2. Domača naloga

Matevž Gombač in Alan Krumenaker

V temu poročilu sva z Metropolis - Hastingsovim algoritmom ocenjevala povprečno višino moških študentov glede na podan vzorec velikosti 30. Pri tem modelu smo poznali apriorno porazdelitev - normalno - s parametri apriorni_mi in apriorna_varianca, za znano varianco pa smo uzeli kar znano_varianco.

```
apriorni_mi <- 1.78
apriorna_varianca <- 0.2
znana_varianca <- 0.1
podatki <- c(1.91, 1.94, 1.68, 1.75, 1.81, 1.83, 1.91, 1.95, 1.77, 1.98,
       1.81, 1.75, 1.89, 1.89, 1.83, 1.89, 1.99, 1.65, 1.82, 1.65,
       1.73, 1.73, 1.88, 1.81, 1.84, 1.83, 1.84, 1.72, 1.91, 1.63)
n <- length(podatki)
vzorcno_povprecje <- mean(podatki)
```

Za uporabo algoritma potrebujemo najprej verjetje, apriorno in predlagalno porazdelitev.

```
verjetje <- function(mi){
  prod(dnorm(podatki, mean = mi, sd = znana_varianca))
}



apriorna <- function(x){
  dnorm(x, mean = apriorni_mi, sd = apriorna_varianca)
}

predlagalna <- function(x, y)
  dnorm(x, mean = y, sd = znana_varianca)
```

Sedaj smo pripravljeni za implementacijo Metropolis_Hastingsovega algoritma:

```
mh <- function(iteracije, zacetni_mi, varianca=znana_varianca){
  t = numeric(iteracije)
  sprejeti <- 0 #šteje število sprejetih parametrov
  trenutni_mi <- zacetni_mi
  for(i in 1:iteracije){
    #simuliramo kandidata
    predlagan_mi <- rnorm(1, mean = trenutni_mi, sd = varianca)
```

```
    #print(predlagan_mi)
        predlagalno_razmerje <- (verjetje(predlagan_mi)*apriorna(predlagan_mi)*predlagal
        #print(predlagalno_razmerje)
        alpha <- min(predlagalno_razmerje, 1)
        u <- runif(1)
      if (u <= alpha) {
        trenutni_mi <- predlagan_mi
        sprejeti <- 1 + sprejeti
        t[i] <- trenutni_mi
      }else{
        trenutni_mi <- trenutni_mi
        t[i] <- trenutni_mi
      }
  }
  #return(tail(t, 100))
  #print(sprejeti)
  return(t)

}
```

Seveda bomo tudi preizkusili algoritem na našem primeru. Če ne bo drugače omenjeno bomo gledali algoritem na 40000 korakih in potem na zadnji vrednosti pogledali, kakšno porazdelitev dobimo. V našem primeru pa slučajno poznamo tudi pravo - analitično izpeljano - aposteriorno porazdelitev. To bomo uporabili, da primerjamo naše algoritemske rezultate s pravilnimi in opazimo možna odstopanja. Poglejmo si torej, kakšno zaporedje dobimo:

```
#analitično izpeljane enačbe za pravo aposteriorno porazdelitev
aposteriorni_mi <- ((1/(apriorna_varianca)^2)/(1/(apriorna_varianca)^2 + (n/(znana_varia
aposteriorna_varianca <- (znana_varianca^2 * apriorna_varianca^2 * n) / ((znana_varianca
prava_aposteriorna <- function(mi){
  dnorm(mi, mean = aposteriorni_mi, sd =sqrt(aposteriorna_varianca))
}
# pravi_apost_mi = 1.820331
prava_apost_sd = 0.01818182
```

```
smiselna_zacetna <- 1.8 #180cm recimo da so veliki

risi_zaporedje <- function(st_iteracij=40000, zac_vr = smiselna_zacetna, burnin = FALSE)
x11()
  it_risanja <- 1:st_iteracij
  theta_risanja <- mh(st_iteracij, zac_vr)
  #theta <- mhlog(length(it_risanja), 1)
  if (burnin) {
    it_risanja <- tail(it_risanja, st_iteracij - 10000)
    theta_risanja <- tail(theta_risanja, st_iteracij - 10000)
```

```
  }
  G = plot(it_risanja, theta_risanja, ylab="Theta", xlab="Iteracije", type="l")
  abline(h=aposteriorni_mi-1.96*prava_apost_sd, col="green")
  abline(h=aposteriorni_mi+1.96*prava_apost_sd, col="green")
  abline(h=aposteriorni_mi-3*prava_apost_sd, col="red")
  abline(h=aposteriorni_mi+3*prava_apost_sd, col="red")
  return(G)

}
```

```
risi_zaporedje()
```

```
## NULL
```

