PSTAT 134 HW 4

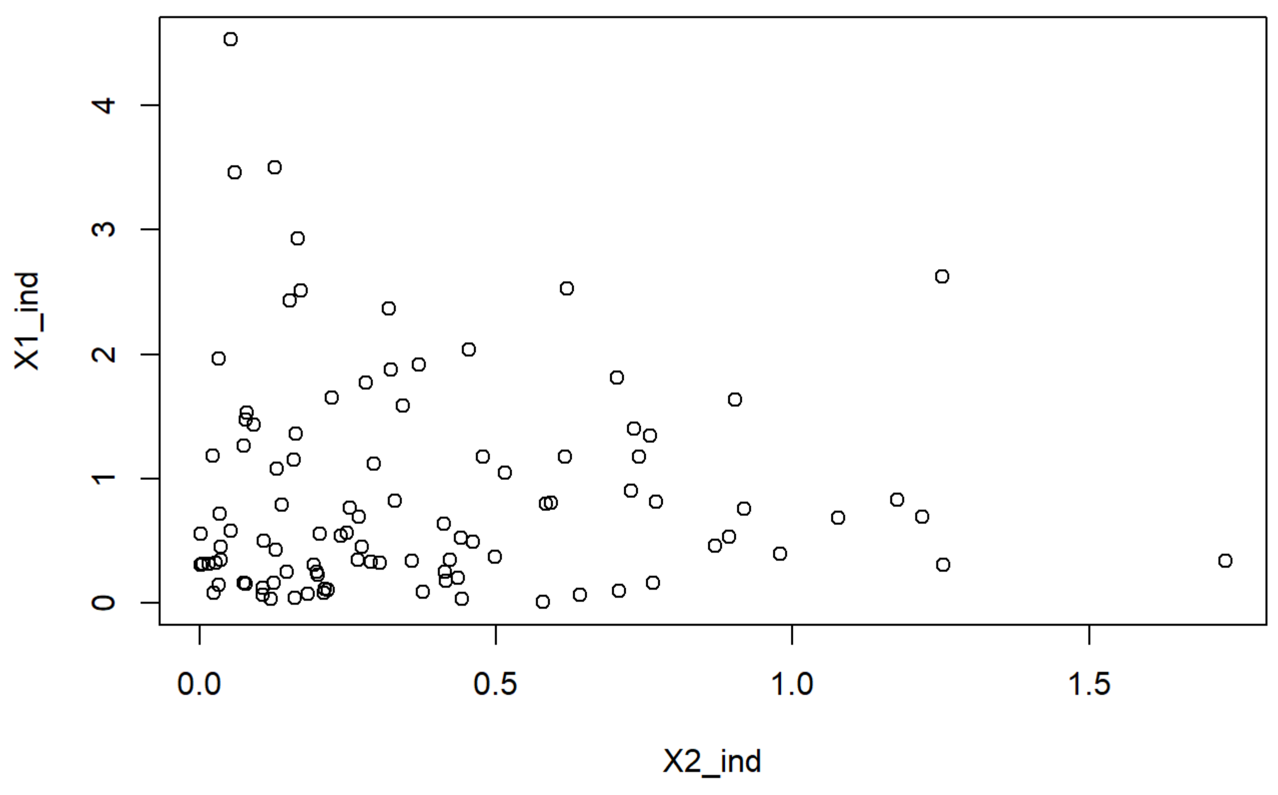
TJ Sipin

2022-07-27

Problem 1 (Gaussian Copula).

We derive an algorithm to generate two RVs $X_1 \sim \mathrm{Exp}(1)$ and $X_2 \sim \mathrm{Exp}(3)$.

