Computational Statistics - Suggested Solution for Exam

Frank Miller, IDA, Linköping University

2024-05-13

Assignment 1

Question 1.1

```
g <- function(x){
  x1 <- x[1]
  x2 <- x[2]
  -x1^2-x1^2*x2^2-2*x1*x2+2*x1+2
}

gradient <- function(x){
  x1  <- x[1]
  x2  <- x[2]
  dx1 <- -2*x1-2*x1*x2^2-2*x2+2
  dx2 <- -2*x1^2*x2-2*x1
  c(dx1, dx2)
}

hessian <- function(x){
  x1   <- x[1]
  x2   <- x[2]
  hx11 <- -2-2*x2^2
  hx12 <- -4*x1*x2-2
  hx22 <- -2*x1^2
  matrix(c(hx11, hx12, hx12, hx22), ncol=2)
}

# the Newton algorithm
newton <- function(x0)
{
  xt  <- x0
  xt1 <- x0 + 2
  while(sum((xt-xt1)^2)>0.0001)
  {
    xt1 <- xt
    # the core equation for Newton
    xt  <- xt1 - solve(hessian(xt1)) %*% gradient(xt1)
    # printing current iteration - in order to see what happens (which is
asked in Part b)
    print(xt)
  }
  # determine if local max, min, or saddle point
```

```
  eval <- sort(eigen(hessian(xt))$values)
  message(paste("Eigenvalues of the Hessian: ", eval[1], eval[2]))
  if (eval[2]<0) message("Both eigenvalues of the Hessian are negative. This
is a (local) maximum.")
  if (eval[1]<0 & eval[2]>0) message("The Hessian has a negative and a
positive eigenvalue. This is a saddle point.")
  if (eval[1]>0) message("Both eigenvalues of the Hessian are positive. This
is a (local) minimum.")
  # output of newton-function:
  xt
}
```

Question 1.2

```
sol1 <- newton(c(2, 0))
```

```
##            [,1]
## [1,]  1.3333333
## [2,] -0.3333333
##            [,1]
## [1,]  1.2410901
## [2,] -0.7442348
##            [,1]
## [1,]  1.0237697
## [2,] -0.9252917
##            [,1]
## [1,]  1.0044994
## [2,] -0.9932296
##            [,1]
## [1,]  1.0000256
## [2,] -0.9999341

## Eigenvalues of the Hessian:  -5.23576170879168 -0.764077329382244

## Both eigenvalues of the Hessian are negative. This is a (local) maximum.
```

```
sol2 <- newton(c(-1, -2))
```

```
##       [,1]
## [1,] -0.65
## [2,] -0.75
##            [,1]
## [1,] -0.4212980
## [2,]  0.4693814
##            [,1]
## [1,] -0.2944317
## [2,]  1.9415362
##            [,1]
## [1,] -0.1463403
## [2,]  3.6411701
##            [,1]
## [1,] -0.1708772
```

```
## [2,]  6.7581129
##            [,1]
## [1,] -0.2562939
## [2,]  2.0211055
##            [,1]
## [1,] -0.1872322
## [2,]  3.9396145
##            [,1]
## [1,] -0.1059095
## [2,]  6.4434362
##            [,1]
## [1,] -0.1008975
## [2,]  9.6050448
##             [,1]
## [1,]  0.05997255
## [2,] 24.73735499
##             [,1]
## [1,]  0.03238945
## [2,] 13.74953864
##            [,1]
## [1,] 0.01204251
## [2,] 5.79567256
##             [,1]
## [1,] 0.001278017
## [2,] 1.548633061
##              [,1]
## [1,] 3.408517e-06
## [2,] 1.002152e+00
##              [,1]
## [1,] 1.166791e-11
## [2,] 1.000000e+00

## Eigenvalues of the Hessian:  -4.82842717488454 0.828427116182473

## The Hessian has a negative and a positive eigenvalue. This is a saddle
point.

sol3 <- newton(c(0, 1))

##      [,1]
## [1,]    0
## [2,]    1

## Eigenvalues of the Hessian:  -4.82842712474619 0.82842712474619
## The Hessian has a negative and a positive eigenvalue. This is a saddle
point.
```

Only with starting value (2, 0), the iterations arrive at (1, -1) which is a local maximum based on the eigenvalues of the Hessian matrix at (1, -1). With (-1, -2) as starting value, the iterations arrives at (0, 1); this is a saddle point since one eigenvalue is negative and one is

positive. With the starting point at (0, 1), the algorithm terminates directly since the gradient is (0, 0), there.

