Modern Computational Statistics: Alternatives to MCMC

Practical 1: Importance Sampling

Within population genetics, a standard model for the evolution of the frequency, x, of an allele at a locus has the following stationary distribution

$$f(x) \propto x^{\alpha_1 - 1}(1-x)^{\alpha_2 - 1} \exp\{-\sigma_1 x - \sigma_2 x^2\}, \text{ for } 0 \leq x \leq 1.$$

Where $\alpha_1 > 0$ and $\alpha_2 > 0$. We will write $\alpha = (\alpha_1, \alpha_2)$, and $\sigma = (\sigma_1, \sigma_2)$. It is often of interest to generate a sample from this distribution, calculate expecations with respect to it, or calculate the normalising constant:

$$C(\alpha, \sigma) = \int_0^1 f(x) \mathrm{d}x.$$

We will see how importance sampling can be used to do these.

Firstly download the files `IS_fn.R` and `IS_example.R`. The former contains a function to evaluate $f(x)$ or $\log f(x)$; the latter gives an example of using Importance Sampling. We will describe the code in `IS_example.R`.

```
source("IS_fn.R")
```

This just loads in the function `dWF` which evaluates the density $f(x)$ (or the log-density). You may want to look at the file `IS_fn.R` to see what the arguments of the function are, and how it works. (Note: you need to be running `R` in the same directory to where you have stored `IS_fn.R`)

```
alpha=c(3,0.5)
sigma=c(5,1)

x=seq(0.001,0.999,by=0.001)
par(mfrow=c(2,2))
logf=dWF(x,alpha,sigma,log=T)
plot(x,logf,type="l",lwd=2,xlab="x",ylab="log(f)")

plot(x,exp(logf-max(logf)),type="l",lwd=2,xlab="x",ylab="f")
```

This just sets some values for the parameter vectors α and σ (you may want to try changing these later, to see what happens). Then it plots what the density (and log-density) looks like. Looking at such plots is helpful for when you want to construct a proposal density: as you want the proposal density to mimic the target density.

Here, as $0 \leq x \leq 1$, a default choice for a proposal will be a uniform density.

```
N=10000 ##number of samples
x=runif(N) ##generate from proposal
##calculate weights
w=dWF(x,alpha,sigma)/dunif(x)
```

This performs Importance Sampling, with a sample of size N. This involves just simulating from the proposal, and then calculating (un-normalised) weights as the ratio of the target to the proposal.

```
logChat=log(sum(w)/N)##estimate of log of normalising constant
w=w/sum(w) ##normalised weights
```

We can then use the un-normalised weights to estimate $C(\alpha, \sigma)$ (or in this case $\log C(\alpha, \sigma)$). We can also normalise the weights, to then give us a weighted-sample from $f(x)$.

In some cases, we can get numerically instability in calculating weights. Often it is better to work on the log-scale. Example of how you do this is given by:

```
logw=dWF(x,alpha,sigma,log=T)-dunif(x,log=T)
maxw=max(logw)
w=exp(logw-maxw)
sumw=sum(w)
w=w/sumw ##normalised weights
logChat=maxw+log(sumw/N) ##estimate of log of normalising constant
```

To see the accuracy of the Importance Sampling, we can calculate the Effective Sample Size. Also plotting the weights can be useful:

```
ESS=sum(w)^2/sum(w^2) ##Effective Sample Size
hist(w)
```

We can use the weighted sample to obtain estimate of expectations or probability of events:

```
sum(w*x) ##estimate of mean frequency
sum(w*(x>0.99)) ##estimate of probability frequency > 0.99
```

In some cases people re-sample from the weighted sample, which gives an IID sample from the approximation to the target

```
M=1000 ##sample size
index=sample(1:N,prob=w,size=M,rep=T) ##index of particles to keep
xsam=x[index]
hist(xsam,breaks=seq(0,1,by=0.01))
```

Plotting a histogram, and comparing with the plot of $f(x)$ shows how reasonable an approximation we have. [Note using a density plot does not work as well here, as standard density plots can be inaccurate at the boundary of x.]