Z zeleno smo označili 95% referencni interval prave aposteriorne porazdelitve in z rdečo 99.7% referencni interval. Opazimo, da naše zaporedje konvergira kar v redu.

Poglejmo sedaj zaporedje samo za prvih nekaj členov, kjer nekaj pomeni med 500 in 5000.

```
risi_zaporedje(2000)
```

```
## NULL
```

Poglejmo še dogajanje, če ignoriramo prvih nekaj členov.

```
risi_zaporedje(burnin = TRUE)
```

```
## NULL
```

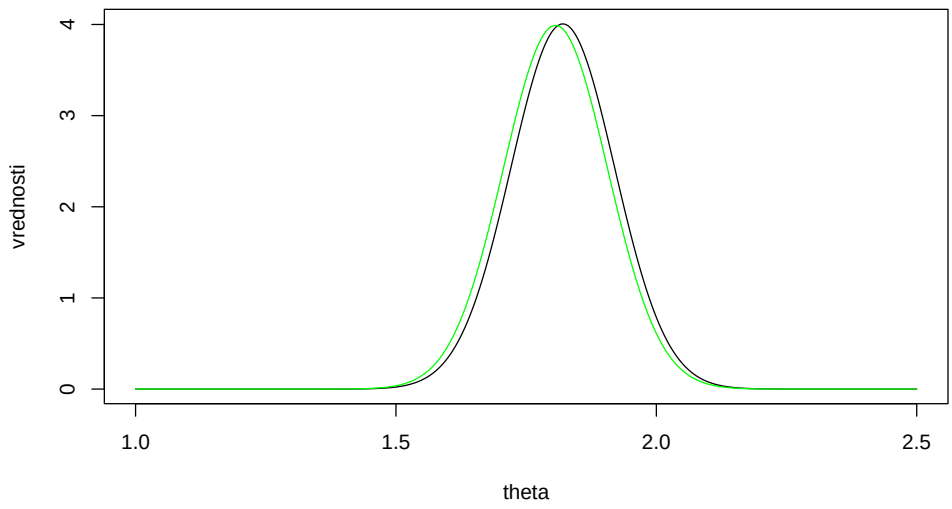
Zaenkrat ne opazimo preveliki razlike, vendar se moramo zavedati, da naše teste poganjemo na zelo dobri začetni vrednosti. Kasneje - ob slabših začetnih vrednostih in rahlo prilagojenem algoritmu - bomo videli razlike.

Sedaj bomo pa še dobljeno aposteriorno porazdelitev primerjali s tisto pravo analitično. Za aposteriorno porazdelitev bomo vzeli normalno, kjer je povprečje zadnja dobljena vrednost z algoritmom, varianca pa znana od prej. Normalno porazdelitev seveda dobimo iz konjugirane.

```
iteracije <- 40000
vrednosti <- mh(iteracije, smiselna_zacetna)
mh_parameter <- vrednosti[iteracije]

theta <- seq(1, 2.5, 0.0001)
vrednosti <- prava_aposteriorna(theta)
f <- dnorm(theta, mean = mh_parameter, sd = znana_varianca)# f je risanje mh aposterior

x11()
plot(theta, vrednosti, type = "l")
lines(theta, f, col="green")
```



Naš končni parameter $\hat{\theta}$ bo zadnji izračunani parameter v algoritmu.

```
theta_hat = tail(mh(40000, 1.8),1)
pravi_theta = aposteriorni_mi
```

Za dobro začetno vrednost je tudi naša ocena precej dobra.