Output of solution and function value at solution:

```
sol1

##            [,1]
## [1,]  1.0000256
## [2,] -0.9999341

sol2

##              [,1]
## [1,] 1.166791e-11
## [2,] 1.000000e+00

g(sol1)

## [1] 4

g(sol2)

## [1] 2
```

The function value at (1, -1) is 4, at (0, 1), it is 2.

Question 1.3

If the solution found is a saddle point or a local minimum, one can generate a new starting point and re-run the Newton algorithm. If it is a saddle point, one might do a larger step into the direction of ascent based on the eigenvector which belongs to the negative eigenvalue of the Hessian. If it is a minimum, a larger step into a random direction might be done.

Question 1.4

The steepest ascent algorithm together with a line search step (alpha-halving) in each iteration ensures uphill steps while in the Newton, downhill steps are possible. The region where the algorithm converges to the true maximum is usually larger for the steepest ascent than for the Newton (due to ensuring uphill steps and using smaller steps). This means that Newton is more sensitive to a good choice of the staring value. We have indeed seen here that only one out of three starting values converged to the maximum for the Newton algorithm. We need not to calculate and program the Hessian for the steepest ascent which can be an advantage over Newton. Steepest ascent is slower than Newton; depending on the choice of the step size, it can be much slower.

Assignment 2

Question 2.1

```
f <- function(x){
  ifelse(x<0, 0, (5/6)*exp(-x)*(1+cos(2*x)))
}
```

Since cos(…) is between -1 and 1, we have $0 \leq \big(1 + \cos(2 * x)\big) \leq 2$. Consequently, the following function e is always larger than f:

```
e <- function(x){
  ifelse(x<0, 0, exp(-x)*10/6)
}
```

Note that $\exp(-x), x \geq 0$, is a probability density from the exponential distribution with rate-parameter $\lambda = 1$. Therefore, the scaling factor is $10/6 = 5/3$.

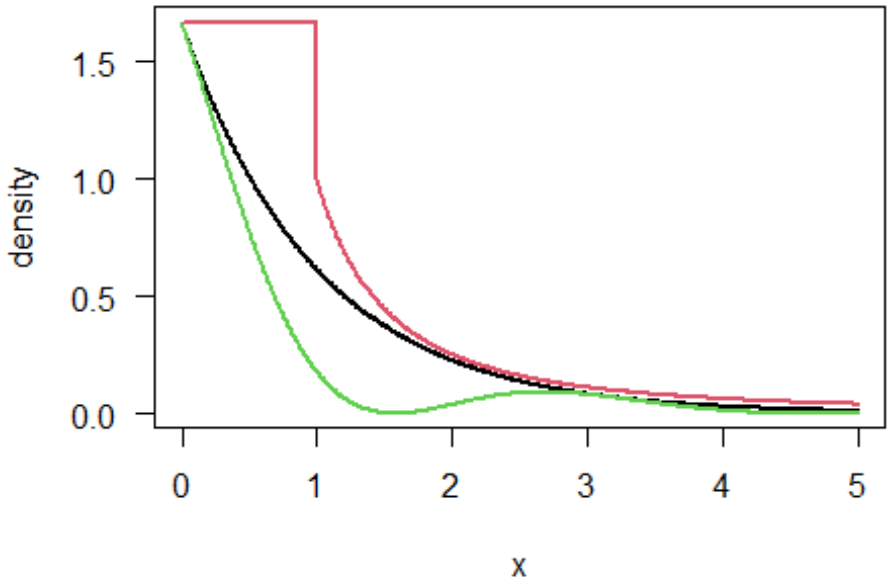
The following function e_{bad} is also $> f$ everywhere, but it has the following disadvantages compared to e:

- $e_{bad} > e$ which leads to more waist (more rejections)
- it is not as straightforward to figure out the scaling factor, but can be done: $\int_0^\infty e_{bad}\,(x)dx = 10/6 + \int_1^\infty e_{bad}\,(x)dx = 16/6 = 8/3$
- To generate random variables according to e_{bad} is more complex and a separate inverse-CDF-method needs to be done including integrating and inverting the density e_{bad}

```
ebad <- function(x){
  ifelse(x<0, 0, ifelse(x<1, 10/6, 1/x^2))
}
```