Exercises

(1) It is possible to analytically calculate $f(x)$ in two cases. One is where $\sigma = (0, 0)$; the other where $\alpha = (1, 1)$ and $\sigma_2 = 0$. In both these cases, try and evaluate the accuracy of Importance Sampling by comparing the estimates with the true values (for $C(\alpha, \sigma)$, or for the mean of the distribution, or for the probability calculation).

(2) **Group Work**. In groups try and construct a good proposal distribution for this problem. It needs to work well for a range of α and σ values. In particular when α_1 and/or α_2 are close to 0; and when some of σ_1 and/or σ_2 are large.

You should construct two functions for this: one that simulates from the proposal, and one that calculates the density of the proposal. You will (probably) want the proposal (and these functions) to depend on α and σ. At the end of the session, each group should email their two functions to `p.fearnhead@lancs.ac.uk`: and tomorrow we will see which group did best.

Practical 2: Particle Filters

The aim of this practical is to see how to implement a simple particle filter; to get some idea how features of the model affect the Monte Carlo error in the filter; and to see how different approaches to parameter estimation within a partical filter work.

To do this we will work with a simple model of stochastic volatility:

$$X_t|X_{t-1} = x_{t-1} \sim \mathrm{N}(\phi x_{t-1}, \sigma^2),$$

$$Y_t|X_t = x_t \sim \mathrm{N}\left(0, \exp\{2(\gamma + x_t)\}\right).$$

Normally we would expect $\phi \approx 1$. The variance of the stationary distribution of the X_t process is $\sigma^2/(1-\rho^2)$.

Code for analysing this model is in `SIR.R` and `SIR_example.R`. The former has functions for implementing the SIR filter, and simulating data; the latter uses these functions to simulate and analyse data.

Basic Particle Filter

First consider `SIR_example.R`.

```
source("SIR.R")

##First simulate data:
n=100 ##number of data points

##parameters

phi=0.8
sigma=sqrt(1-phi^2) ##this choice fixes the marginal variance of X_t at stationarity to 1
gamma=1

par=c(phi,sigma,gamma) ##parameter vector

data=SVsim(n,par)
y=data$y ##observations
truth=data$x ##true state
```

This first loads in the functions in `SIR.R`, sets the parameters for the model, and then simulates data. Note that we have defined the parameters so that as we change `phi`, the variance of the model at stationarity is unchanged. The vector `par`, defined as a parameter vector with entries as above, is required as an input in the function `SVsim` that simulates data. The output of `SVsim` is a list for the observations and the states: and we have stored these in vectors `y` and `truth` respectively.

```
prior=c(0,sigma/sqrt(1-phi^2)) ##prior for SIR filter
N=100 ##number of particles
PLOT=T
par(ask=T) ## this will produce a prompt after each plot
#par(ask=F)## use to remove the prompt after each plot
out=SIRfilterSV(N,y,par,prior,truth,PLOT=T)
```

`SIRfilterSV` runs a particle filter. If you look at the file `SIR.R` there will be some information on the inputs for this function. As part of input, this needs the prior parameters (mean and standard

deviation) for X_1, which are stored in `prior`; the parameters of the model, which are stored in `par`; and the number of particles to use, `N`. The function has the option of plotting the approximation to the filtering density at each iteration: and this is governed by the flag `PLOT`. These plots of the approximation to the filtering density can also show the true value of X_t at each time, with this being input as `truth`.

In order to understand what `SIRfilterSV` does, I find it helpful to run the function bit and bit (and then see what each bit of the function does). Thus rather than running the function as above, I would copy and paste the `R` commands. In order for this to work we need to have defined everything the function `SIRfilterSV` expects. Looking at the definition:

```
SIRfilterSV=function(N,y,par,prior,truth=NULL,PLOT=T)
```

we can see that we have correctly defined each of the objects in `R` that this function expects.

