Radiata Pine Example (Lecture 2)

Leah South

Our running example is a simple linear regression model about strength of radiata pine trees.

This is a continuation of the first lecture code.

Below are the R packages we'll be using in this document.

```
library(MASS) # multivariate normal
library(coda) # assessing convergence and sample quality
library(psych) # bivariate plots
library(mcmcse) # Multivariate ESS
library(mvtnorm) # For multivariate normal
library(KSD) # For KSD
library(ZVCV) # For control variates
```

```
MALA_fn <- function(theta0,sigma_mala,iters,varNames){
  d <- length(theta0)
  samples <- d_loglike <- d_logprior <- matrix(NaN,nrow=iters,ncol=d)
  loglike <- logprior <- rep(NaN,iters)
  samples[1,] <- theta0

  temp <- der_loglike(samples[1,],radiata)
  loglike[1] <- temp$loglike
  d_loglike[1,] <- temp$der_loglike

  temp <- der_logprior(samples[1,],radiata)
```

```
  logprior[1] <- temp$logprior
  d_logprior[1,] <- temp$der_logprior

  for (i in 1:(iters-1)){
    mymean <- samples[i,] + 1/2*sigma_mala%*%(d_loglike[i,]+d_logprior[i,])
    samples_prop <-  rmvnorm(n=1, mymean, sigma_mala)

    temp <- der_loglike(samples_prop,radiata)
    loglike_prop <- temp$loglike
    d_loglike_prop <- temp$der_loglike

    temp <- der_logprior(samples_prop,radiata)
    logprior_prop <- temp$logprior
    d_logprior_prop <- temp$der_logprior

    mymean_prop <- t(samples_prop) + 1/2*sigma_mala%*%(d_loglike_prop+d_logprior_prop)

    transition_totheta <- dmvnorm(t(samples[i,]),mymean_prop,sigma_mala,log=TRUE)
    transition_toprop <- dmvnorm(samples_prop,mymean,sigma_mala,log=TRUE)

    log_mh <- (loglike_prop - loglike[i]) + logprior_prop - logprior[i] + transition_totheta - transitio

    # determine whether to accept or reject
    if (exp(log_mh) > runif(1,0,1)){
      # then accept the proposal
      samples[i+1,] <- samples_prop
      loglike[i+1] <- loglike_prop
      logprior[i+1] <- logprior_prop
      d_loglike[i+1,] <- d_loglike_prop
      d_logprior[i+1,] <- d_logprior_prop
    } else{
      samples[i+1,] <- samples[i,]
      loglike[i+1] <- loglike[i]
      logprior[i+1] <- logprior[i]
      d_loglike[i+1,] <- d_loglike[i,]
      d_logprior[i+1,] <- d_logprior[i,]
    }
  }
  samples <- as.mcmc(samples)
  varnames(samples) <- varNames

  return(list(samples=samples,loglike=loglike,logprior=logprior,der_loglike=d_loglike,der_logprior=d_log

}

ULA_fn <- function(theta0,cov_ULA,iters,varNames){
  d <- length(theta0)
  samples <- d_loglike <- d_logprior <- matrix(NaN,nrow=iters,ncol=d)
  samples[1,] <- theta0

  d_loglike[1,] <- der_loglike(samples[1,],radiata)$der_loglike
  d_logprior[1,] <- der_logprior(samples[1,],radiata)$der_logprior
```

```
  for (i in 1:(iters-1)){
    mymean <- samples[i,] + 1/2*cov_ULA%*%(d_loglike[i,]+d_logprior[i,])
    samples[i+1,] <-  rmvnorm(n=1, mymean, cov_ULA)

    d_loglike[i+1,] <- der_loglike(samples[i+1,],radiata)$der_loglike
