Christian Kammerer, Jakob Lindner

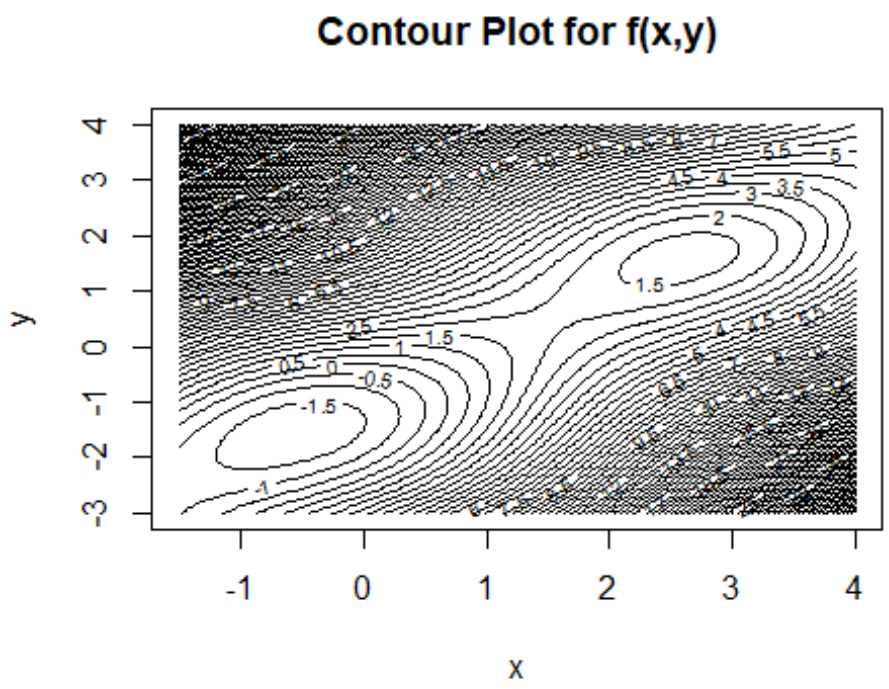
Report for Computer Lab 2 in Computational Statistics

Question 1: Optimization of a two-dimensional function

$$g(x) = sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1$$

a. Make a contour plot of the function.

```
# using example code from lecture
x1grid <- -30:80/20
x2grid <- -60:80/20
dx1 <- length(x1grid)
dx2 <- length(x2grid)
dx <- dx1*dx2

fx <- matrix(rep(NA, dx), nrow=dx1)
for (i in 1:dx1)
  for (j in 1:dx2)
  {
    x <- x1grid[i]
    y <- x2grid[j]
    fx[i, j] <- f(x, y)
  }
mfx <- matrix(fx, nrow=dx1, ncol=dx2)


contour(x1grid, x2grid, mfx, nlevels=100, xlab="x", ylab="y", main="Contour
Plot for f(x,y)")
```



The high amount of lines in the plot makes it hard to read in the steeper areas, but it was chosen to illustrate that the function has two (local) minimums.

b. Derive the gradient and Hessian matrix for the function and write R-code for them.

```
# get gradient
deriv(expression(sin(x+y) + (x-y)^2 - 1.5*x + 2.5*y + 1), namevec = c("x",
"y"))
```

```
## expression({
##     .expr1 <- x + y
##     .expr3 <- x - y
##     .expr11 <- cos(.expr1)
##     .expr12 <- 2 * .expr3
##     .value <- sin(.expr1) + .expr3^2 - 1.5 * x + 2.5 * y + 1
##     .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
##         "y")))
##     .grad[, "x"] <- .expr11 + .expr12 - 1.5
##     .grad[, "y"] <- .expr11 - .expr12 + 2.5
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
grad_g <- function(x, y) {
  grad_x <- cos(x + y) + 2 * (x - y) - 1.5
  grad_y <- cos(x + y) - 2 * (x - y) + 2.5
  return(c(grad_x, grad_y))
}
```

```r
hessian_g <- function(x, y) {
  h11 <- -sin(x+y) + 2
  h12 <- -sin(x+y) - 2
  h21 <- -sin(x+y) - 2
  h22 <- -sin(x+y) + 2
  return(matrix(c(h11, h12, h21, h22), nrow = 2, byrow = TRUE))
}
```

