PC6a - Optimisation

Question 1

```
logLik<-function(theta,x) {
  logL <- -n*theta+sum(x)*log(theta)
}

gr_logLik<-function(theta,x) {
  gr <- -n+sum(x)/theta
}

x <- scan("countdata.dat")
n <- length(x)

cat('Estimated parameters\n')
```

```
## Estimated parameters
```

```
result1 <- optim(4,logLik,gr=NULL,x,method="Brent", lower=0,upper=10,control=list(fnscal
print(result1$par)
```

```
## [1] 3.06
```

```
result2 <- optim(4,logLik,x=x,gr_logLik,
                 method="BFGS",control=list(fnscale=-1,maxit=300,trace=TRUE,REPORT=5))
```

```
## initial  value -48.412149

## Warning in log(theta): NaNs produced

## Warning in log(theta): NaNs produced

## iter   5 value -72.462196
## final  value -72.469929
## converged
```

```
names(result2)  # all properties of the object result2
```

```
## [1] "par"         "value"       "counts"      "convergence" "message"
```

```
print(result2$par)
```

```
## [1] 3.060001
```

Question 2 [Exercise 4 of Section 12.8.4 of Jones et al.]

```
# Solution to Rosenbrock exercise.

gsection <- function(ftn, x.l, x.r, x.m, tol = 1e-9) {
  # applies the golden-section algorithm to maximise ftn
  # we assume that ftn is a function of a single variable
  # and that x.l < x.m < x.r and ftn(x.l), ftn(x.r) <= ftn(x.m)
  #
  # the algorithm iteratively refines x.l, x.r, and x.m and terminates
  # when x.r - x.l <= tol, then returns x.m

  # golden ratio plus one
  gr1 <- 1 + (1 + sqrt(5))/2

  # successively refine x.l, x.r, and x.m
  f.l <- ftn(x.l)
  f.r <- ftn(x.r)
  f.m <- ftn(x.m)
  while ((x.r - x.l) > tol) {
    if ((x.r - x.m) > (x.m - x.l)) {
      y <- x.m + (x.r - x.m)/gr1
      f.y <- ftn(y)
      if (f.y >= f.m) {
        x.l <- x.m
        f.l <- f.m
        x.m <- y
        f.m <- f.y
      } else {
        x.r <- y
        f.r <- f.y
      }
    } else {
      y <- x.m - (x.m - x.l)/gr1
      f.y <- ftn(y)
      if (f.y >= f.m) {
        x.r <- x.m
```