Če si pogledamo še intervale zaupanja, ugotovimo, da so sicer kar veliki, ampak še vedno dobro sovpadajo s pravimi:

```
#ocena in intervali
(iz <- qnorm(c(0.025, 0.975), mean = theta_hat , sd = znana_varianca))
```

```
## [1] 1.626418 2.018411
```

```
(iz_pravi = qnorm(c(0.025, 0.975), mean = pravi_theta, sd = sqrt(aposteriorna_varianca))
```

```
## [1] 1.625146 2.015515
```

```
library(HDInterval)
aposteriorna.sample <- rnorm(1000000, mean = mh_parameter, sd = znana_varianca)
(iz.hdi <- hdi(aposteriorna.sample, credMass = 0.95))
```

```
##    lower    upper
## 1.607730 1.999839
## attr(,"credMass")
## [1] 0.95
```

```
aposteriorna.sample_pravi <- rnorm(1000000, mean = pravi_theta, sd = sqrt(aposteriorna_v
(iz.hdi_pravi <- hdi(aposteriorna.sample_pravi, credMass = 0.95))
```

```
##    lower    upper
```

```
## 1.625124 2.015494
## attr(,"credMass")
## [1] 0.95
```

Do sedaj smo algoritem uporabljali na neki zelo smiselni začetni vrednosti 1.8. Poglejmo, kaj se zgodi, če ga uporabimo na neki nesmiselni vrednosti, recimo 3 ali 1:

```
#mh(40000, 3)
#mh(40000, 1)
```

Če poženemo Metropolis - Hastingsov algoritem kot je implementiran zgoraj, nam kaj kmalu odpove. Namreč, za zgornjih začetnih vrednostih nam že javlja napake: Error in if (u <= alpha) { : missing value where TRUE/FALSE needed. Napake se zgodijo ob preverjanju, kakšna je dobljena verjetnost sprejetja novega predlaganega povprečja; torej ali je manjša ali večja od parametra, ki ga dobimo iz vzorčenja iz enakomerne porazdelitve na intervalu $[0, 1]$. Glede na napake sklepamo, da pride do težave z numeriko, saj so izpeljana razmerja lahko poljubno velika in potem R tega ne interpretira pravilno.

Iz tega razloga implementiramo algoritem s pomočjo logaritma.Poglejmo, če deluje in pojasnimo zakaj.

```
mhlog <- function(iteracije, zacetni_mi, varianca=znana_varianca){
  t = numeric(iteracije)
  sprejeti <- 0 #šteje število sprejetih parametrov
  trenutni_mi <- zacetni_mi
  for(i in 1:iteracije){
    predlagan_mi <- rnorm(1, mean = trenutni_mi, sd = varianca)
    #print(predlagan_mi)
        predlagalno_razmerje <- log(verjetje(predlagan_mi)) + log(apriorna(predlagan_mi)
      if (! (is.finite(predlagalno_razmerje))){
        alpha <- 0}
      else{
        alpha <- min(0, predlagalno_razmerje)}
      #print(alpha)

      u <- runif(1)
      if (log(u) <= alpha) {
        trenutni_mi <- predlagan_mi
        sprejeti <- 1 + sprejeti
        t[i] <- trenutni_mi
      }else{
        trenutni <- trenutni_mi
        t[i] <- trenutni_mi
      }

  }
  return(t)
```

```
  #return(tail(t, 100))
  #return(t)
}
```