c. Write your own function applying the Newton algorithm that has the starting value (x0, y0) as a parameter.

```r
newton <- function(starting_point) {
  x0 <- starting_point[1]
  y0 <- starting_point[2]

  x_t <- c(x0, y0)

  it <- 0
  while(TRUE) {
    # function only defined for this range
    stopifnot(x_t[1] > -1.5 & x_t[1] < 4 &x_t[2] > -3 &x_t[2] < 4 )


    x_t1 <- x_t - solve(hessian_g(x_t[1],x_t[2])) %*% grad_g(x_t[1],x_t[2])

    # use absolute stopping criterion or a maximum number of iterations
    if (norm(x_t1 - x_t, type="2") < 0.005 | it >= 10000){
      break
    }

    x_t <- x_t1
    it <- it + 1
  }

  return(c(x_t, it))
}
```

d. Test several starting values such that you have examples which give at least three different results.

```r
newton(c(0,-1)) # finds minimum with convergence in 3 iterations
```

```
## [1] -0.5471897 -1.5471897  3.0000000
```

```r
newton(c(2,-1)) # converges in 3 iteration
```

```
## [1] 1.5471897 0.5471897 3.0000000
```

```r
newton(c(3,1)) # finds minimum in 2 iterations, but only local
```

```
## [1] 2.594424 1.594424 2.000000
```

e. Investigate the candidate results which you have got. Is one of the candidates a global minimum? Which type of solutions are the other candidates?

Using (0, -1) as starting value results in finding a minimum. It is even global minimum, referring to the contour plot. (3,1) gives a minimum as well, but only a local one. The function converges as well when starting with (2,-1) but only finds the saddle point between the two minima.

Question 2: Maximum likelihood

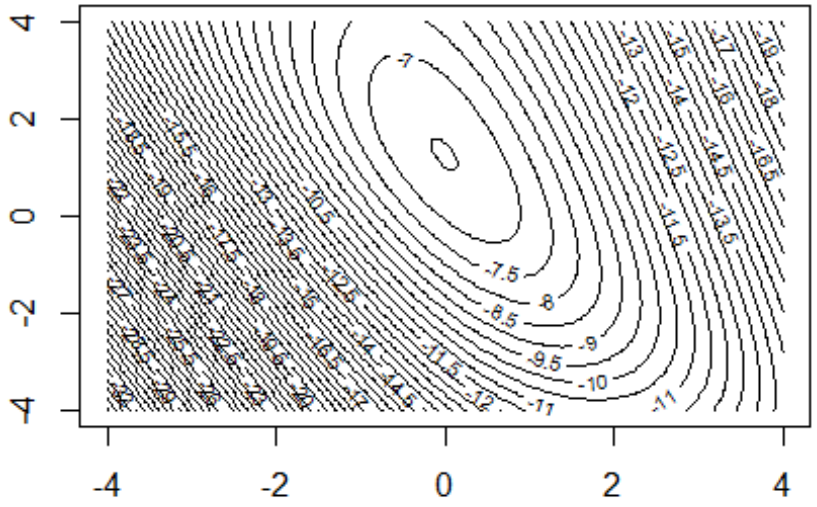
a. Write a function for an ML-estimator for (β0, β1) using the steepest ascent method with a step-size-reducing line search (back-tracking). For this, you can use and modify the code for the steepest ascent example of the lecture. The function should count the number of function and gradient evaluations.

```
steepestasc <- function(x0, eps=1e-5, alpha0=1, factor=0.5)
{
  func_evals <- 0
  grad_evals <- 0
  xt   <- x0
  conv <- 999
  points(xt[1], xt[2], col=2, pch=4, lwd=3)
  while(conv>eps)
  {
    alpha <- alpha0
    xt1   <- xt
    xt    <- xt1 + alpha*gradient(xt1)
    grad_evals <- grad_evals + 1

    while (g(xt)<g(xt1))
    {
      func_evals <- func_evals + 2 # for the two evals in while condition

      # adjust alpha by given factor
      alpha <- alpha*factor

      xt    <- xt1 + alpha*gradient(xt1)
      grad_evals <- grad_evals + 1
    }
    func_evals <- func_evals + 2 # for the two evals in while condition that
broke the loop

    points(xt[1], xt[2], col=2, pch=4, lwd=1)
    conv <- max(abs(xt - xt1))
  }
  points(xt[1], xt[2], col=4, pch=4, lwd=3)

  cat("Function evaluations:", func_evals, "\n")
  cat("Gradient evaluations:", grad_evals, "\n")
```