The function follows closely the algorithms in lectures.

```
n=length(y) ##number of observations
d=1 ##dimension

 ##(1)simulate particles
 particles=matrix(0,nrow=N,ncol=d) ## particles are x_t
 for(j in 1:d) particles[,j]=rnorm(N,prior[j],prior[j+d])
```

The first two lines are just calculating the number of observations, and then the dimension of the state vector. Then the first part of the algorithm is to simulate the particles from the prior. The function assumes that the prior is a normal distribution, with mean and standard deviation given by the vector `prior`.

```
  ##(2) calculate weights
  w=dnorm(y[1],0,exp(par[3]+particles[,1]))
```

Next we calculate the weights of the particles, which are proportional to the likelihood. Our weighted particles are given by `particles` and `w` respectively.

```
##(3) log of estimate of likelihood
 logl=log(mean(w))
```

Our estimate of the likelihood $p(y_1)$ is just the mean of the weights. It is numerically more stable to store the log of this.

```
 ##OPTIONAL -- STORAGE OF FEATURES OF FILTER -- MEAN AND VAR
  M.st=matrix(0,nrow=n,ncol=d)
  V.st=matrix(0,nrow=n,ncol=d)
  w=w/sum(w)
  for(j in 1:d){
    M.st[1,]=sum(w*particles[,j])
    V.st[1,]=sum(w*particles[,j]^2)-M.st[1,]^2
  }
```

This part of code is just calculating the estimate to the $\mathrm{E}(X_1|y_1)$ and $\mathrm{Var}(X_1|y_1)$. We will in fact calculate, store and return the approximations to the filtering means and variances at each time-point.

```
 ##PLOT
   if(PLOT){
   par(mfrow=c(1,2))
   plot(range(particles[,1]),c(0,max(w)),xlab="X_t",ylab="Weight",type="n")
   for(j in 1:N) lines(c(particles[j,1],particles[j,1]),c(0,w[j]))
   if(length(truth>=1)) abline(v=truth[1],col=2)

   plot(density(particles[,1],weights=w),xlab="X_t",ylab="Density",main="")
   if(length(truth>=1)) abline(v=truth[1],col=2)
}
```

This just plots the weighted particles. It produces two plots: one is the particles and weights, but this can be hard to interpret; the second is a density plot, which is easier to interpret.

Next we have a loop. If we are copying and pasting the commands, to see what is happening, then we cannot copy the loop in. Instead type `i=2`, and continue with

```
 ##(1) Resample
      index=sample(1:N,prob=w,size=N,rep=T)
      particles=particles[index,]
      if(d==1) particles=matrix(particles,ncol=1)
```

This resamples the particles. *As mentioned in lectures: there are better resampling approaches than multi-nomial resampling*, we are using it just for simplicity.

The last line is there just because, in the previous line, `R` automatically makes `particles` a vector: and later on we assume it is a matrix.

```
 ##(2) Propagate particles
    #x_t=phi x_{t-1}+N(0,sigma^2); phi=par[1]; sigma=par[2]
    particles[,1]=particles[,1]*par[1]+rnorm(N,0,par[2])
```

We next propagate the particles, according to the state dynamics: $X_t = \phi X_{t-1} + \epsilon_t$, where $\epsilon_t \sim \mathrm{N}(0, \sigma^2)$.

```
  ##(3) Weight
      w=dnorm(y[i],0,exp(par[3]+particles[,1]))

  ##(4) update log of estimate of likelihood
   logl=logl+log(mean(w))
```

Here we calculate the new weights. And then our new estimate of the log-likelihood. Here we have used the fact that the mean of the weights is an estimate of $p(y_i|y_{i|i-1})$, and $\log p(y_{1:i}) = \log p(y_{1:i-1}) + \log p(y_i|y_{1:i-1})$. The remaining code than calculates the approximation to the filtering mean and variance; and then does the plots.