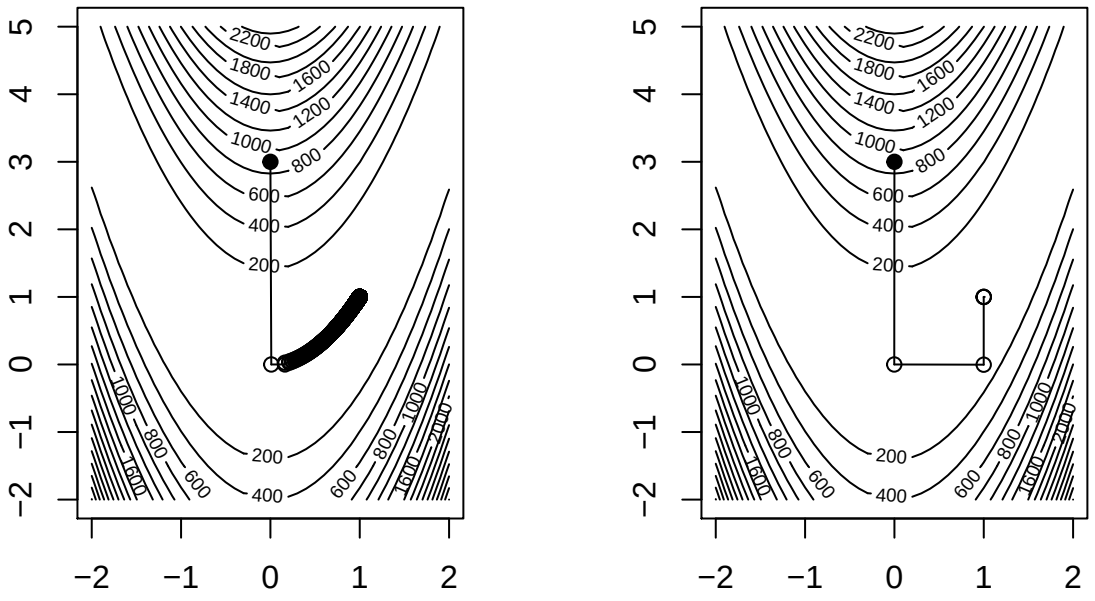
```
        f.r <- f.m
        x.m <- y
        f.m <- f.y
      } else {
        x.l <- y
        f.l <- f.y
      }
    }
  }
  return(x.m)
}

line.search <- function(f, x, y, tol = 1e-9, a.max = 2^5) {
    # f is a real function that takes a vector of length d
    # x and y are vectors of length d
    # line.search uses gsection to find a >= 0 such that
    #   g(a) = f(x + a*y) has a local maximum at a,
    #   within a tolerance of tol
    # if no local max is found then we use 0 or a.max for a
    # the value returned is x + a*y

    if (sum(abs(y)) == 0) return(x) # g(a) constant

    g <- function(a) return(f(x + a*y))

    # find a triple a.l < a.m < a.r such that
    # g(a.l) <= g(a.m) and g(a.m) >= g(a.r)
    # a.l
    a.l <- 0
    g.l <- g(a.l)
    # a.m
    a.m <- 1
    g.m <- g(a.m)
    while ((g.m < g.l) & (a.m > tol)) {
        a.m <- a.m/2
        g.m <- g(a.m)
    }
    # if a suitable a.m was not found then use 0 for a
    if ((a.m <= tol) & (g.m < g.l)) return(x)
    # a.r
    a.r <- 2*a.m
    g.r <- g(a.r)
    while ((g.m < g.r) & (a.r < a.max)) {
        a.m <- a.r
```

```
        g.m <- g.r
        a.r <- 2*a.m
        g.r <- g(a.r)
    }
    # if a suitable a.r was not found then use a.max for a
    if ((a.r >= a.max) & (g.m < g.r)) return(x + a.max*y)

    # apply golden-section algorithm to g to find a
    a <- gsection(g, a.l, a.r, a.m)
    return(x + a*y)
}

g <- function(x) -(1 - x[1])^2 - 100*(x[2] - x[1]^2)^2
gradg <- function(x) c(2*(1 - x[1]) + 400*(x[2] - x[1]^2)*x[1], -200*(x[2] - x[1]^2))

ascent <- function(f, grad.f, x0, tol = 1e-9, n.max = 100) {
    # steepest ascent algorithm
    # find a local max of f starting at x0
    # function grad.f is the gradient of f

    x <- x0
    points(x[1], x[2], pch=19)
    x.old <- x
    x <- line.search(f, x, grad.f(x))
    n <- 1
    while ((f(x) - f(x.old) > tol) & (n < n.max)) {
        lines(c(x.old[1], x[1]), c(x.old[2], x[2]))
        points(x[1], x[2])
        x.old <- x
        x <- line.search(f, x, grad.f(x))
        n <- n + 1
    }
    return(x)
}

Rosenbrock <- function(x) {
  g <- (1 - x[1])^2 + 100*(x[2] - x[1]^2)^2
  g1 <- -2*(1 - x[1]) - 400*(x[2] - x[1]^2)*x[1]
  g2 <- 200*(x[2] - x[1]^2)
  g11 <- 2 - 400*x[2] + 1200*x[1]^2
  g12 <- -400*x[1]
  g22 <- 200
  return(list(g, c(g1, g2), matrix(c(g11, g12, g12, g22), 2, 2)))
}
```

```
newton <- function(f3, x0, tol = 1e-9, n.max = 100) {
    # Newton's method for optimisation, starting at x0
    # f3 is a function that given x returns the list
    # {f(x), grad f(x), Hessian f(x)}, for some f

    x <- x0
    f3.x <- f3(x)
    n <- 0
    points(x[1], x[2], pch=19)
    while ((max(abs(f3.x[[2]])) > tol) & (n < n.max)) {
        x.old <- x
        x <- x - solve(f3.x[[3]], f3.x[[2]])
        f3.x <- f3(x)
        n <- n + 1
        lines(c(x.old[1], x[1]), c(x.old[2], x[2]))
        points(x[1], x[2])
    }
    if (n == n.max) {
        cat('newton failed to converge\n')
    } else {
        return(x)
    }
}

x <- seq(-2, 2, .1)
y <- seq(-2, 5, .1)
z <- matrix(nrow = length(x), ncol = length(y))
for (i in 1:length(x)) {
  for (j in 1:length(y)) {
    z[i, j] <- Rosenbrock(c(x[i], y[j]))[[1]]
  }
}
par(mfrow=c(1,2))
contour(x, y, z, nlevels = 20)
ascent(g, gradg, c(0, 3), n.max=10000)
```

```
## [1] 0.9993652 0.9987308
```

```
contour(x, y, z, nlevels = 20)
newton(Rosenbrock, c(0, 3))
```




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

In the left figure, we see the path for the Steepest Ascent method. The path for Newton's method is shown on the right. We seems that Newton's method only needs 3 steps to find the minumum, while the Steepest Ascent method requires much more steps (more then $4,700$)! Also, the final answer report by Newton's method is closer to the true minimum than found by Steepest Ascent algorithm.

