Bayesian_HW_4_Whitson

Paul Whitson

5/6/2020

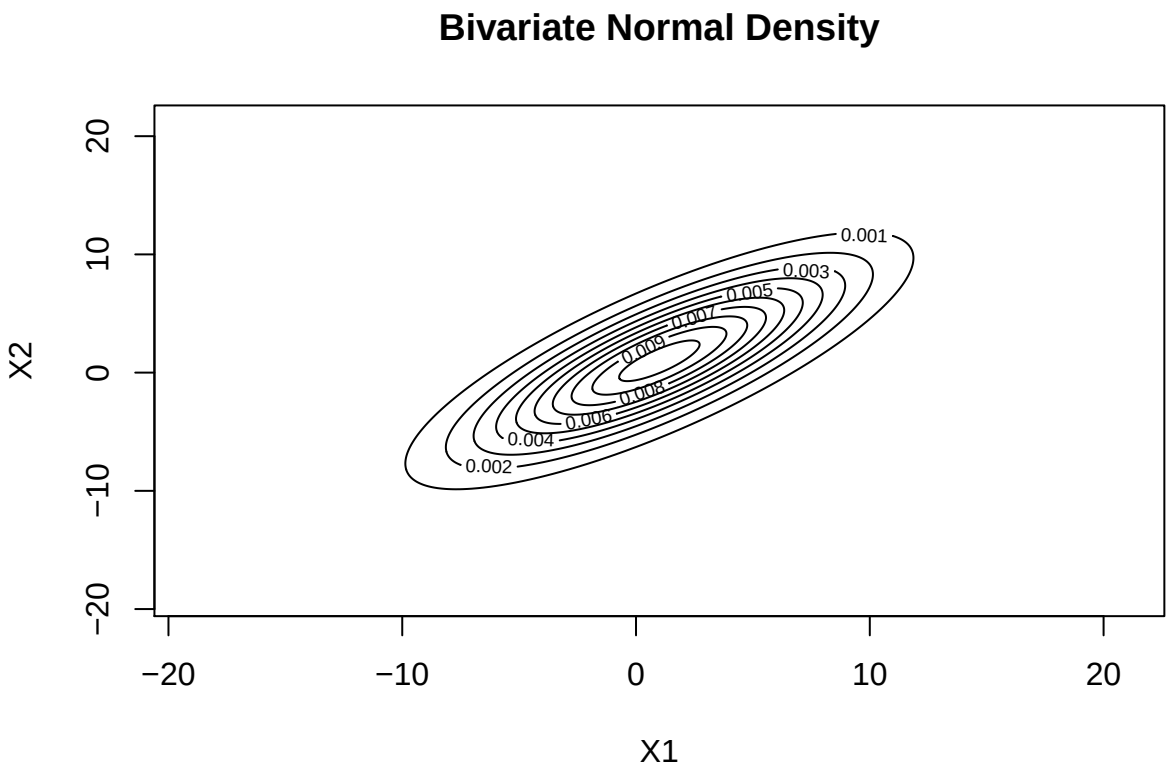
1 Bivariate Normal Distribution

Create function to calculate density of a bivariate normal distribution with given parameters:

```
#Set parameters of bivariate distribution: means, standard deviation, and correlation
myDenParams <-c(1,1,5,5,0.8)
myArgmts <- c(1,1)

#create function to calculate density at a point specified by arguments x1 and x2:
den <- function(argmts, params) {
  z <- ((argmts[1]-params[1])^2)/(params[3]^2) -
    2*params[5]*(argmts[1]-params[1])*(argmts[2]-params[2])/(params[3]*params[4]) +
    ((argmts[2]-params[2])^2)/(params[4]^2)

  dens <- (2*pi*params[3]*params[4]*sqrt(1-params[5]^2))^(-1)*exp(-z/(2*(1-params[5]^2)))
  return(dens)
}

den(myArgmts, myDenParams)
```

```
## [1] 0.01061033
```

```
#Create contour plot of density for comparison with density generated later via Metropolis algorithm:
#Create sequence of values for x1 and x2:
x1 <- seq(myDenParams[1]-4*myDenParams[3], myDenParams[1]+4*myDenParams[3], length.out = 1000)
x2 <- seq(myDenParams[2]-4*myDenParams[4], myDenParams[2]+4*myDenParams[4], length.out = 1000)

#Initialize matrix to store density information
DensityMatrix <- matrix(nrow = 1000, ncol = 1000, dimnames = list(paste0("X1=",round(x1,2)), paste0("X2=

#Calculate density at each point:
for (i in 1:length(x1)) {
  for (j in 1:length(x2))
  DensityMatrix[i,j] <- den(c(x1[i],x2[j]), myDenParams)
}

#Display contour plot:
contour(x=x1, y=x2, z=DensityMatrix, xlab="X1", ylab = "X2", main = "Bivariate Normal Density")
```