    d_logprior[i+1,] <- der_logprior(samples[i+1,],radiata)$der_logprior

  }
  samples <- as.mcmc(samples)
  varnames(samples) <- varNames

  return(list(samples=samples,der_loglike=d_loglike,der_logprior=d_logprior))
}
```

```
# Loading in the data
load("radiata.rda")

# A function to compute the log likelihood for this example
loglike <- function(part_vals,options){
  #Getting data from list
  x <- options$x1
  y <- options$y

  #Log likelihood for the inputted data and parameters
  mean_reg <- part_vals[1]+part_vals[2]*(x-mean(x))
  loglike <- sum(dnorm(y,mean_reg,sqrt(exp(part_vals[3])),log = TRUE))
  return(loglike)
}

# A function to compute the log prior density for this example
logprior <- function(part_vals,options){

  #Parameters of inverse gamma (the prior for parameter 3 in this example)
  a <- 3
  b <- (2*300^2)^(-1)

  #Log of the prior
  logprior <- dnorm(part_vals[1],3000,10^3,log = TRUE) +
    dnorm(part_vals[2],185,10^2,log =TRUE) +
    part_vals[3]-1/b/exp(part_vals[3]) -
    log(gamma(a))-a*log(b)-part_vals[3]*(a+1)

  return(logprior)
}


# A function to compute the gradients of the log likelihood
# for this example
der_loglike <- function(part_vals,options){
  #Getting data from list
  x <- options$x1
  y <- options$y

  #Log likelihood for the inputted data and parameters
```

```
  mean_reg <- part_vals[1]+part_vals[2]*(x-mean(x))
  loglike <- sum(dnorm(y,mean_reg,sqrt(exp(part_vals[3])),log = TRUE))

  diffy <- y-mean_reg
  diffx <- x-mean(x)

  der_loglike = c(sum(diffy/exp(part_vals[3])),
                  sum(diffx*diffy/exp(part_vals[3])),
                  sum(diffy^2/exp(part_vals[3])/2-1/2))

  return (list(loglike = loglike, der_loglike = der_loglike))
}

# A function to compute the gradients of the log prior
# for this example
der_logprior <- function(part_vals,options){

  #Parameters of inverse gamma (the prior for parameter 3 in this example)
  a <- 3
  b <- (2*300^2)^(-1)

  #Log of the prior
  logprior <- dnorm(part_vals[1],3000,10^3,log = TRUE) +
    dnorm(part_vals[2],185,10^2,log =TRUE) +
    part_vals[3]-1/b/exp(part_vals[3]) -
    log(gamma(a))-a*log(b)-part_vals[3]*(a+1)

  der_logprior <- c((3000-part_vals[1])/10^6,
                    (185-part_vals[2])/10^4,
                    1/b/exp(part_vals[3])-a)

  return (list(logprior = logprior, der_logprior = der_logprior))
}

# A function to simulate data from the prior
# (for overdispersed initial starts in MCMC)
simprior <- function(N,options){

  #Drawing N samples from the prior for each parameter
  part_vals <- matrix(rep(0,N*3),nrow=N,ncol=3)
  part_vals[,1] <- rnorm(N,mean=3000,sd=10^3)
  part_vals[,2] <- rnorm(N,mean=185,sd=10^2)
  part_vals[,3] <- log(rgamma(N,3,(2*300^2))^(-1))

  return(part_vals)

}

# Names of the three variables
varNames <- c("alpha","beta","sigma^2")

# An efficient random walk covariance
sigma_rw <- c(2660,134,0.044)*diag(3)
```