Zakaj so se pojavile težave? Kot smo razmišljali prej, se R-ju za zelo nesmiselno začetno vrednost poruši numerika. Bolj natančno: pri izračunu nove predlagane vrednosti izračunamo v ulomku po vrsti več vrednosti. Ob množenju velikih števil pridemo do prekoračitve, ta velika števila pa potem med seboj še množimo in delimo. Na primer, če delimo med seboj dve neskončnosti, ne vemo, kaj se bo zgodilo in R vrne Nan. Ko potem v pogojnem stavku primerjamo simulirano vrednost iz enakomerne porazdelitve na $[0, 1]$ z NaN, se seveda pojavi napaka.

Zakaj torej logaritem popravi algoritem? Kljub temu, da je logaritem monotono naraščajoča funkcija, nam zelo "zmanjša" vrednosti, torej to naraščanje je zelo počasno. To nam dovoli, da uporabljamo veliko bolj nesmiselne številke, saj so vrednosti na logaritemski lestvici veliko bližje. Hkrati pa se zaradi monotonosti neenakosti pri preverjanju sprejetja ne obrnejo.

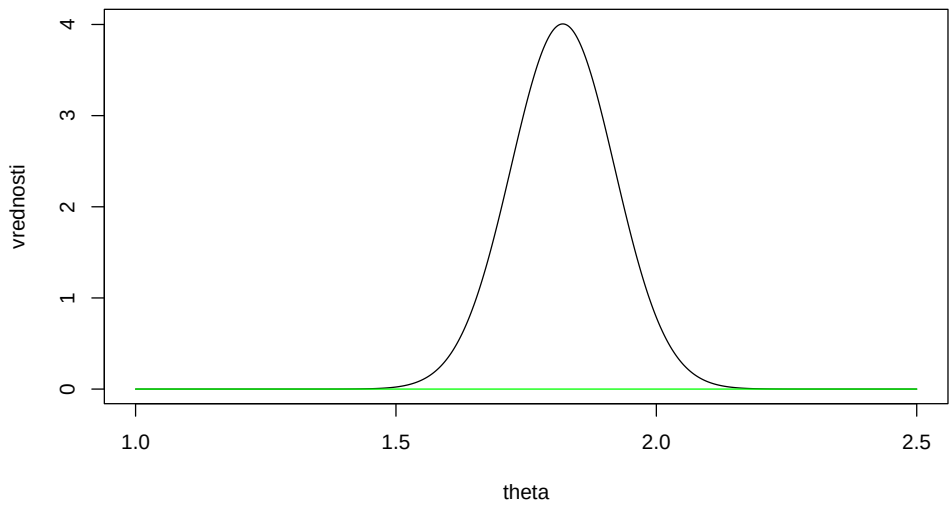
Kljub implementaciji z logaritmom pa se še vedno lahko pojavijo težave. Te odpravimo z dodatnim pogojnim stavkom, ki filtrira neveljavne vrednosti (Inf, Nan...). V tem primeru vedno vzamemo za verjetnost sprejetja 0 in čakamo, dokler ne dobimo neke smiselne vrednosti. To povzroči slabšo hitrost konvergence. Podobno bi lahko naredili v navadnem Metropolis - Hastingsu, vendar bi v primeru implementacije brez logaritma bila ta konvergenca še toliko slabša.