2 Metropolis Rule

Create a function to perform one step of the Metropolis algorithm:

- identify candidate for new point assuming a Gaussian distribution with mean 0 and standard deviation
- determine probability density at current (initial) point and at the new (candidate point)
- accept or reject candidate point per Metropolis criterion

```
oneStep <-function(denFun, initSt, denParams, sigProp) {
  #denFun is density function
  #initSt is initial vector of observations (in this case, initial values of x1 and x2)
  #denParams is vector of parameters of the distribution
  #sigProp is standard deviation of the distribution that generates proposals.

  #compute density at current (initial) state:
  dens0 <- denFun(initSt, denParams)

  #select candidate for new state: generate random values ~N(0,sigProp) and add to initSt
  newState <- initSt + rnorm(length(initSt), 0, sigProp)

  #compute density at new state:
  densNew <- denFun(newState, denParams)
```

```
  #determine whether new state is accepted
  acceptIndicator <- rbinom(1,1, min(1,densNew/dens0))
  #if new density is greater than old, accept newState with probability 1.
  #if it is less, accept newState with probability densNew/dens0

  #output the coordinates of the new points, plus an indicator of whether they were accepted or not
  Output <- c(newState, acceptIndicator)

  return(Output)
}

#Example:
(oneStep(den, c(2,-1), myDenParams, .05))
```

```
## [1]  2.074669 -1.014513  1.000000
```

3: Run MCMC

Create function to implement MCMC algorithm:

```
MCMC <- function(initSt, denParams, nIter, sigProp) {

  #Initialize matrices to store results:
  trajectory <- matrix(nrow = 0, ncol = 3)
  rejections <- matrix(nrow = 0, ncol = 3)

  #Store initial state as first "accepted" value:
  trajectory <- rbind(trajectory, c(initSt, 1))

  #Store initial state parameters as the first "current state"
  currentSt <- initSt

  for (i in 1:nIter) { #for each iteration:

    #generate vector of points from oneStep, based on current state and parameters supplied:
      candidate <- oneStep(den, currentSt, denParams, sigProp)

      if (candidate[3]==1) { #if value from oneStep is "accepted"
        trajectory <- rbind(trajectory, candidate) #append values to "accepted" matrix
        currentSt <- candidate[1:2]} #make accepted candidate the new "currentSt" for the next iteratio

      else {
        rejections <- rbind(rejections, candidate) #append rejected candidate to "rejected" matrix
      }
  }

  return(list(trajectory = trajectory, rejections = rejections))
}
```

Run multiple trajectories of algorithm for various values of sigProp, the standard deviation of the "proposal" distribution:

