0.1)
> for(i in 1:16){
+ plot(tList, tpmArray[,2,inds[1,i], inds[2,i] ], pch = 17, col = 'red',
+ ylim = c(0,max(tpmArray[,,inds[1,i], inds[2,i]])),
+ yaxt = 'n', xlab = "dt")
+ MCp <- tpmArray[,3,inds[1,i], inds[2,i] ] #MC prob
+ plotCI(tList, MCp, pch = 4, col = 'green', ui=MCp+1.96*sqrt(MCp*(1-MCp)/nSim),
+ li=MCp-1.96*sqrt(MCp*(1-MCp)/nSim), add = TRUE)
+ points(tList, tpmArray[,1,inds[1,i], inds[2,i] ], col='purple', pch = 16)
+ }
```

Let's repeat some of the heatmap images to check the multibd package with larger populations:

```
> tList  <- c( .5, 1)
> gridLength = 256
> a0 = 235 # S_0
> b0 = 15 # I_0
> A = 0
> B = gridLength - 1
> alpha = 3.2 #3.2 #this is death rate
> beta = .025 #.019 #this is transition or infection rates
> nSim <- 10000
> tpmArray <- array(NA, dim= c(length(tList),2, (a0+1), 240 )) #store a subset of trans
> for(i in 1:length(tList)){
+   t.end <- tList[i]
+   system.time( tpm2 <- dbd_prob(t.end, a0, b0, drates1, brates2, drates2, trans,
+                                              a=A, B))#, computeMode=2))
+   #MC simulation "ground truth"
+   tpm.MC <- getTrans.MC(nSim, t.end, a0, b0, beta, alpha)
+   tpm3 <- tpm.MC[1:(a0+1), ]
+
+   #store subset of matrices containing about 99 percent of the mass:
```

```
+   tpmArray[i,1,,] <- tpm2[1:(a0+1),1:240]
+   tpmArray[i,2,,] <- tpm3[1:(a0+1),1:240]
+ }
```

The below plots are for $t = .5, 1$ respectively:

```
> par(mfrow=(c(2,1)))
> image(tpmArray[1,1,,1:60], main = "Continued Fraction approximation, t=.5")
> image(tpmArray[1,2,,1:60], main = "Monte Carlo estimates")
> par(mfrow=(c(2,1)))
> image(tpmArray[2,1,,1:60], main = "Continued Fraction approximation, t=1")
> image(tpmArray[2,2,,1:60], main = "Monte Carlo estimates")
```