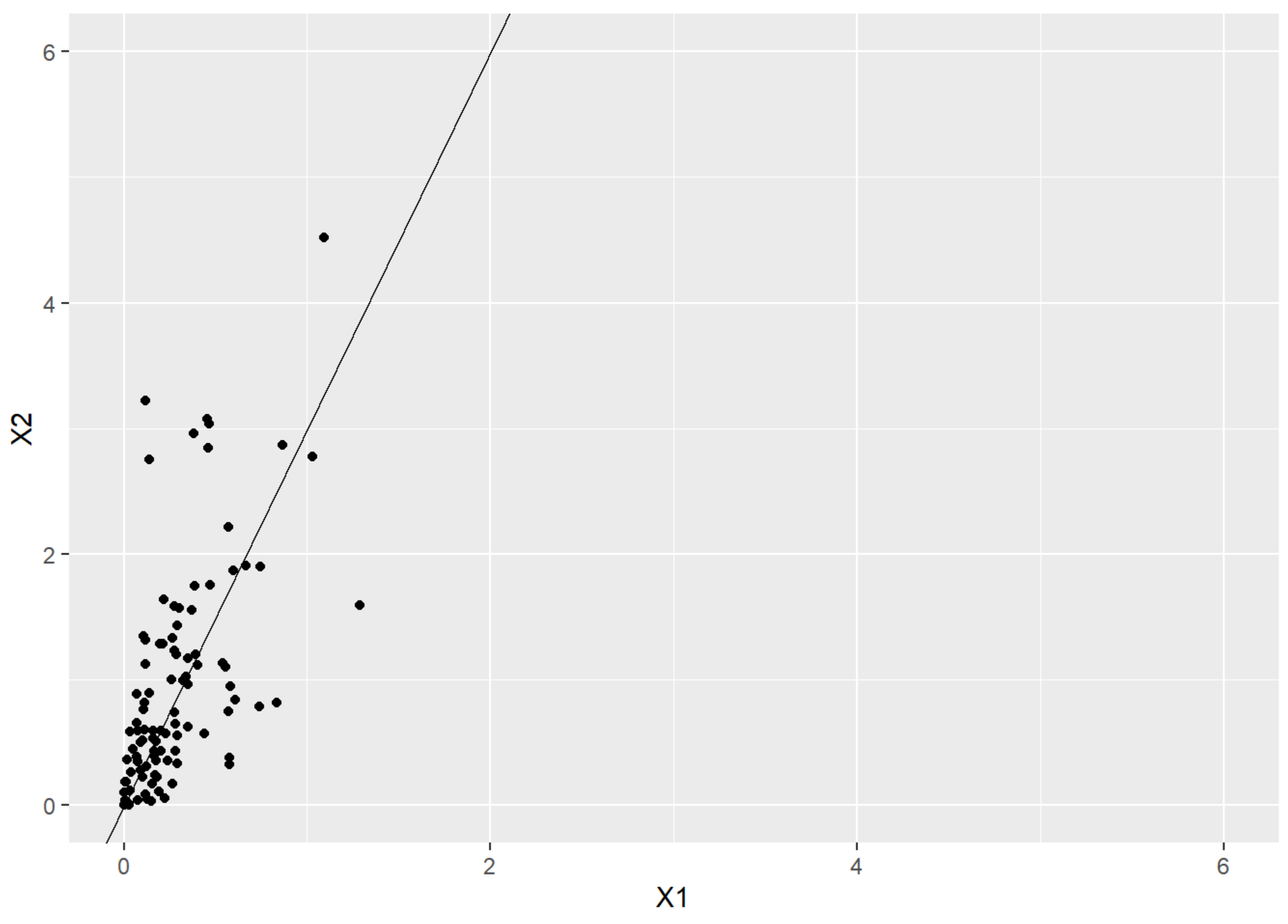
```
# independent case: get input from uniform distribution
U1 <- runif(100)

lambda_1 <- 1
X1_ind <- -1/lambda_1 * log(1 - U1)

U2 <- runif(100)
lambda_2 <- 3
X2_ind <- -1/lambda_2 * log(1 - U2)

p1 <- hist(X1_ind, plot = FALSE, breaks = 100)
p2 <- hist(X2_ind, plot = FALSE, breaks = 100)

plot(X2_ind, X1_ind)
```