If we return to `SIR_example.R`, then there is further code for plotting the filtered estimates of the state against the truth. It also contains one simple way of looking at the Monte Carlo accuracy of the SIR filter: by running the filter multiple times on the same data set.

Questions

1. Investigate how the choice of ϕ and n affect the accuracy of estimates of the state, and the Monte Carlo error of the filter. For the latter, use $N = 100$ particles.

2. How would you adapt the code, if the state mode was

$$X_t = \phi X_{t-1} + \epsilon_t,$$

where ϵ_t has a t_{10} distribution, with variance σ^2?

Estimating Parameters

We will now use the same model to look at estimating parameters using the SIR filter. We will consider inference for ϕ. It makes sense to assume that γ and $\tau^2 = \sigma^2/(1-\rho^2)$ are known. That is, we know the stationary variance of the X_t process, rather than the variance of the noise in the dynamics of the X_t process. It also makes sense to re-parameterise the model in terms of

$$\text{logit}(\phi) = \log\left(\frac{\phi}{1-\phi}\right),$$

which can take any value on the real line. This avoids problems that Kernel Density Estimation methods have with random variables that have a bounded support (edge-effects).

Within `SIR.R` is a function `SIRfilterSVpar` that runs an SIR filter for this model. The inputs are described within the `SIR.R` file, though this time `prior` is the prior mean and standard deviation of X_1 and logit(ϕ); and `par` is the vector with τ and γ as entries. We also need to input h, the bandwidth in the KDE approach, and a flag `LW` that determines with the Liu/West shrinkage is applied. The output of the function includes the (approximations to) the filtering mean and variance for X_t and logit(ϕ).

The code is very similar to that above. Each particle now has two components, a value for X_t and a value for logit(ϕ). Then, the main changes are to the propagate step, which now has two parts. This involves first updating the values for logit(ϕ):

```
##(2) Propagate particles
  #First update parameters
  if(h>0 && V.st[i-1,2]>0){##JITTER
        if(LW){#Liu-West -- shrinkage
           lambda=sqrt(1-h) ##
           particles[,2]=lambda*particles[,2]+(1-lambda)*M.st[i-1,2]
        }
        #Jitter --for both LW and non LW
        particles[,2]=particles[,2]+rnorm(N,0,sqrt(h*V.st[i-1,2]))
  }
```

This is only needed if both $h > 0$ and our estimate of the posterior variance is greater than zero. (We check the latter, as due to numerical inaccuracies, we sometimes get estimates of the posterior variance that are negative – and this will then cause the code to crash.) Note that we are using the fact that we have already stored the posterior mean and variance of logit(ϕ).

After updating the values of logit(ϕ), we then propagate the state component:

```
#x_t=phi x_{t-1}+N(0,sigma^2);
## phi=exp(particles[,2])/(1+exp(particles[,2]); sigma=par[1]*sqrt(1-phi^2)
phi=exp(particles[,2])/(1+exp(particles[,2]))
sigma=par[1]*sqrt(1-phi^2)
particles[,1]=particles[,1]*phi+rnorm(N,0,sigma)
```

This is slightly more complicated than previously, because of the different parameterisation that we are using.

Questions

3. Investigate how the three different methods (corresponding to $h = 0$; and $h > 0$ with and without shrinkage) perform. How is this affected by the prior, the value of n or N, and the value of h? The code in `SIR_example.R` give examples of plots you could use to compare them qualitatively.

4. An alternative approach is to base the variance of the *noise* added to the parameters in a different way. One possibility is to choose it as $h\sigma_0^2/n$, where σ_0^2 is the prior variance. How could you adapt the code so as to implement this? If you do, how does this method perform.

Solution to 2.