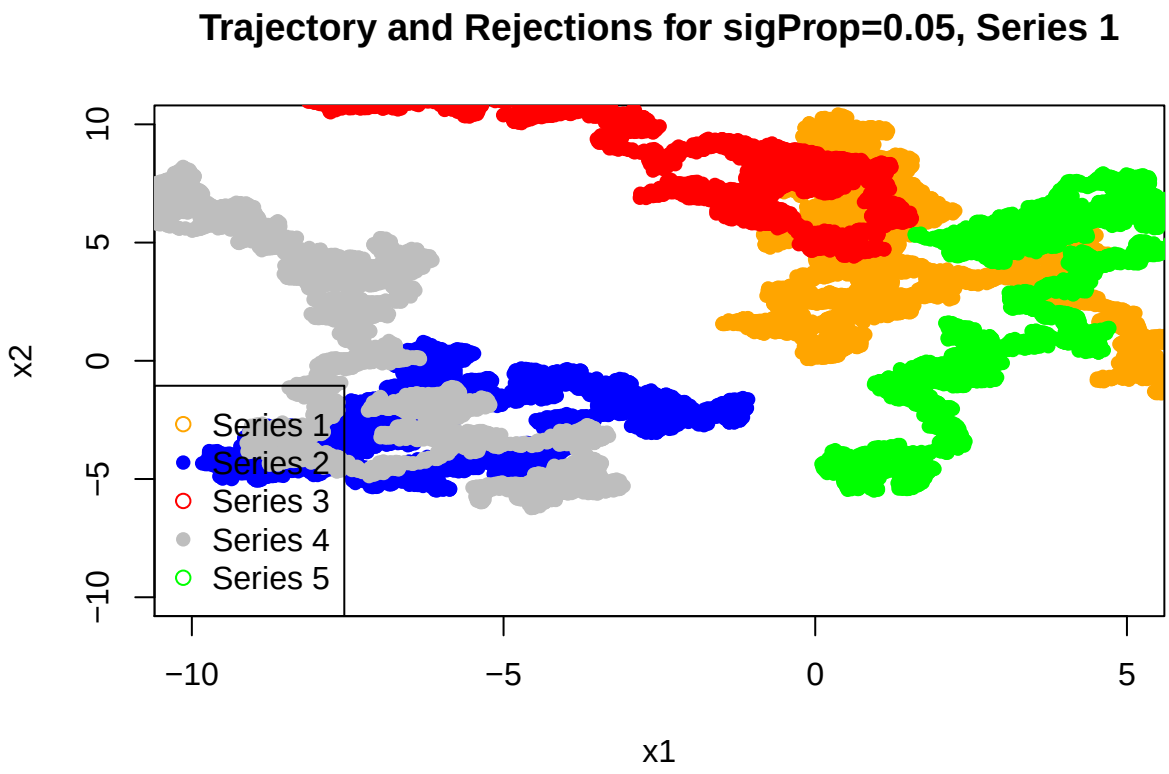
```r
#list of values of sigProp to explore:
sigList <- c(0.05, 0.1, 0.5, 1.0, 2.0)

#Initialize list to store results:
GridSearchOutput <- replicate(5, vector("list", 5), simplify = FALSE)
names(GridSearchOutput) <- paste0("sigProp=", sigList)

#Run 5 different trajectories for each value of sigProp:

for (i in 1:length(sigList)) { #for each element in sigList

  sigProp <- sigList[i] #apply function to the ith element of sigList

  for (j in 1:5) { #perfom 5 random walks for each value of sigProp
    init=c(runif(1,-15,15), runif(1,-15, 15)) #pick a random starting point
    GridSearchOutput[[i]][[j]] <- MCMC(initSt = init, myDenParams, nIter = 10000, sigProp = sigProp)
  }
}
```

Output:

The following graph shows 5 "accepted" trajectories for sigProp = 0.05. While they all begin in different places, they all eventually find their way to the region of high density near the point (0,0).