This plot is relatively homoscedastic, though there is more concentration closer to the origin, due to the inverse CDF $F^{-1}(U) = \frac{1}{\lambda}\log(1 - U)$ used to obtain our samples.

```r
# helper function for n dim gaussian vector generation

n_dim_gaussvec <- function(dim, mean_vec, cov_matrix){
  A <- t(chol(cov_matrix))
  vec <- A %*% rnorm(dim) + mean_vec
  return(vec)
}


mu = rep(0,2)
covmat <- matrix(0.7, nrow = 2, ncol = 2)
diag(covmat) <- 1
x <- matrix(0, nrow = 100, ncol = 2)

for (i in 1:100){
  # Step 1: generate 2 dim gaussian vector
  gauss_vec <- n_dim_gaussvec(2, mu, covmat)
  # Step 2: Compute cdf of each sample
  # Recall we assumed each component is normal mean 0 and var 1 due to mean
  # vec and diag of cov matrix
  unif_samples <- pnorm(gauss_vec)
  # Step 3: Use inverse CDF to obtain correlated 2 samples
  x[i,1] <- -1/lambda_2 * log(1 - unif_samples[1])
  x[i,2] <- -1/lambda_1 * log(1 - unif_samples[2])
}

x_mat <- matrix(x, ncol = 2)
x_df <- data.frame(x_mat)

ggplot(data = x_df) +
  geom_point(aes(x = X1,
                y = X2)) +
  scale_x_continuous(limits = c(0,6)) +
  scale_y_continuous(limits = c(0,6)) +
  geom_abline(intercept = 0,
             slope = 3)
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



Here, we see that there is less homoscedasticity from the correlation coefficient $\rho(X_1, X_2) = 0.7$ We still see that the majority of points tend to group around the origin, again from the inverse CDF used to obtain the samples.

Problem 2 (Monte Carlo simulation)

To solve this problem, we first recognize that the area encompassed by CKD is the same as the area encompassed by $CAD \cap \not{A}KD$. This can be done using Monte Carlo simulation, producing a ratio of points within a drawn quarter radius $\sqrt{x^2 + y^2}$. Then subtract from it the ratio of points within the former quarter radius and the quarter radius given by $\sqrt{(x-1)^2 + y^2}$. Note that $x, y \in [0, 1]$.

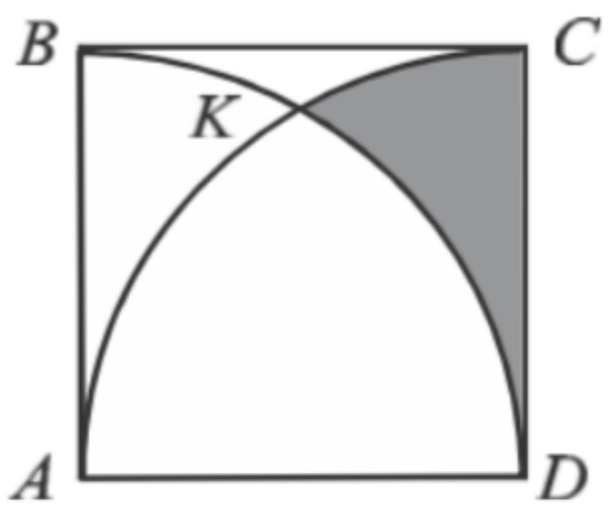


Figure 1: Unit-length square with two quarter circles

```
u1 <- runif(1000)
u2 <- runif(1000)
r1 <- sqrt(u1**2 + u2**2)
r2 <- sqrt((u1-1)**2 + u2**2)
CAD <- ifelse(r2 <= 1, 1, 0)
AKD <- ifelse((r1 <= 1) & (r2 <=1), 1, 0)
sum(CAD)/1000 - sum(AKD)/1000
```

```
## [1] 0.148
```

Here, we see that the approximate area is about 0.162.

Problem 3 (M-H & Gibbs Sampler)

Our target density is $f(x,y,z) = C\exp(-(x+y+z+xy+yz+xz))$ for some unknown but fixed constant C and $x,y,z>0$.

a.

We use the Metropolis Hastings Sampler to estimate $\mathbb{E}(XYZ)$. In this approach, we use the standard normal transition kernel to simplify our ratio r_n.

```
set.seed(1)
target = function(x,y,z){
  return(ifelse((x<0) || (y<0) || (z<0),
              0,
              exp(-(x+y+z+x*y+y*z+x*z))))
}

# niter = number of iterations, start_pt = starting point of your MCMC
# proposal sd = sd of transition Kernel which is Gaussian.



MHsim <- function(target_density,niter,start_pt, proposal_sd){
  x = rep(0,niter)
  y = rep(0,niter)
  z = rep(0,niter)
  X = c(x,y,z)
  X[1] = start_pt
  for(i in 2:niter){
    current_x = x[i-1]
    current_y = y[i-1]
    current_z = z[i-1]
    proposed_x = rnorm(1,mean=current_x,sd=proposal_sd)
    proposed_y = rnorm(1,mean=current_y,sd=proposal_sd)
    proposed_z = rnorm(1,mean=current_z,sd=proposal_sd)
    r_n = target_density(proposed_x,
                         proposed_y,
                         proposed_z)/target_density(current_x,
                                                    current_y,
                                                    current_z)
    # r_n is acceptance probability
    # if higher than Unif[0,1], choose to move to diff state

    if(runif(1)<r_n){
      x[i] = proposed_x
      y[i] = proposed_y
      z[i] = proposed_z
      } else {
        # else, stay
        x[i] = current_x
        y[i] = current_y
        z[i] = current_z
      }
  }
  return(list(x,y,z))
}

set.seed(1)
niter <- 10000
burn_in <- 500
a <- MHsim(target, niter, 1, 1)
theta <- vector()
for (i in seq(1,niter)) {
  theta[i] = a[[1]][i]*a[[2]][i]*a[[3]][i]
}
result <- (1/(niter-500))*sum(theta[(burn_in+1):niter])
result
```