You only need to change the code for propagating particles, to

```
##(2) Propagate particles
    #x_t=phi x_{t-1}+sqrt(0.8)*sigma*T_10  ; phi=par[1]; sigma=par[2]
    particles[,1]=particles[,1]*par[1]+par[2]* rt(N,df=10) *sqrt(0.8)
```

Where we have used that if $\epsilon \sim t_{10}$, then

$$\mathrm{Var}(\epsilon) = 10/8 = \frac{1}{0.8}.$$

Solution to 4. You only need to change the code for adding noise to the parameter component of the particles, to

```
#Jitter --for both LW and non LW
        particles[,2]=particles[,2]+rnorm(N,0,sqrt(h*prior[4]^2/i))
```

Practical 3: Pseudo Marginal MCMC

The aim of this practical is to get some experience using the Pseudo-Marginal MCMC algorithm. We will look at the Stochastic Volatility model of Practical 2.

$$X_t|X_{t-1} = x_{t-1} \sim \mathrm{N}(\phi x_{t-1}, \sigma^2),$$

$$Y_t|X_t = x_t \sim \mathrm{N}\left(0, \exp\{\gamma x_t\}\right).$$

As described in Practical 2, it makes sense to work with parameters ϕ, τ and γ, where $\tau^2 = \sigma^2/(1-\rho^2)$ is the variance of the stationary distribution of the X_t process. Furthermore we will work with a parameter vector $\theta = (\mathrm{logit}(\phi), \log(\tau), \gamma)$, so that each component of the parameter vector can take values on the real line. We will assume independent normal priors for each component of θ.

We will implement a Random-Walk MCMC algorithm, but with the likelihood being estimated by a Particle Filter. We will consider implementing the MCMC where we update each component of θ one at a time, using a normal proposal centered around the current value. Note that this is a symmetric proposal, so the ratio of proposals cancels in the acceptance probability.

Code in `PseudoMarginalMCMC.R` shows how to implement a Pseudo (or Particle) MCMC algorithm, when just ϕ is unknown. (We will initially investiage this case, and then look at extending it to the case where τ and γ are also unknown.)

```
source("SIR.R") ##Load in function SIRfilterSV; and SVsim
##simulate data -- remember observations are data$y
data=SVsim(100,c(0.9, sqrt(1-0.9^2), 1)) ##parameters are (phi,sigma,gamma)
```

This part of the code reads in the two functions we will use – one to simulate data, and one to run the SIR filter. Then it simulates data with $n = 100$ observations; $\phi = 0.9$, $\tau = 1$ and $\gamma = 1$. The observations are stored as `data$y`.

```
###PRIOR on logit(phi); log(tau); and gamma are normal; means and sd given in vector
##  p.m and p.sd respectively
p.m=c(3,-1,1);p.sd=c(1,1,1)
```

This defines the prior mean and standard deviation for the components of θ: initially we will only use the first component of each of these, as only the first component of θ (i.e. ϕ) is unknown.

```
### random walk sds given by prop.sd
prop.sd=c(0.5,0.1,0.1)

M=1000 #number of MCMC iterations
N=100 #number of particles

###Storage
state.st=matrix(0,nrow=M,ncol=4)
acc.n=rep(0,3) ##accept prob for each update
```

We now define the tuning parameters of the MCMC algorithm: the standard deviation of the random-walks; the number of MCMC iterations; and the number of particles used in the SIR filter. We then define a matrix to store the state of the MCMC after each iteration, and to record the number of acceptances (which can be useful to tune the MCMC algorithm).

```
##Initialise
state.cur=rep(0,4)  ##State is logit(phi),log(tau),gamma, log(Z)
state.cur[1:3]=rnorm(3,p.m,p.sd)
###Initially just assume phi is unknown
state.cur[3]=1 ##gamma=1
state.cur[2]=0 ##log-tau=0
```

This part initialises the MCMC. Our state will be a value of θ and our (log of the) estimate of the Likelihood for this value of θ. Here we initiate θ from the prior, but with θ_2 and θ_3 set to be equal to the (known) true values.