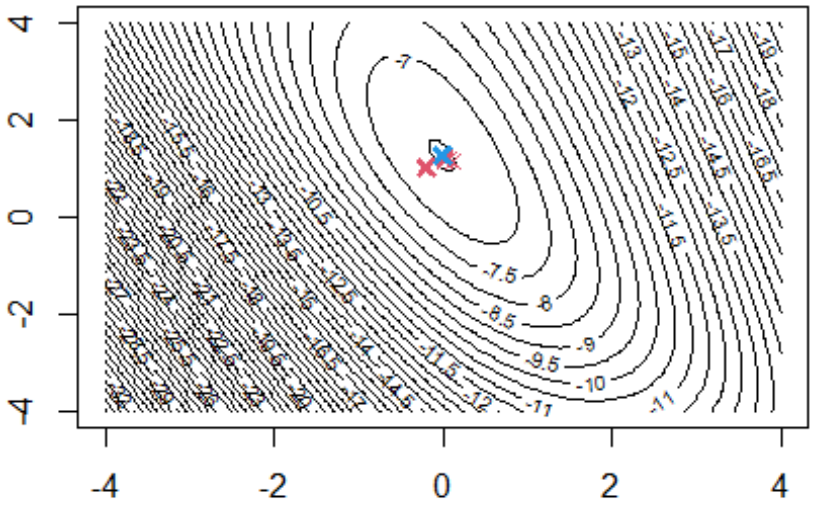
```
  xt
}
```

For the implementation the code from the lecture was used. Additionally, the gradient and function evaluations get counted and the factor with which the alpha value is adjusted every step becomes a parameter of the function. To be able to trust five digits of the result (see task b.) the euclidean distance as stopping criterion was replaced by the maximum absolute difference. This ensures that every parameter is stable as it cannot be compensated by a high precision for the other parameter.

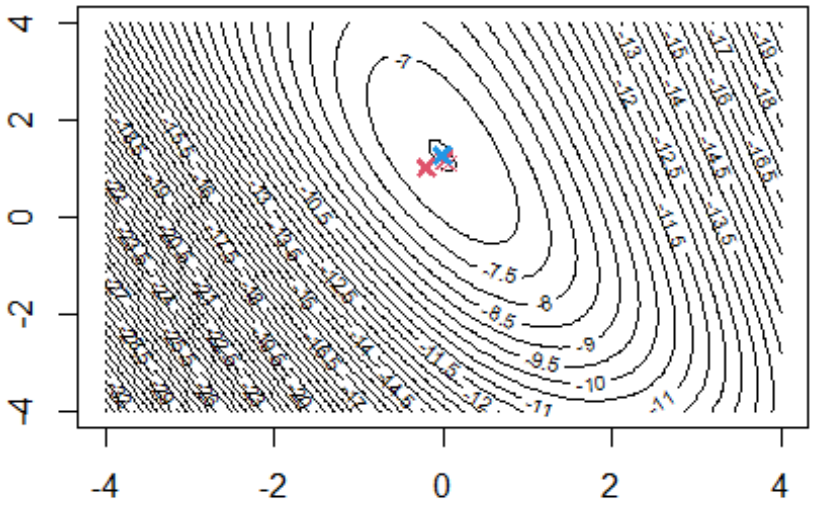
b. Compute the ML-estimator with the function from a. for the data (xi, yi) above. Use a stopping criterion such that you can trust five digits of both parameter estimates for β0 and β1. Use the starting value (β0, β1) = (−0.2, 1). The exact way to use backtracking can be varied. Try two variants and compare the number of function and gradient evaluations performed to convergence.




```
contour(x1grid, x2grid, mgx, nlevels=50)
steepestasc(c(-0.2, 1), factor=0.5)
```




```
## Function evaluations: 108
## Gradient evaluations: 54

## [1] -0.009353495  1.262798921

contour(x1grid, x2grid, mgx, nlevels=50)
steepestasc(c(-0.2, 1), factor=0.4)
```




```
## Function evaluations: 78
## Gradient evaluations: 39

## [1] -0.009350725  1.262797821
```