```
## [1] 0.08715248
```

From the Metropolis Hastings Sampler, we estimate that $\mathbb{E}(XYZ) \approx 0.087$ with burn-in period set to 500 and number of iterations set to 10,000. We use the formula $\mathbb{E}(XYZ) \approx \frac{\sum_{i=501}^{10,000} X_i Y_i Z_i}{10,000-500}$.

b.

The Gibbs Sampling algorithm is the exact same as the Metropolis-Hastings algorithm, except at each time step t, we always move to another state. After randomly drawing the index i of components to choose which vector component to move, we sample from the conditional distribution $p(x_i|x_{-i})$, where x_{-i} is the set of all components not x_i.

We have found that the following conditional probabilities:

$$\begin{aligned} f(x|y,z) &= e^{-x(1+y+z)}(1+y+z) \\ &\sim \text{Exp}(\lambda = 1+y+z) \\ f(y|x,z) &= e^{-y(1+x+z)}(1+x+z) \\ &\sim \text{Exp}(\lambda = 1+x+z) \\ f(z|x,y) &= e^{-z(1+x+y)}(1+x+y) \\ &\sim \text{Exp}(\lambda = 1+x+y). \end{aligned}$$

So depending on which index we randomly choose, we sample the new state from one of the above distributions.

```
gibbs <- function(niter,start_pt){
  X = c(start_pt[1], numeric(niter))
  Y = c(start_pt[2], numeric(niter))
  Z = c(start_pt[3], numeric(niter))



  for(i in 2:niter){
    j <- sample(c(1,2,3),1)
    if (j == 1){
      X[i] <- rexp(n = 1, rate = (1+Y[i-1]+Z[i-1]))
      Y[i] <- Y[i-1]
      Z[i] <- Z[i-1]
      } else if (j == 2){
        Y[i] <- rexp(n = 1, rate = (1+X[i-1]+Z[i-1]))
        X[i] <- X[i-1]
        Z[i] <- Z[i-1]
        } else if (j == 3){
          Z[i] <- rexp(n = 1, rate = (1+Y[i-1]+X[i-1]))
          X[i] <- X[i-1]
          Y[i] <- Y[i-1]
        }
    }
  return(list(X,Y,Z))
}

set.seed(1)
niter <- 10000
burn_in <- 500
b <- gibbs(niter, c(1,1,1))



theta_b <- vector()
for (i in 1:niter) {
  theta_b[i] = b[[1]][i]*b[[2]][i]*b[[3]][i]
}
result <- (1/(niter-500))*sum(theta_b[(burn_in+1):niter])

result
```

```
## [1] 0.08905299
```

From the Gibbs Sampler, we estimate that $\mathbb{E}(XYZ) \approx 0.089$, with initial point at $(1,1,1)$ using the same burn-in period and formula from part (a). This is very similar to the result from the Metropolis-Hastings Sampler, so we are satisfied.

Problem 4 (SIR)

We use the SIR algorithm to generate some permutation $(X_1, \ldots, X_{100})$ where $X_i = 1, \ldots, 100$ and $i = 1, \ldots, 100$ and $X_i \neq X_j$ if $i \neq j$. Our target distribution is a random vector from a conditional distribution $L|T > 285000$, where $T \equiv \sum_{i=1}^{100} iX_i$.

First, we take $n = 10000$ samples to put in our `priors` matrix of dimension $100 \times n$, where each column is a random permutation. We would take $100!$ samples if we wanted a complete set of all permutations, however, life doesn't always give what we need due to computing limitations. We calculate T for each sample and save it in a vector. Then we compute the weights, an indicator function on whether each T at each index i is greater than our threshold 285000, and normalize them to use when sampling.

```
### create X1,...,X100 first
set.seed(123)


# algorithm start
# step 1: sample from g_0(L) 10000 times
# aka sample L 10000 times
n = 10000

priors <- matrix(0, nrow = 100, ncol = n)
T <- c(numeric(n))
for (i in 1:n){
  T[i] = 0
  priors[,i] <- sample(seq(1,100), 100)
  for (j in seq(1,100)) {
    T[i] = T[i] + (j * priors[,i][j])
  }
}

# step 2: compute weights, note wi = f_0/g_0 =
weights = rep(0, n)
for (i in 1:n) {
  if (T[i] > 285000) {
    weights[i] = 1
  }
}

# normalize weights
weights = weights / sum(weights)

# step 3: get 1 sample from posterior f(L|T > 285000)
posteriors <- priors[,sample(ncol(priors), size = 1, replace = TRUE, prob = weights)]

posteriors
```

```
##   [1]   6  13   2  28  45  88  89   8   9  30  17  20  42  83  94   3  16  92
##  [19]  25  36  35  15  34   1  77  70  71  44  32  14  78  43  73  29  65  61
##  [37]   4  54  67  76  47   5  72  22  12  11  90  58  46  41  86  23  55  48
##  [55]  49  84  85  37   7  64  50  62  24  91  52  19  51  75  63  10  26  87
##  [73]  33  21  66  99  69  74  31  39  57  82  95  96  27  40  68  53  59  80
##  [91]  38  97  98 100  81  60  18  56  79  93
```