Za vizualizacijo zgornjih opažanj poženimo algoritem pri manj smiselnih zacetnih verdnostih (glede na naše podatke):

```
nesmiselna_zacetna <- 5 #nekako dvomimo, da je nekdo visok 5m
iteracije <- 40000
vrednostilog <- mhlog(iteracije, nesmiselna_zacetna)
mh_parameterlog <- vrednostilog[iteracije]

theta <- seq(1, 2.5, 0.0001)
vrednosti <- prava_aposteriorna(theta)
g <- dnorm(theta, mean = mh_parameterlog, sd = znana_varianca)# g je risanje mhlog apos

x11()
plot(theta, vrednosti, type = "l")
#plot(theta,g,type = "l", col= "green")
lines(theta, g, col="green")
```



Vidimo, da se tudi za manj smiselno začetno vrednost logaritmiran algoritem dobro izkaže. Poglejmo pa si, kaj se dogaja s konvergenco; narišimo najprej celotno zaporedje, potem pa še z ustreznim *burn-in*.

```
risi_zaporedjelog <- function(st_iteracij=40000, zac_vr = nesmiselna_zacetna, burnin = F
x11()
  it_risanja <- 1:st_iteracij
  theta_risanja <- mhlog(st_iteracij, zac_vr,varianca)
  #theta <- mhlog(length(it_risanja), 1)
  if (burnin) {
    it_risanja <- tail(it_risanja, st_iteracij - 10000)
    theta_risanja <- tail(theta_risanja, st_iteracij - 10000)
  }
  plot(it_risanja, theta_risanja, ylab="Theta", xlab="Iteracije", type="l")
  abline(h=aposteriorni_mi-1.96*prava_apost_sd, col="green")
  abline(h=aposteriorni_mi+1.96*prava_apost_sd, col="green")
  abline(h=aposteriorni_mi-3*prava_apost_sd, col="red")
  abline(h=aposteriorni_mi+3*prava_apost_sd, col="red")
}
```

```
risi_zaporedjelog()
```

Kot pričakovano je konvergenca slabša, vendar algoritem deluje! Poglejmo, kaj se zgodi, če algoritmu ne pustimo velikega števila korakov:

```
risi_zaporedjelog(2000)
```

Občasno je že malo korakov zadosti, vendar se to ne zgodi vedno. Prvih nekaj je lahko vrednosti še vedno zelo daleč od željenega rezultata, zato lahko smiselno uporabimo *burn-in*.

```
risi_zaporedjelog(burnin = TRUE)
```

Če vrednosti dejansko konvergirajo, lahko z *burn-in* vidimo, da se od nekje naprej zelo dobro prilegajo željeni.

Sedaj pa vzemimo neko smiselno začetno vrednost in poženimo algoritem pri različnih variancah. Ker smo merili višino študentov, bomo pretiravali v obe smeri, vendar še vedno nekako smiselno. Najprej, če vzamemo zelo majhno varianco, npr:

```
majhna_var = 0.001
risi_zaporedjelog(2000, smiselna_zacetna,burnin = FALSE, majhna_var)
```

```
risi_zaporedjelog(40000,smiselna_zacetna, burnin = FALSE, majhna_var)
```

Poskusimo sedaj še z zelo veliko:

```
velika_var = 1
risi_zaporedjelog(2000, smiselna_zacetna,burnin = FALSE, velika_var)
```

```
risi_zaporedjelog(40000, smiselna_zacetna,burnin = FALSE, velika_var)
```

Opažanje: pri majhni varianci se členi zaporedja razlikujejo zelo malo, kar je tudi smiselno glede na implementacijo algoritma - namreč, naslednji predlagan člen je izbran "blizu" (=normalno okrog) začetne vrednosti. Če je začetna vrednost smiselna in varianca majhna, upravičeno pričakujemo, da bo naslednji predlagalni člen blizu smiselni vrednosti. Potem pa induktivno tudi naslednji in naslednji... Simetrično pri veliki varianci sploh ni zares važno kakšna je začetna vrednost - naslednji člen je lahko z relativno veliko verjetnostjo izbran daleč od smiselne vrednosti.

V najinem mnenju je vseeno bolje uporabljati majhno varianco, vendar je v tem primeru treba imeti dobro apriorno vedenje o začetni verdnosti. Če je ta pogoj izpolnjen, potem smo lahko bolj prepričani, da algoritem še vedno deluje.