Question 2.2

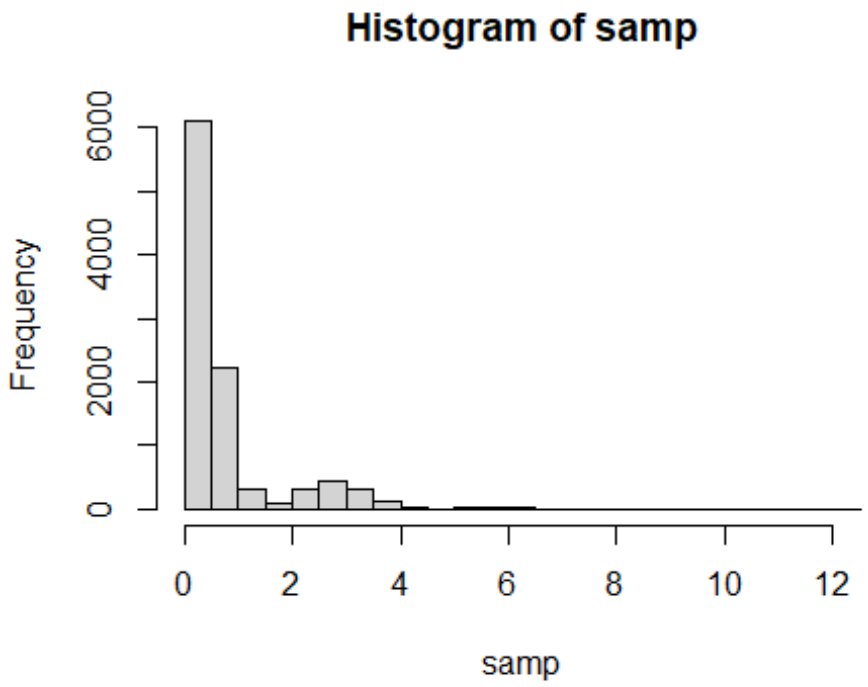
```
xv <- seq(0, 5, by=0.01)
plot(xv, e(xv), type="l", xlab="x", ylab="density", lwd=2, las=1)
lines(xv, f(xv), col=3, lwd=2)
lines(xv, ebad(xv), col=2, lwd=2)
```



Question 2.3

```
gen_sample <- function(n){
  samp <- c()
  i <- 0
  while (i<n){                    # if wrong number of samples -0.5p
    x <- rexp(1, rate=1)          # if density different from scaled envelope
is used: -1p
    u <- runif(1, min=0, max=1) # if wrong distribution -1p
    if (u <= f(x)/e(x)){          # if not unscaled envelope used -0.5p; if not
right structure u<=f/e, additional -0.5p
      samp <- c(samp, x)
      i    <- i+1
    }
  }
  samp
}

n <- 10000
samp <- gen_sample(n)
hist(samp, breaks=20)
```



Question 2.4

Estimate $P(X > 1)$:

```
phat <- mean((samp>1))
phat
```

```
## [1] 0.167
```

A measure of uncertainty for this estimate: Taking a single of the 10000 samples and defining 1 if it is ≥ 1 and 0 otherwise, it has a $Bin(1, p)$-distribution; therefore, it has variance $p(1-p)$. The average of 10000 has variance $p(1-p)/n$. The standard error of the estimate is therefore $\sqrt{p(1-p)/n}$ and an estimate for it can be obtained if we put in the estimate for p, which we have obtained before:

```
sephat <- sqrt(phat*(1-phat)/n)
sephat
```

```
## [1] 0.003729759
```

A 95%-confidence interval (not required here), is

```
phat + qnorm(c(0.025, 0.975))*sephat
```

```
## [1] 0.1596898 0.1743102
```

Estimate EX:

```r
muhat <- mean(samp)
muhat
```

```
## [1] 0.7319238
```

A measure of uncertainty can be obtained using the Jacknife method as follows (an alternative is bootstrap)

```r
resamp <- c()
for (i in 1:n){
  resamp <- c(resamp, mean(samp[-i]))
}
semuhat <- sqrt((n-1)^2/n) * sd(resamp)
semuhat
```

```
## [1] 0.009977331
```

A 95%-confidence interval (not required here), is

```r
muhat + qnorm(c(0.025, 0.975))*semuhat
```

```
## [1] 0.7123686 0.7514790
```

Another method: bootstrap

```r
B   <- 1000
bms <- c()
for (b in 1:B){
  rsamp <- sample(samp, size=n, replace=T)
  bms   <- c(bms, mean(rsamp))
}
sd(bms)
```

```
## [1] 0.009881074
```

Another ad-hoc method, which is very time-expensive, but has a relatively simple idea: create 100 samples a 10000 and derive estimator for $E(X)$ as before - the variability of these 100 $E(X)$-estimates tell us about the uncertainty of a single estimate.

```r
A    <- 100
ams <- c()
for (a in 1:A){
  rsamp <- gen_sample(n)
  ams   <- c(ams, mean(rsamp))
}
sd(ams)
```

```
## [1] 0.009591346
```