According to the contour plot the minimum is correctly approximated for both factors, for the default value 0.5 and for 0.4. But the lower factor results in fewer evaluations of the function and the gradient. In other words, it converges faster.

c. Use now the function optim with both the BFGS and the Nelder-Mead algorithm. Do you obtain the same results as for b.? Is there any difference in the precision of the result? Compare the number of function and gradient evaluations that are given in the standard output of optim.

```
optim(c(-0.2, 1), g, gr=gradient, method="BFGS", control=list(fnscale=-1))

## $par
## [1] -0.009356126  1.262812832
##
## $value
## [1] -6.484279
##
## $counts
## function gradient
##       12        8
##
## $convergence
## [1] 0
##
```

```
## $message
## NULL
```

```
optim(c(-0.2, 1), g, method="Nelder-Mead",control=list(fnscale=-1))
```

```
## $par
## [1] -0.009423433  1.262738266
##
## $value
## [1] -6.484279
##
## $counts
## function gradient
##       47       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Both have good precision, the precision of Nelder-Mead is a bot lower than for BFGS and steepest ascent. BGFS does the fewest evaluations, followed my Nelder-Mead, while steepest ascent takes more than double the amount of total evaluations.

d. Use the function glm in R to obtain an ML-solution and compare it with your results before.

```
model <- glm(y ~ x, family="binomial")
model$iter # the number of iterations of IWLS used.
```

```
## [1] 4
```

```
model$coefficients
```

```
##  (Intercept)            x
## -0.009359853  1.262823430
```

The glm()-function shows similar results as the steepest ascent with factor 0.4, where it is the same for the same 5 digits for both betas. Steepest ascent with factor 0.5 is the same in 5 digits for beta0 and in 4 for beta1, same goes for BFGS. Nelson-Mead results in the same beta0 and beta1 in 3 digits. GLM takes 4 iterations of IWLS. But this number cannot really be used to compare convergence speed to the other algorithms.

Apendix

```
# Quesion 1
f <- function(x, y) {
  return (sin(x+y) + (x-y)^2 - 1.5*x + 2.5*y + 1)
}

# using example code from lecture
x1grid <- -30:80/20
x2grid <- -60:80/20
dx1 <- length(x1grid)
```

```r
dx2 <- length(x2grid)
dx <- dx1*dx2

fx <- matrix(rep(NA, dx), nrow=dx1)
for (i in 1:dx1)
  for (j in 1:dx2)
  {
    x <- x1grid[i]
    y <- x2grid[j]
    fx[i, j] <- f(x, y)
  }
mfx <- matrix(fx, nrow=dx1, ncol=dx2)


contour(x1grid, x2grid, mfx, nlevels=100, xlab="x", ylab="y", main="Contour
Plot for f(x,y)")


# get gradient
deriv(expression(sin(x+y) + (x-y)^2 - 1.5*x + 2.5*y + 1), namevec = c("x",
"y"))

grad_g <- function(x, y) {
  grad_x <- cos(x + y) + 2 * (x - y) - 1.5
  grad_y <- cos(x + y) - 2 * (x - y) + 2.5
  return(c(grad_x, grad_y))
}

hessian_g <- function(x, y) {
  h11 <- -sin(x+y) + 2
  h12 <- -sin(x+y) - 2
  h21 <- -sin(x+y) - 2
  h22 <- -sin(x+y) + 2
  return(matrix(c(h11, h12, h21, h22), nrow = 2, byrow = TRUE))