Question 3 [Exercise 4 of Section 12.8.6 of Jones et al.]

```
f <- function(x) {
  # bug: if x[2] is too large then exp(x[2]) is Inf and cos(x[2]-exp(x[2])) is NaN
  # just set cos(x[2]-exp(x[2])) to 0 in this case
  s <- sum(x^2)
  y <- -(s-2)*(s-1)*s*(s+1)*(s+2)*(2-sin(x[1]^2-x[2]^2)*cos(x[2]-exp(x[2])))
  if (is.nan(y)) {
    y <- -(s-2)*(s-1)*s*(s+1)*(s+2)*2
  }
  return(y)
}

gradf <- function(x) {
  # bug: if x[2] is too large then exp(x[2]) is Inf and cos(x[2]-exp(x[2])) is NaN
  # just set cos(x[2]-exp(x[2])) and sin(x[2]-exp(x[2])) to 0 in this case
  s <- sum(x^2)
  f1 <- -2*x[1]*((s-1)*s*(s+1)*(s+2) + (s-2)*s*(s+1)*(s+2) + (s-2)*(s-1)*(s+1)*(s+2) +
                   (s-2)*(s-1)*s*(s+2) + (s-2)*(s-1)*s*(s+1))*(2-sin(x[1]^2-x[2]^2)*cos(
    2*x[1]*(s-2)*(s-1)*s*(s+1)*(s+2)*cos(x[1]^2-x[2]^2)*cos(x[2]-exp(x[2]))
  f2 <- -2*x[2]*((s-1)*s*(s+1)*(s+2) + (s-2)*s*(s+1)*(s+2) + (s-2)*(s-1)*(s+1)*(s+2) +
                   (s-2)*(s-1)*s*(s+2) + (s-2)*(s-1)*s*(s+1))*(2-sin(x[1]^2-x[2]^2)*cos(
    2*x[2]*(s-2)*(s-1)*s*(s+1)*(s+2)*cos(x[1]^2-x[2]^2)*cos(x[2]-exp(x[2])) -
    (1 - exp(x[2]))*(s-2)*(s-1)*s*(s+1)*(s+2)*sin(x[1]^2-x[2]^2)*sin(x[2]-exp(x[2]))
  if (is.nan(f1)) {
    f1 <- -2*x[1]*((s-1)*s*(s+1)*(s+2) + (s-2)*s*(s+1)*(s+2) + (s-2)*(s-1)*(s+1)*(s+2) +
                     (s-2)*(s-1)*s*(s+2) + (s-2)*(s-1)*s*(s+1))*2
  }
  if (is.nan(f2)) {
    f2 <- -2*x[2]*((s-1)*s*(s+1)*(s+2) + (s-2)*s*(s+1)*(s+2) + (s-2)*(s-1)*(s+1)*(s+2) +
                     (s-2)*(s-1)*s*(s+2) + (s-2)*(s-1)*s*(s+1))*2
  }
  return(c(f1, f2))
}

PlotFunction <- function(pts){
  x <- seq(-1.5, 1.5, .05)
  y <- seq(-1.5, 1.5, .05)
  z = matrix(data=NA, nrow=length(x), ncol=length(x))
  for(i in 1:length(x))  {
    for(j in 1:length(y)) {
      z[i,j] = max(-3,f(c(x[i], y[j])))
    }
```

```
  }
  library(plot3D)
  persp3D(x,y,z,theta=-45, phi=60, axes=TRUE,scale=2, box=TRUE, nticks=5,
        ticktype="detailed")
  points3D(pts$x,pts$y,pts$z, pch=19, col = "black", cex= 1.5, add=T)
}

# use numerical approx of derivative to check gradf is correct
check.gradf <- function(x, e=1e-8) {
  f1 <- (f(x+c(e,0)) - f(x))/e
  f2 <- (f(x+c(0,e)) - f(x))/e
  return(c(gradf(x) - c(f1, f2)))
}
check.gradf(c(0,0))
```

```
## [1] 8e-08 8e-08
```

```
check.gradf(c(1,1))
```

```
## [1] 3.557636e-06 3.275489e-06
```

```
check.gradf(c(1,-1))
```

```
## [1] 3.223097e-06 4.776903e-06
```

```
setwd("C:/R/R-4.0.5/library/spuRs/resources/scripts") # change path to suit
source("ascent.r")

# apply ascent using random starting points
set.seed(5)   # the same starting points for everybody!
n <- 100 # number of starting points
x <- matrix(nrow = n, ncol = 2)
for (i in 1:n) {
  x[i,] <- ascent(f, gradf, runif(2, -1.5, 1.5), n.max=500)
  x[i,] <- round(x[i,], digits=4)
}
```

```
## Warning in cos(x[2] - exp(x[2])): NaNs produced

## Warning in cos(x[2] - exp(x[2])): NaNs produced
```

```
## Warning in cos(x[2] - exp(x[2])): NaNs produced

## Warning in cos(x[2] - exp(x[2])): NaNs produced

## Warning in cos(x[2] - exp(x[2])): NaNs produced

## Warning in cos(x[2] - exp(x[2])): NaNs produced
```

```
xx <- unique(x)   # filter out same optima
print(xx)
```