Comparing a Single Run

We can use the KSD package in R, although this package doesn't implement the inverse multiquadric (IMQ) kernel that gives better theoretical properties and scaling with dimensions.

An implementation of the KSD with an IMQ kernel is also in the R code below.

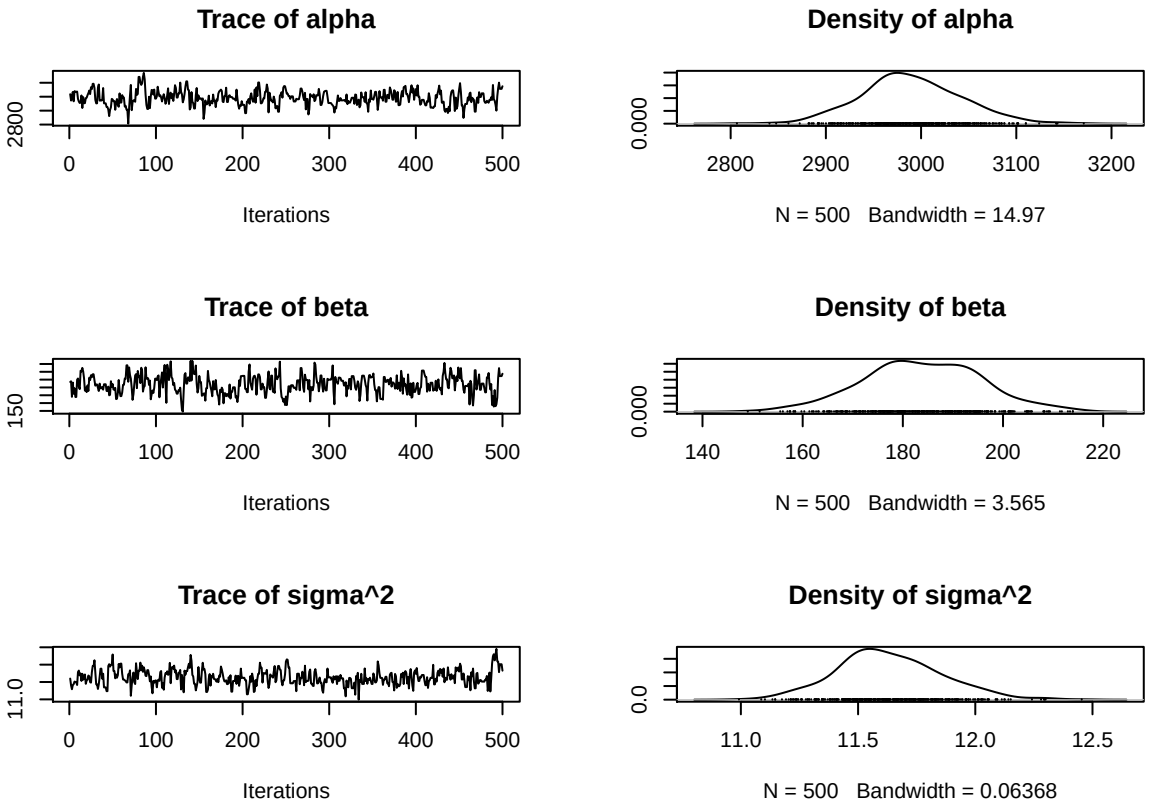
```
kernal_IMQ <- function(x, beta, band_width){
  s_band_width <- sqrt(band_width)
  x = x / s_band_width

  k0 = (1+x)^beta
  k1 <- beta*(1+x)^(beta - 1) / s_band_width
  k2 <- beta*(beta-1)*(1+x)^(beta - 2) / band_width

  res <- list(k0 = k0,k1 = k1,k2 = k2)
  return(res)
}

imqKSD <- function(samples, grads, band_width = 1){
  n <- nrow(samples)
  d <- ncol(samples)

  norms <- rowSums(samples^2)
  norms <- norms*matrix(1, n, n)
  mu <- tcrossprod(samples, samples)
  norms <- norms - mu
  norms <- norms + t(norms)
  norms <- norms - diag(diag(norms))

  sample_grad <- rowSums(samples*grads)
  sample_grad <- sample_grad*matrix(1, n, n)
  mu <- tcrossprod(samples, grads)
  sample_grad <- sample_grad - mu
  sample_grad <- sample_grad + t(sample_grad)
  sample_grad <- sample_grad - diag(diag(sample_grad))

  k_res <- kernal_IMQ(norms, -0.5, band_width)
  k_0 <- k_res$k0; k_1 <- k_res$k1; k_2 <- k_res$k2
  k_gram <- -2*d*k_1 - 4*norms*k_2
  k_gram <- k_gram - 2*k_1*sample_grad
  k_gram <- k_gram + tcrossprod(grads, grads)*k_0

  res <- sqrt(sum(k_gram))/n
  return(res)
}
```