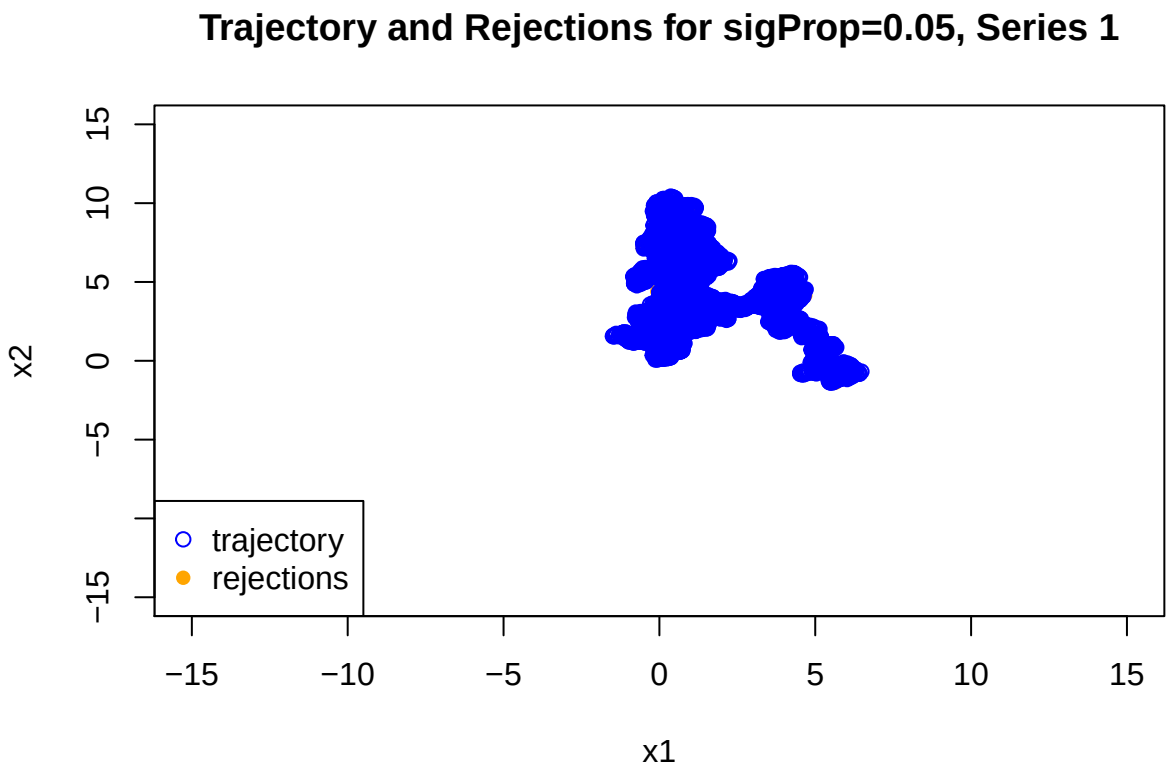
However, because the sigProp is so small, it takes some of the trajectories a long time to find this region, and they do not explore the parameter space very thoroughly. See the following section for results with other values of sigProp.

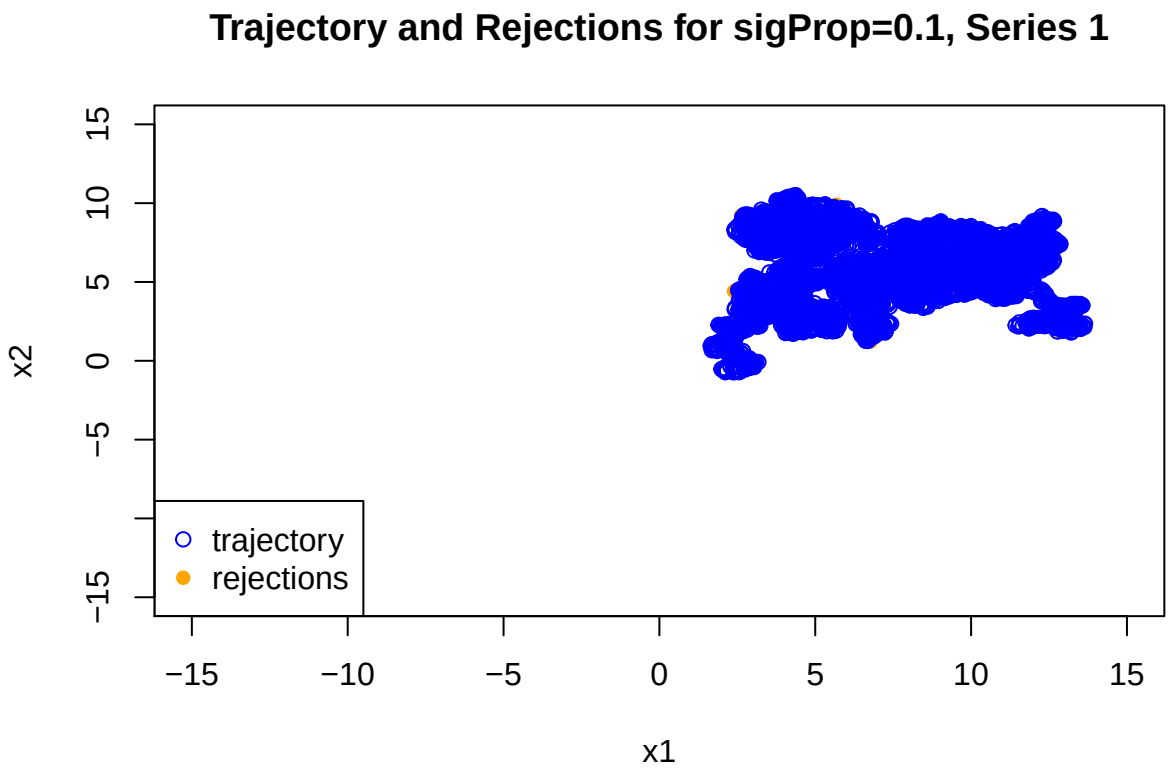
```r
    plot(GridSearchOutput[[1]][[1]]$trajectory[,1:2],
         col="orange", pch=16, lwd = 0.5, #format for rejection points
         ylim=c(-10,10), xlim=c(-10,5), xlab="x1",ylab="x2", #format axes
         main = paste0("Trajectory and Rejections for ", names(GridSearchOutput)[[1]], ", Series ", 1) )
    points(GridSearchOutput[[1]][[2]]$trajectory[,1:2],col="blue", pch=16, lwd = 1) #add accepted traje
    points(GridSearchOutput[[1]][[3]]$trajectory[,1:2],col="red", pch=16, lwd = 1)
    points(GridSearchOutput[[1]][[4]]$trajectory[,1:2],col="gray", pch=16, lwd = 1)
    points(GridSearchOutput[[1]][[5]]$trajectory[,1:2],col="green", pch=16, lwd = 1)
    legend("bottomleft",legend=c("Series 1", "Series 2", "Series 3", "Series 4", "Series 5"),
           col=c("orange", "blue", "red", "gray", "green"),pch=c(1,16))
```