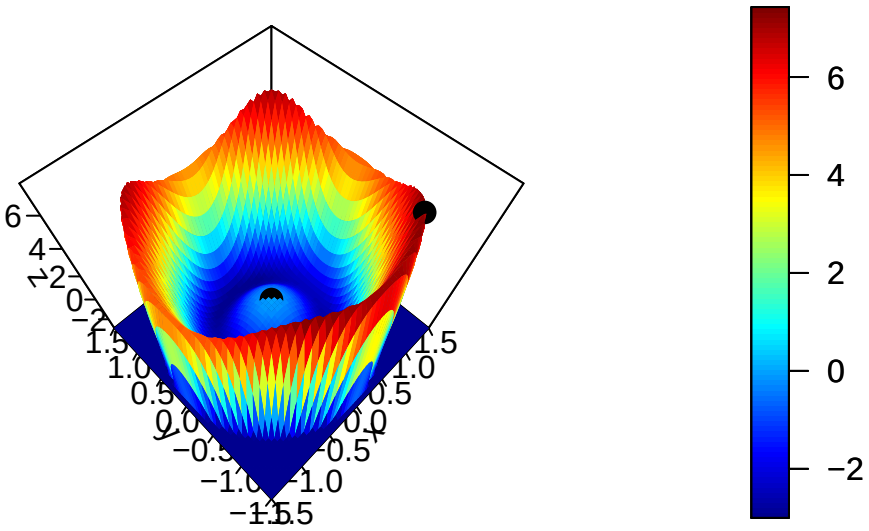
```
##          [,1]    [,2]
##  [1,] -0.9063  0.9072
##  [2,]  0.6655 -1.0941
##  [3,]  0.6654 -1.0942
##  [4,] -0.6655 -1.0941
##  [5,]  0.0000  0.0000
##  [6,]  0.9063  0.9072
##  [7,] -0.6654 -1.0942
##  [8,] -0.6654 -1.0941
##  [9,]  0.6654 -1.0941
```

```
# below I've tried to plot the points (black circles) in the 3D surface plot
# at least one of the optima is located at the mountain ridge
zz <- rep(0,nrow(xx))
for(i in nrow(xx)){zz[i]<-f(xx[i,])}
pts <-data.frame(x=xx[,1],y=xx[,2],z=zz)
PlotFunction(pts)


#-----------------------------------------
# some fancy code for plotting the solutions

# a version of ascent that returns intermediate values
ascent2 <- function(f, grad.f, x0, tol = 1e-9, n.max = 100) {
  # steepest ascent algorithm
  # find a local max of f starting at x0
  # function grad.f is the gradient of f
  x <- rbind(x0, line.search(f, x0, grad.f(x0)))
  n <- 2
  while ((f(x[n,]) - f(x[n-1,]) > tol) & (n <= n.max)) {
    x <- rbind(x, line.search(f, x[n,], grad.f(x[n,])))
    n <- n + 1
  }
```

```
  return(x)
}

# apply ascent using random starting points
n <- 10 # number of starting points
xlist <- list()
for (i in 1:n) {
  xlist[[i]] <- ascent2(f, gradf, runif(2, -1.5, 1.5), n.max=500)
}
```

```
## Warning in cos(x[2] - exp(x[2])): NaNs produced

## Warning in cos(x[2] - exp(x[2])): NaNs produced

## Warning in cos(x[2] - exp(x[2])): NaNs produced

## Warning in cos(x[2] - exp(x[2])): NaNs produced

## Warning in cos(x[2] - exp(x[2])): NaNs produced
```

```
# plot contours with trajectories on top
fgrid <- c()
for (x in seq(-1.5, 1.5, .02)) {
  for (y in seq(-1.5, 1.5, .02)) {
    fgrid <- c(fgrid, x, y, max(f(c(x,y)),-3))
  }
}
fgrid <- data.frame(matrix(fgrid, ncol=3, byrow=T))
names(fgrid) <- c('x','y','z')
library(lattice)
```



```
print(contourplot(z~x*y, fgrid, line.list=xlist,
                  panel = function(x, y, z, line.list, ...) {
                    panel.contourplot(x, y, z, ...)
                    for (ln in line.list) {
                      panel.lines(ln[,1], ln[,2], ...)
                      panel.points(ln[dim(ln)[1],1], ln[dim(ln)[1],2], ...)
                    }
                  },
                  cuts=20, contour=F, labels=F, region=T, colorkey=F))
```