Running a MALA and ULA chain

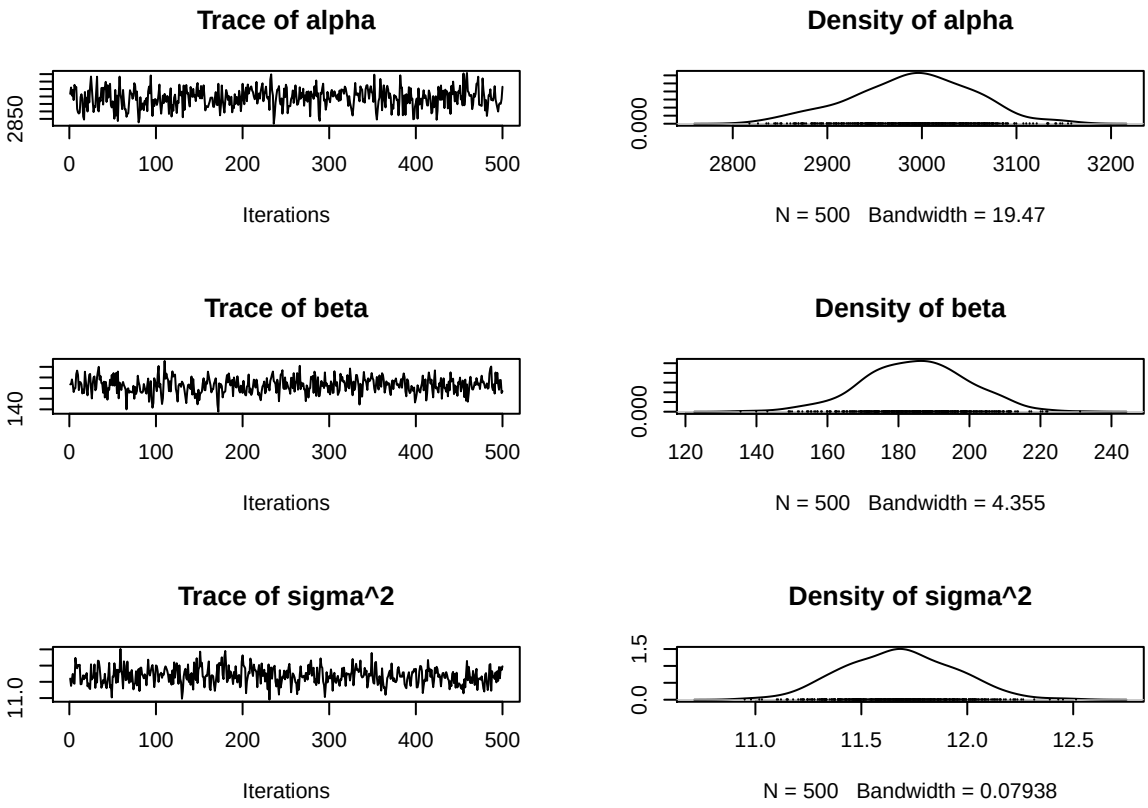
500 iterations for each chain.

```
set.seed(1)
its <- 500 # number of MCMC iterations
theta0 <- c(3018,187,11.6)
chain_MALA <- MALA_fn(theta0,sigma_rw,its,varNames)
chain_ULA <- ULA_fn(theta0,1.5*sigma_rw,its,varNames)
```

```
plot(chain_MALA$samples)
```




```
plot(chain_ULA$samples)
```



Calculating the KSD

Calculating the kernel Stein discrepancy for both methods using two different kernels. The IMQ kernel results are more trustworthy. It looks like ULA is a bit worse with both methods.

```
# Getting the two KSD's for MALA
samples <- as.matrix(chain_MALA$samples)
gradients <- chain_MALA$der_loglike + chain_MALA$der_logprior

KSD(samples, gradients)$ksd
```

```
## [1] 0.07093413
```

```
imqKSD(samples, gradients)
```

```
## [1] 0.2760284
```

```
# Getting the two KSD's for ULA
samples <- as.matrix(chain_ULA$samples)
gradients <- chain_ULA$der_loglike + chain_ULA$der_logprior

KSD(samples, gradients)$ksd
```

```
## [1] 0.52392
```

```
imqKSD(samples, gradients)
```

```
## [1] 0.3445914
```

Estimating expectations with control variates

Now we'll estimate the marginal posterior means for our three parameters. In practice we only want to use one estimate, and a control variate approach is typically going to be better. Take a look at the multiple chains section of this document an illustration.