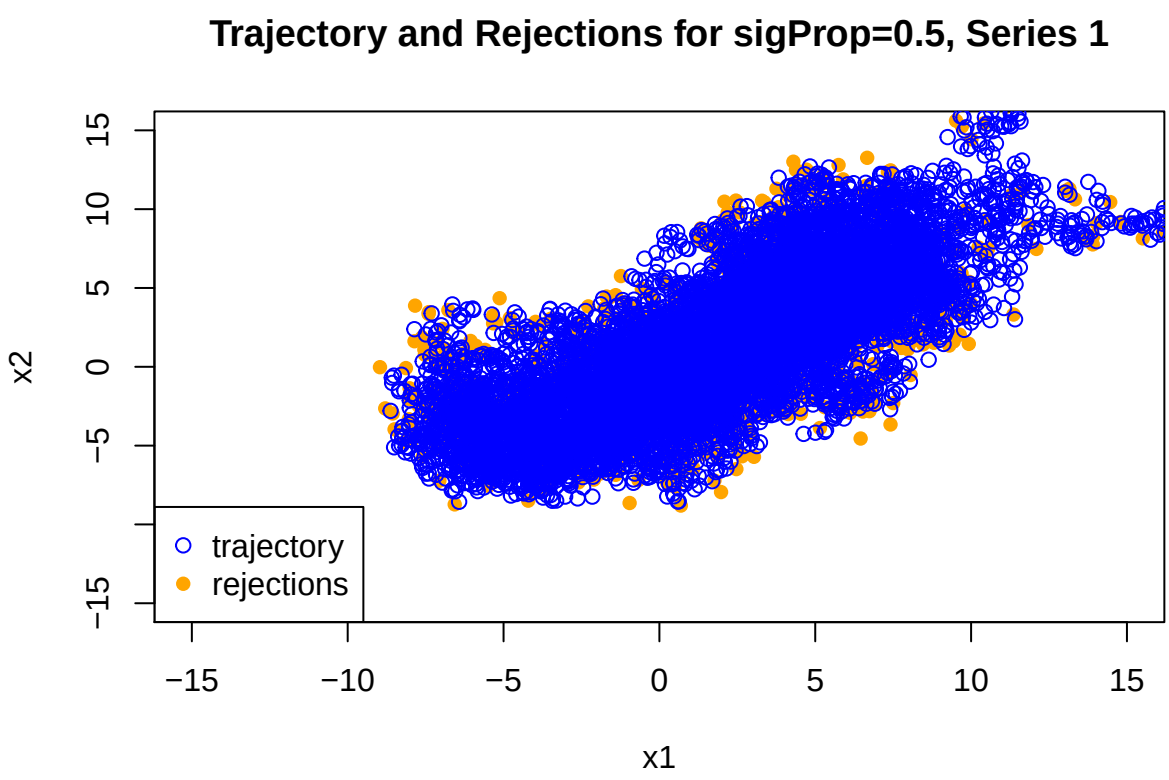


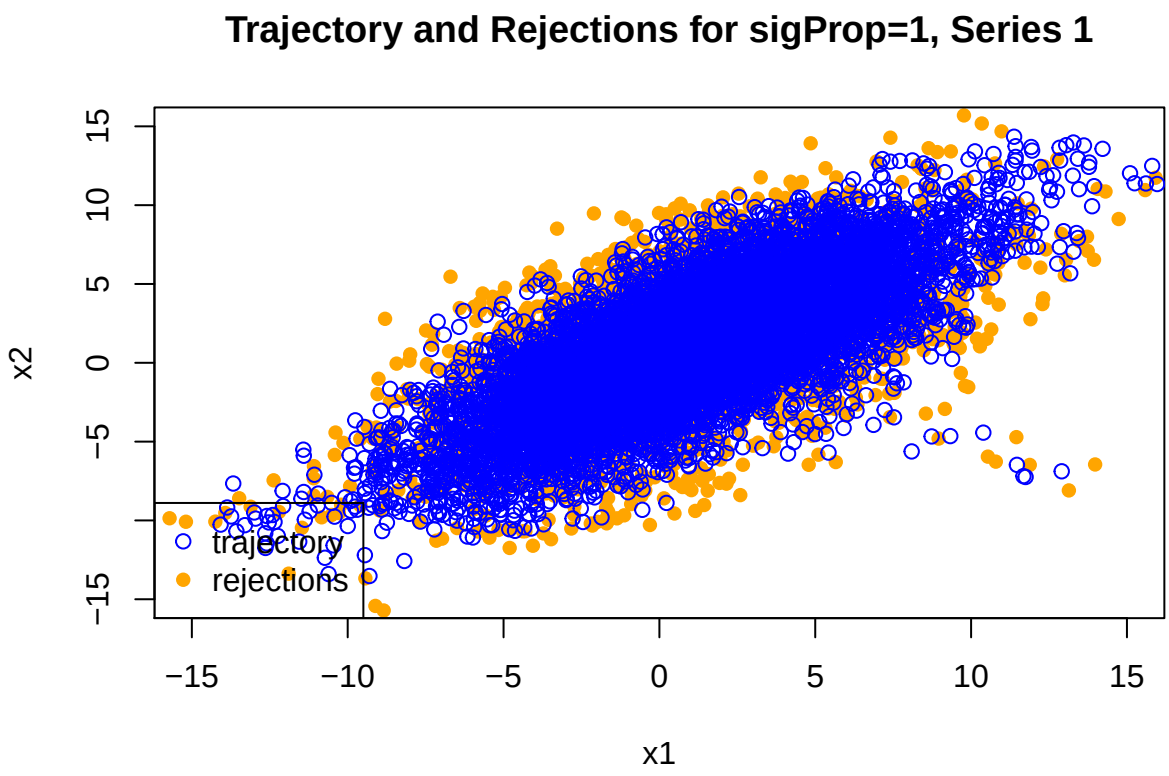
The graphs below show an example trajectory (and rejections) for 5 different values of sigProp, the standard deviation of the distribution used to select candidate points for the Metropolis algorithm. The values of sigProp evaluated are: 0.05, 0.1, 0.5, 1.0, and 2.0.