Next we get our initial estimate of the likelihood. We will store the log of this estimate (which is what our SIR filter outputs):

```
phi=exp(state.cur[1])/(1+exp(state.cur[1]))
sigma=exp(state.cur[2])*sqrt(1-phi^2)
par=c(phi,sigma,state.cur[3] ) ##parameters needed for input to PF
prior=c(0,exp(state.cur[2])) ##Prior for X_1 is N(0,tau^2)
state.cur[4]=SIRfilterSV(N,data$y,par,prior,PLOT=F)$l
```

The first four lines are just calculating the parameter values to input in the SIR filter, and the prior for X_1. Remember the function `SIRfilterSV` inputs the number of particles, the observations, a vector (ϕ, σ, γ) and a vector, $(0, \tau)$ of the prior mean and standard deviation of X_1. The output of this function is a list, whose entry labelled `l` is the log of the estimate of the likelihood.

We then have a loop `for(i in 1:M)` over the iterations of the MCMC algorithm. Each iteration will involve updating each of the components of θ. Currently we only update θ_1:

```
##Update logit(phi)
lp.prop=rnorm(1,state.cur[1],prop.sd[1])
```

This proposes a new value for $\theta_1 = \text{logit}(\phi)$.

```
##Generate estimate of log(p(data|parameters))
phi=exp(lp.prop)/(1+exp(lp.prop))
sigma=exp(state.cur[2])*sqrt(1-phi^2)
par=c(phi,sigma,state.cur[3] ) ##parameters needed for input to PF
prior=c(0,exp(state.cur[2])) ##Prior for X_1 is N(0,tau^2)
Z.prop=SIRfilterSV(N,data$y,par,prior,PLOT=F)$l
```

This then generates our new estimate of the likelihood. Again we store the log of this estimate.

```
##Acceptance probability: again proposal ratio cancels
acc.p=exp(Z.prop-state.cur[4]+ sum(dnorm(c(lp.prop,state.cur[2:3]),p.m,p.sd,log=T))
   -sum(dnorm(state.cur[1:3],p.m,p.sd,log=T)) )
```

This calculates the acceptance probability. Strictly the acceptance probability is the minimum of this and 1, but we will simulate acceptance by simulating a standard uniform, and see if it is less than `acc.p`. This will result in always accepting if `acc.p` is greater than 1.

```
if(runif(1)<acc.p){
state.cur[1]=lp.prop
```

```
state.cur[4]=Z.prop
acc.n[1]=acc.n[1]+1
}
```

Finally we see if we accept the proposed value. If so, we update the state of the MCMC algorithm.

At the end of an iteration of the MCMC algorithm we store the current state

```
state.st[i,]=state.cur
```

Questions

(1) Investigate how the Particle MCMC algorithm mixes. In particular, look at how this is affected by the choice of N, the number of observations, n, and the choice of standard deviation of the random walk proposal. In thinking about how you should choose N, it is natural to fix the total number of simulations NM, where M is the number of MCMC iterations.

What happens if N is too small?

You may want to look at trace plots, and acf-plots of the output of the MCMC algorithm.

(2) Write code to update $\theta_2 = \log(\tau)$ and $\theta_3 = \gamma$.

(3) Investigate the performance of the Particle MCMC when all three parameters are unknown.

.