Notice that our gradients correspond to the samples on the log scale, so we aren't exponentiating `samples[,3]`.

```
# A gold standard of approximation that you wouldn't normally have
gold_standard <- c(2991.992, 184.609, 112689.957) # from a 1 million iteration run.

### MALA
samples <- as.matrix(chain_MALA$samples)
gradients <- chain_MALA$der_loglike + chain_MALA$der_logprior
integrand <- samples
integrand[,3] <- exp(integrand[,3])

# Vanilla Monte Carlo integration
colMeans(integrand)
```

```
##      alpha       beta    sigma^2
##   2989.263    183.832 115544.617
```

```
# ZV-CV with a first order polynomial
zvcv(integrand, samples, gradients,
     options = list(polyorder = 1, regul_reg = FALSE))$expectation
```

```
##         alpha    beta  sigma^2
## [1,] 2993.356 184.563 113249.4
```

```
# Control Functionals
CF_crossval(integrand, samples, gradients, kernel_function = "RQ",
            sigma_list = list(0.001,0.01,0.1,1,10), folds = 2)$expectation
```

```
##            [,1]
## [1,]   2990.949
## [2,]    183.481
## [3,] 109732.507
```

```
# Semi-Exact Control Functionals
SECF_crossval(integrand, samples, gradients, polyorder = 1, kernel_function = "RQ",
              sigma_list = list(0.001,0.01,0.1,1,10), folds = 2)$expectation
```

```
##             [,1]
## [1,]   2992.0162
## [2,]    184.5534
## [3,] 113294.8763
```

```
### ULA
samples <- as.matrix(chain_ULA$samples)
gradients <- chain_ULA$der_loglike + chain_ULA$der_logprior
integrand <- samples
integrand[,3] <- exp(integrand[,3])

# Vanilla Monte Carlo integration
colMeans(integrand)
```

```
##       alpha        beta     sigma^2
##   2990.3627    184.8737 122390.9021
```

```r
# ZV-CV with a first order polynomial
zvcv(integrand, samples, gradients,
     options = list(polyorder = 1, regul_reg = FALSE))$expectation
```

```
##         alpha     beta  sigma^2
## [1,] 2991.001 184.4521 118438.2
```

```r
# Control Functionals
CF_crossval(integrand, samples, gradients, kernel_function = "RQ",
            sigma_list = list(0.001,0.01,0.1,1,10), folds = 2)$expectation
```

```
##             [,1]
## [1,]   2990.3185
## [2,]    184.8737
## [3,] 111048.6479
```

```r
# Semi-Exact Control Functionals
SECF_crossval(integrand, samples, gradients, polyorder = 1, kernel_function = "RQ",
              sigma_list = list(0.001,0.01,0.1,1,10), folds = 2)$expectation
```

```
##             [,1]
## [1,]   2991.7509
## [2,]    184.5052
## [3,] 118426.4912
```

Comparing Multiple Runs (for illustration only)

Running multiple chains

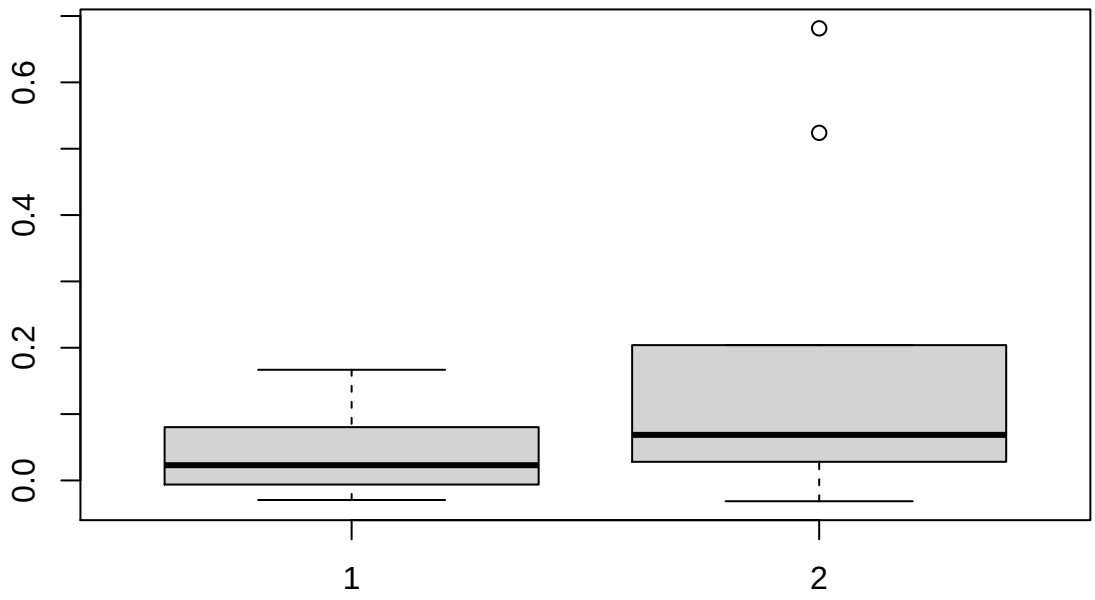
We run 10 chains with 500 iterations each for both for MALA and ULA.

```
set.seed(1)
n_reps <- 10 # number of chains
its <- 500 # number of MCMC iterations
theta0 <- c(3018,187,11.6)
chains_MALA <- chains_ULA <- list()
for (i in 1:n_reps){
  chains_MALA[[i]] <- MALA_fn(theta0,sigma_rw,its,varNames)
  chains_ULA[[i]] <- ULA_fn(theta0,1.5*sigma_rw,its,varNames)
}
```