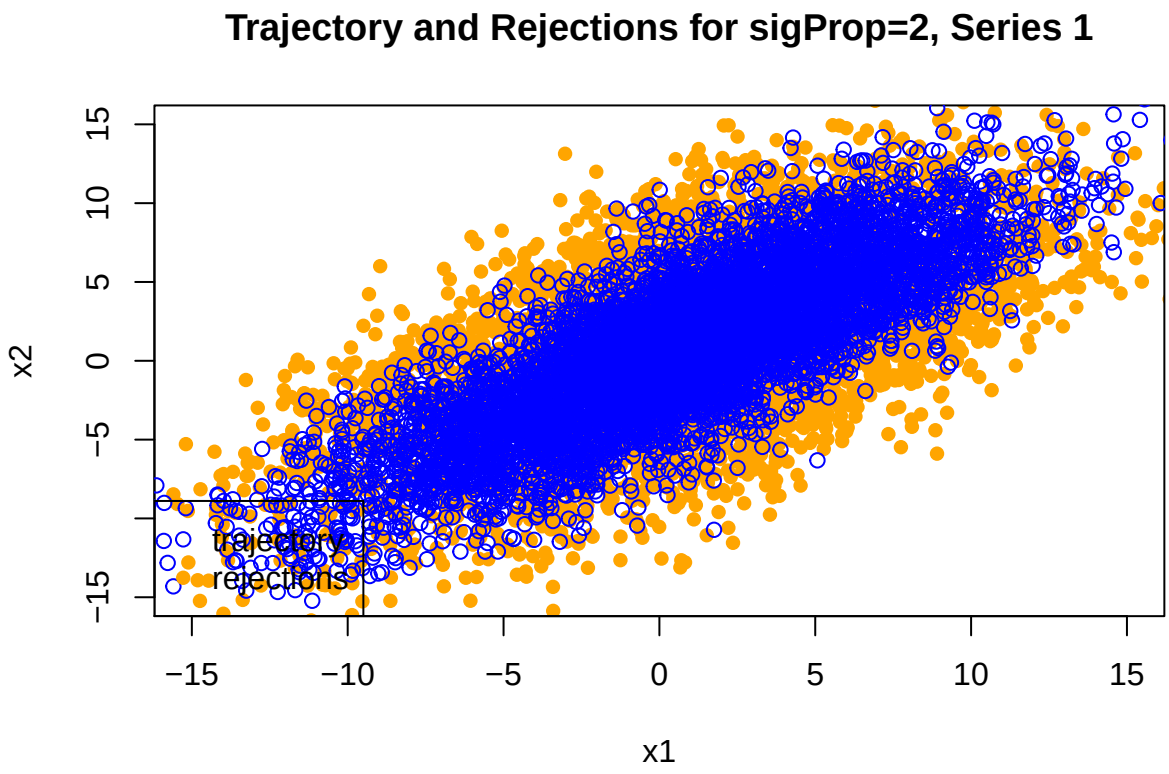
```
for (i in 1:5) {
  plot(GridSearchOutput[[i]][[1]]$rejections[,1:2],#plot rejections as bottom-most layer
       col="orange",pch=16, lwd = 0.1, #format for rejection points
       ylim=c(-15,15), xlim=c(-15,15), xlab="x1",ylab="x2", #format axes
       main = paste0("Trajectory and Rejections for ", names(GridSearchOutput)[[i]], ", Series ", 1) )
  points(GridSearchOutput[[i]][[1]]$trajectory[,1:2],col="blue", pch = 1, lwd = 1) #add trajectory
  legend("bottomleft",legend=c("trajectory","rejections"),col=c("blue","orange"),pch=c(1,16))
}
```


Trajectory and Rejections for sigProp=0.05, Series 1
x2
15
10
5
0
−5
−15
trajectory
rejections
−15
−10
−5
0
5
10
15
x1


Trajectory and Rejections for sigProp=0.1, Series 1
x2
15
10
5
0
−5
−15
trajectory
rejections
−15
−10
−5
0
5
10
15
x1


Trajectory and Rejections for sigProp=0.5, Series 1
x2
15
10
5
0
−5
−15
trajectory
rejections
−15
−10
−5
0
5
10
15
x1


Trajectory and Rejections for sigProp=1, Series 1
x2
x1
trajectory
rejections
−15
−10
−5
0
5
10
15



Results and Conclusions:

Based upon the graphs above, the following conclusions may be drawn:

- It is important to select a standard deviation for the proposal distribution that is appropriate for the size of the feature space, the range of possible initial values, and the distribution being studied.
- If the sigProp is too small, the trajectory may spend a long time exploring low-density regions before it even gets to a region of high density, resulting in low efficiency. After reaching an area of high density, the trajectory may be "stuck" there, leaving much of the feature space unexplored. Furthermore, successive points in the trajectory are likely to be highly correlated, leading to "clumpy" behavior and a lower effective sample size.
- Conversely, if sigProp is too large, the algorithm may also be less efficient, because the candidate points are likely to "overshoot" the high-density regions, meaning that many iterations of the algorithm will result in rejected points.
- For the distribution parameters assumed in this example, it appears that values of sigProp between 0.5 and 1.0 appear to give good results. Comparing the plots for these values of sigProp to the bivariate normal density plot generated earlier shows that the algorithm appears to do a good job of approximating the density of the original function.