}

newton <- function(starting_point) {
  x0 <- starting_point[1]
  y0 <- starting_point[2]

  x_t <- c(x0, y0)

  it <- 0
  while(TRUE) {
    x_t1 <- x_t - solve(hessian_g(x_t[1],x_t[2])) %*% grad_g(x_t[1],x_t[2])

    # use absolute stopping criterion or a maximum number of iterations
    if (norm(x_t1 - x_t, type="2") < 0.005 | it >= 10000){
      break
```

```r
    }

    x_t <- x_t1
    it <- it + 1
  }

  return(c(x_t, it))
}

newton(c(0,-1)) # finds minimum with convergence in 7 iterations
newton(c(2,-1)) # does not find an optimum within 1000 iterations, diverges
newton(c(3,2)) # finds minimum in 7 iterations, but only local


# Question 2
x <- c(0,0,0,0.1,0.1,0.3,0.3,0.9,0.9,0.9)
y <- c(0,0,1,0,1,1,1,0,1,1)

# The function g to be maximised, partial derivatives dg1, dg2, and gradient
g <- function(beta)
{
  beta0 <- beta[1]
  beta1 <- beta[2]

  p <- 1 / (1 + exp(-(beta0 + beta1 * x)))  # logistic regression

  log_likelihood <- sum(y * log(p) + (1 - y) * log(1 - p))

  return(log_likelihood)
}


gradient <- function(beta)
{
  beta0 <- beta[1]
  beta1 <- beta[2]

  p <- 1 / (1 + exp(-(beta0 + beta1 * x)))

  grad_beta0 <- sum(y - p)
  grad_beta1 <- sum((y - p) * x)

  return(c(grad_beta0, grad_beta1))
}


steepestasc <- function(x0, eps=1e-5, alpha0=1, factor=0.5)
{
  func_evals <- 0
  grad_evals <- 0
```

```
  xt   <- x0
  conv <- 999
  points(xt[1], xt[2], col=2, pch=4, lwd=3)
  while(conv>eps)
  {
    alpha <- alpha0
    xt1   <- xt
    xt    <- xt1 + alpha*gradient(xt1)
    grad_evals <- grad_evals + 1

    while (g(xt)<g(xt1))
    {
      func_evals <- func_evals + 2 # for the two evals in while condition

      # adjust alpha by given factor
      alpha <- alpha*factor

      xt    <- xt1 + alpha*gradient(xt1)
      grad_evals <- grad_evals + 1
    }
    func_evals <- func_evals + 2 # for the two evals in while condition that
broke the loop

    points(xt[1], xt[2], col=2, pch=4, lwd=1)
    conv <- max(abs(xt - xt1))
  }
  points(xt[1], xt[2], col=4, pch=4, lwd=3)

  cat("Function evaluations:", func_evals, "\n")
  cat("Gradient evaluations:", grad_evals, "\n")

  xt
}

x1grid <- seq(-4, 4, by=0.05)
x2grid <- seq(-4, 4, by=0.05)
dx1 <- length(x1grid)
dx2 <- length(x2grid)
dx  <- dx1*dx2
gx  <- matrix(rep(NA, dx), nrow=dx1)
for (i in 1:dx1)
  for (j in 1:dx2)
  {
    gx[i,j] <- g(c(x1grid[i], x2grid[j]))
  }
mgx <- matrix(gx, nrow=dx1, ncol=dx2)
contour(x1grid, x2grid, mgx, nlevels=50)  # Note: For other functions g, you
might need to choose another nlevels-value to get a good impression of the
function
```

```r
contour(x1grid, x2grid, mgx, nlevels=50)
steepestasc(c(-0.2, 1), factor=0.5)
contour(x1grid, x2grid, mgx, nlevels=50)
steepestasc(c(-0.2, 1), factor=0.4)

optim(c(-0.2, 1), g, gr=gradient, method="BFGS", control=list(fnscale=-1))

optim(c(-0.2, 1), g, method="Nelder-Mead",control=list(fnscale=-1))

model <- glm(y ~ x, family="binomial")
model$iter # the number of iterations of IWLS used.
model$coefficients
```