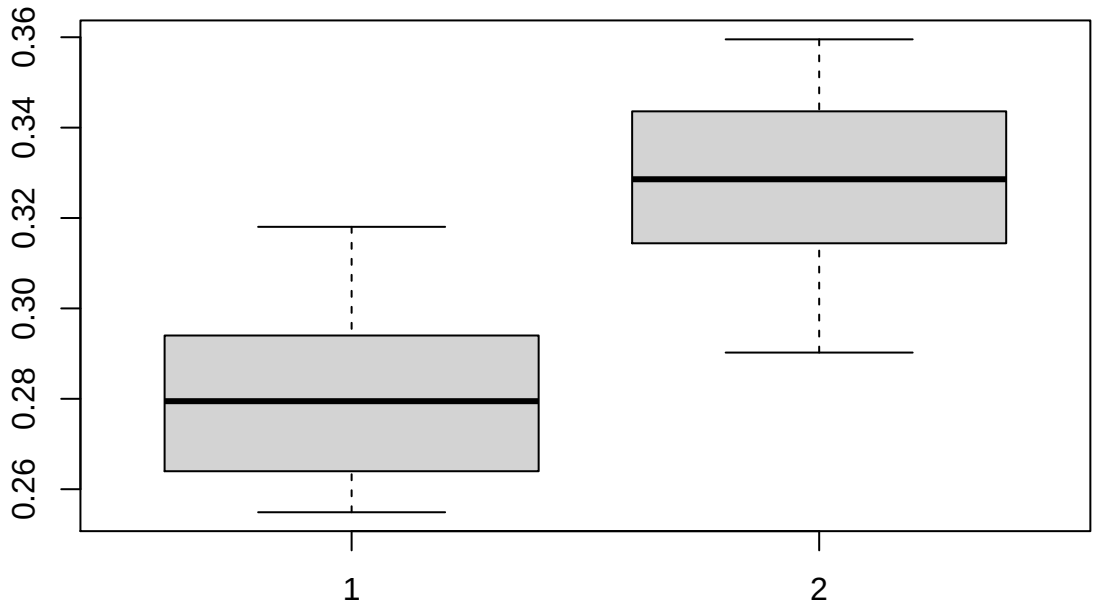
Calculating the KSD

Calculating the kernel Stein discrepancy for all chains using two different kernels. The IMQ kernel results are more trustworthy. It looks like ULA is a bit worse with both methods.

```
KSD_MALA_gaussian <- KSD_ULA_gaussian <- rep(NaN,n_reps)
KSD_MALA_imq <- KSD_ULA_imq <- rep(NaN,n_reps)
for (i in 1:n_reps){
  # Calculating the KSDs for MALA
  samples <- as.matrix(chains_MALA[[i]]$samples)
  gradients <- chains_MALA[[i]]$der_loglike + chains_MALA[[i]]$der_logprior

  KSD_MALA_gaussian[i] <- KSD(samples, gradients)$ksd
  KSD_MALA_imq[i] <- imqKSD(samples, gradients)

  # Calculating the KSDs for ULA
  samples <- as.matrix(chains_ULA[[i]]$samples)
  gradients <- chains_ULA[[i]]$der_loglike + chains_ULA[[i]]$der_logprior

  KSD_ULA_gaussian[i] <- KSD(samples, gradients)$ksd
  KSD_ULA_imq[i] <- imqKSD(samples, gradients)
}

# Boxplots of the KSD
boxplot(KSD_MALA_gaussian,KSD_ULA_gaussian)
```




```
boxplot(KSD_MALA_imq,KSD_ULA_imq)
```