Solution to Qu.2 One example of code to update θ_2 and θ_3 is

```
##update log(tau)
lt.prop=rnorm(1,state.cur[2],prop.sd[2])
phi=exp(state.cur[1])/(1+exp(state.cur[1]))
sigma=exp(lt.prop)*sqrt(1-phi^2)
par=c(phi,sigma,state.cur[3] ) ##parameters needed for input to PF
prior=c(0,exp(lt.prop)) ##Prior for X_1 is N(0,tau^2)
Z.prop=SIRfilterSV(N,data$y,par,prior,PLOT=F)$l
##Acceptance probability: again proposal ratio cancels
acc.p=exp(Z.prop-state.cur[4]+ sum(dnorm(c(state.cur[1],lt.prop,state.cur[3]),p.m,p.sd,log=T))
               -sum(dnorm(state.cur[1:3],p.m,p.sd,log=T)) )
if(runif(1)<acc.p){
state.cur[2]=lt.prop
state.cur[4]=Z.prop
acc.n[2]=acc.n[2]+1
}

##update gamma
g.prop=rnorm(1,state.cur[3],prop.sd[3])
phi=exp(state.cur[1])/(1+exp(state.cur[1]))
sigma=exp(state.cur[2])*sqrt(1-phi^2)
par=c(phi,sigma,g.prop ) ##parameters needed for input to PF
prior=c(0,exp(state.cur[2])) ##Prior for X_1 is N(0,tau^2)
    Z.prop=SIRfilterSV(N,data$y,par,prior,PLOT=F)$l
##Acceptance probability: again proposal ratio cancels
acc.p=exp(Z.prop-state.cur[4]+ sum(dnorm(c(state.cur[1:2],g.prop),p.m,p.sd,log=T))
               -sum(dnorm(state.cur[1:3],p.m,p.sd,log=T)) )
if(runif(1)<acc.p){
state.cur[3]=g.prop
state.cur[4]=Z.prop
acc.n[3]=acc.n[3]+1
}
```

Practical 4: Approximate Bayesian Computation

In this practical we will look at implementing ABC for inference for alpha-stable distributions: seeing how rejection sampling and MCMC algorithms perform; and the importance of the choice of summary statistics. Example code is in `ABC.R`.

To perform ABC we will need to be able to simulate data from an alpha-stable distribution, this can be done using the function `rstable` from package `stabledist`. So first we need to load this package.

```
library(stabledist) ##load library
```

ABC-Rejection Sampling

The alpha-stable distribution has 4 parameters: a shape parameter α, with $0 < \alpha \leq 2$; a skew parameter β, with $-1 \leq \beta \leq 1$; a scale parameter, $\gamma > 0$; and a location parameter δ. We will simulate data, assume α, β and γ are known, and look at using ABC to infer δ.

```
###simulate data to analyse
alpha=1.2 ##shape
beta=0.1 ##skew
gamma=1 ##scale
delta=2 ##location

n=100

y=rstable(n,alpha,beta,gamma,delta) ##data to analyse
```

For ABC we need a prior on the unknown parameter, δ, which we will take to be normal, mean 0, and initially a standard devitation of `p.sd`

```
p.sd=5
```

And then, within ABC we will need to choose appropriate summary statistics. A natural choice here, as δ is a location parameter, could be the sample mean

```
S=function(y){
 return(mean(y))
}

Sobs=S(y) ##observed summary statistics
```

This defines our summary statistic function `S`, and calculates the value of the summary statistic for the observed data. In ABC we will work with a quadratic distance defined as

$$(S(y_{\text{obs}}) - S(y_{\text{sim}}))^T A (S(y_{\text{obs}}) - S(y_{\text{sim}})),$$

for some matrix A. Next we define A (though as we have a single summary statistic, the choice does not currently matter; as we consider other choices, A will determine the weights given to the closeness of the different summaries).

```
d=length(Sobs) ##dimension of summary statistics
A=matrix(0,nrow=d,ncol=d); diag(A)=1 ##matrix for distance -- equal weight for all components
```

This default gives equal weight to all summaries.