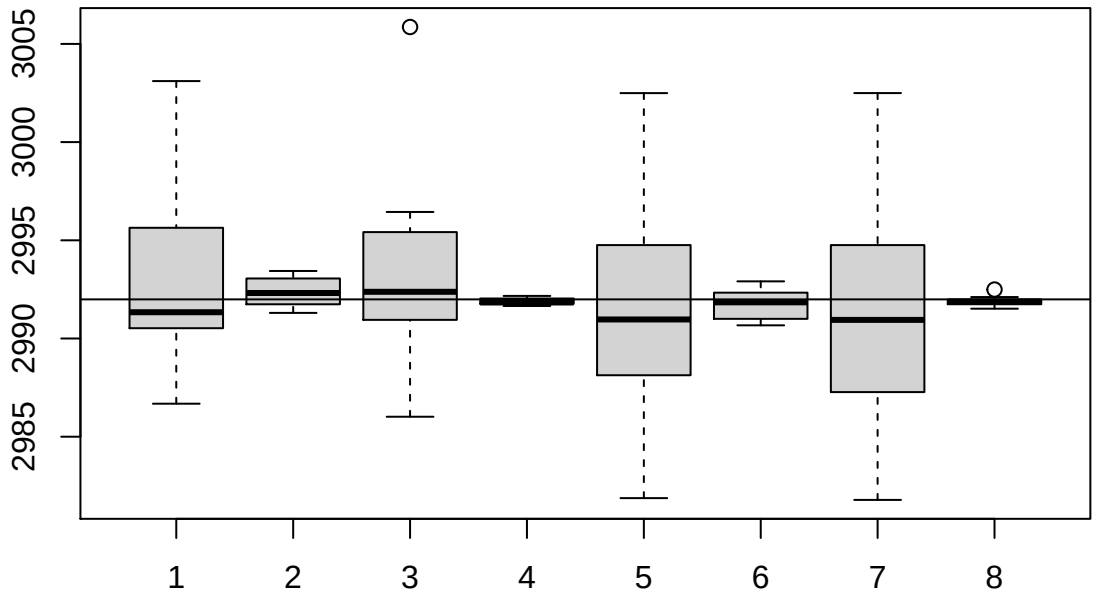
Estimating expectations with control variates

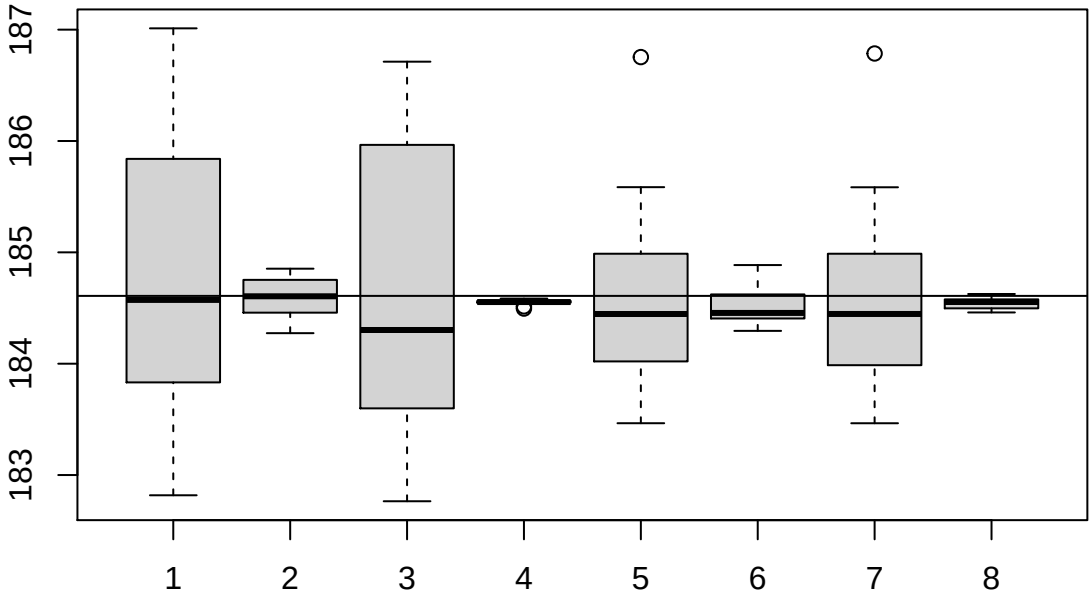
Now we'll estimate the marginal posterior means for our three parameters. In practice we only want to use one estimate, and you can see that the control variate approaches are much better.

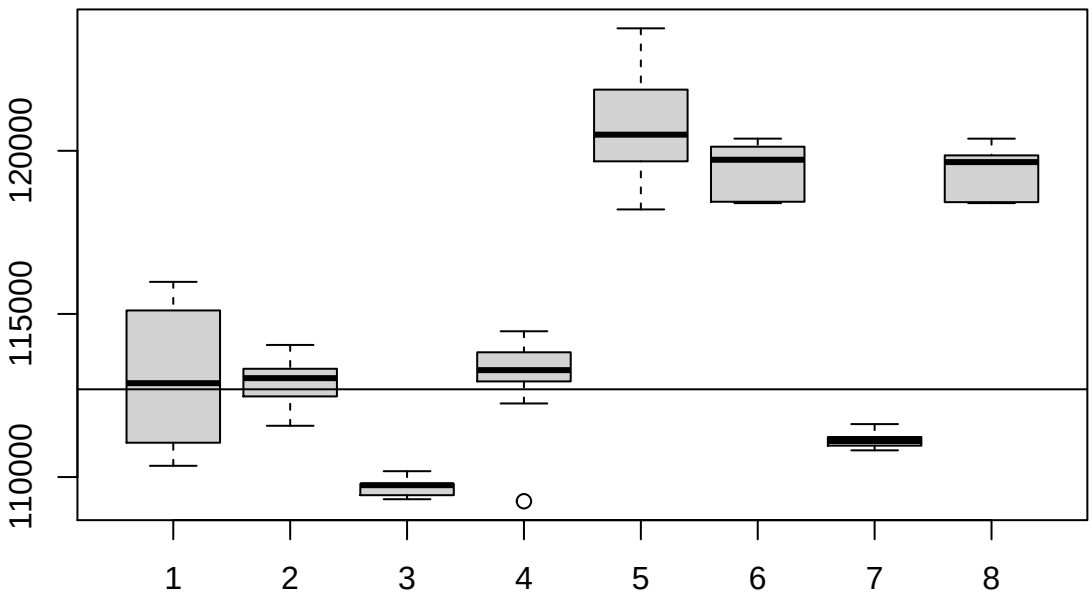
Notice that our gradients correspond to the samples on the log scale, so we aren't exponentiating `samples[,3]`.

```
Vanilla_MALA <- ZV1_MALA <- CF_MALA <- SECF_MALA <- matrix(NaN,nrow=n_reps,ncol=3)
Vanilla_ULA <- ZV1_ULA <- CF_ULA <- SECF_ULA <- matrix(NaN,nrow=n_reps,ncol=3)
for (i in 1:n_reps){
  samples <- as.matrix(chains_MALA[[i]]$samples)
  gradients <- chains_MALA[[i]]$der_loglike + chains_MALA[[i]]$der_logprior
  integrand <- samples
  integrand[,3] <- exp(integrand[,3])

  # In order: vanilla estimate, zero-variance control variates
  # with a first order polynomila, control functionals and
  # semi-exact control functionals with a first order polynomial
  Vanilla_MALA[i,] <- colMeans(integrand)
  ZV1_MALA[i,] <- zvcv(integrand, samples, gradients,
                       options = list(polyorder = 1, regul_reg = FALSE))$expectation
  CF_MALA[i,] <- CF_crossval(integrand, samples, gradients, kernel_function = "RQ",
                             sigma_list = list(0.001,0.01,0.1,1,10), folds = 2)$expectation
  SECF_MALA[i,] <- SECF_crossval(integrand, samples, gradients, polyorder = 1, kernel_function = "RQ",
                                 sigma_list = list(0.001,0.01,0.1,1,10), folds = 2)$expectation

  # In order: vanilla estimate, zero-variance control variates
```

```
  # with a first order polynomila, control functionals and
  # semi-exact control functionals with a first order polynomial
  samples <- as.matrix(chains_ULA[[i]]$samples)
  gradients <- chains_ULA[[i]]$der_loglike + chains_ULA[[i]]$der_logprior
  integrand <- samples
  integrand[,3] <- exp(integrand[,3])

  Vanilla_ULA[i,] <- colMeans(integrand)
  ZV1_ULA[i,] <- zvcv(integrand, samples, gradients,
                      options = list(polyorder = 1, regul_reg = FALSE))$expectation
  CF_ULA[i,] <- CF_crossval(integrand, samples, gradients, kernel_function = "RQ",
                            sigma_list = list(0.001,0.01,0.1,1,10), folds = 2)$expectation
  SECF_ULA[i,] <- SECF_crossval(integrand, samples, gradients, polyorder = 1, kernel_function = "RQ",
                                sigma_list = list(0.001,0.01,0.1,1,10), folds = 2)$expectation
}

# Boxplots of the estimates
gold_standard <- c(2991.992, 184.609, 112689.957) # from a 1 million iteration run.
for (j in 1:3){
  boxplot(Vanilla_MALA[,j],ZV1_MALA[,j],CF_MALA[,j],SECF_MALA[,j],
          Vanilla_ULA[,j],ZV1_ULA[,j],CF_ULA[,j],SECF_ULA[,j])
  abline(h=gold_standard[j])
}
```




187
186
185
184
183
1
2
3
4
5
6
7
8


110000
115000
120000
1
2
3
4
5
6
7
8