First we will consider a rejection sampler. One advantage of rejection sampling, is that we can delay choosing the threshold on the distance until we have done all the simulations (and observed the distribution of this distance). So we will store each value of δ we simulated, together with the distance we obtain for the simulated summary statistics for that parameter value:

```
N=10000 ##number of samples
ABC.out=matrix(0,nrow=N,ncol=2) ##matrix to store (delta,dist) pairs

for(i in 1:N){ ##LOOP
  delta.prop=rnorm(1,0,p.sd)
  ysim=rstable(n,alpha,beta,gamma,delta.prop) ##simulated data
  Ssim=S(ysim) #summaries
  dist=matrix(Sobs-Ssim,nrow=1) %*% A %*% matrix(Sobs-Ssim,ncol=1)
  ABC.out[i,]=c(delta.prop,dist)
}
```

Each iteration just involves simulating a δ value; simulating data; calculating the summary statistic value; calculating the distance between this simulated and the observed summary statistic; and then storing δ and the distance.

To post-process, we just need to choose a threshold, and then keep only δ values whose distance is less than this threshold.

```
h=quantile(ABC.out[,2],0.02) ##threshold for ABC
ABCsam=ABC.out[ABC.out[,2]<h,1] ##sample of delta values
plot(density(ABCsam))
cat("ABC posterior mean",mean(ABCsam),"\tABC posterior variance",var(ABCsam),"\n")
```

The above chooses a threshold based on the quantiles of the distances: and will keep 2% of all simulated data values. It then plots the ABC approximation to the posterior, as well as the posterior mean and variance.

Questions

1. Investigate what affect the choice of h has on the ABC posterior.
2. Look at trying other summary statistics (natural ones are the median; or to include other quantiles, or a trimmed mean) – does this have much affect?
3. How does the efficiency of rejection sampling depend on the prior. Look at increasing the prior-variance.

ABC-MCMC

An alternative implementation of ABC is to use MCMC. Code for this is also available in `ABC.R`. We will use a random-walk proposal within the MCMC algorithm. We need to choose the standard deviation of this proposal, and the threshold on the distance.

```
N=10000 ##number of samples
ABC.out=rep(0,N)
h=0.1
prop.sd=0.1
```

Here N is the number of iterations of the MCMC algorithm; and these will stored in `ABC.out`. We have a threshold `h`, and standard deviation of our proposal `prop.sd`.

```
##initialise
delta.cur=median(y)
```

For our MCMC we need to choose a starting value. Care is needed here (as ABC-MCMC can struggle to move if it is in the tail of the posterior).

```
for(i in 1:N){
  delta.prop=rnorm(1,delta.cur,prop.sd)
  ysim=rstable(n,alpha,beta,gamma,delta.prop) ##simulated data
  Ssim=S(ysim) #summaries
  dist=matrix(Sobs-Ssim,nrow=1) %*% A %*% matrix(Sobs-Ssim,ncol=1)
  if(dist<h){
    acc.p=dnorm(delta.prop,0,p.sd)/dnorm(delta.cur,0,p.sd)
    if(runif(1)<acc.p){
      delta.cur=delta.prop
    }
  }
  ABC.out[i]=delta.cur
}
```

This is the main loop of the MCMC algorithm. We propose a value for δ, simulate data and calculate the summary statistics. We then check if the simulated summary statistic is close enough to the observed value: if it is not we reject; if it is then we accept with a probability that depends on the prior.

Questions

4 Investigate how the choice of h and the prior standard deviation affect mixing of ABC-MCMC.

5 What happens if you change the initial value of the MCMC algorithm: particularly if this is in the tail of the posterior.

Group Work

In groups, consider analysing the data in `Alphadata.R`, using rejection sampling ABC. You can assume $\gamma = 1$, but all other parameters are unknown, with priors

$$\alpha \sim \text{Unif}[0.5, 0.9] \quad \beta \sim \text{Unif}[-0.5, 0.5] \quad \delta \sim \text{N}(0, 2^2).$$

You need to decide what summary statistics to use, and how to choose A and h, appropriate for $N = 50,000$.

Email your summary statistic function `S`; and your choice of `A` and `h` to `p.fearnhead@lancs.ac.uk`.