Exam Optimization

Kasper Rosenkrands

Fall 2019

Contents

1 Introduction

This is a collection of the possible exam subjects at the optimization exam of the 2019 fall semester at Aalborg University.

2 Line search

A line serach algorithm chooses a direction p_k and searches along this direction from the current iterate x_k for a new iterate with a lower function value.

2.1 Search direction

The first step is to find a search direction, the most obvious one being $-\nabla f_k$. This direction have intuitive appeal as this is the direction that decreases f most rapidly. With a search direction in place we can now focus on the step.

2.2 Step size

For demonstration purposes, before looking more into intelligent ways of choosing step sizes, we will implement a simple algorithm which uses a constant step size.

2.2.1 Basic gradient descent algorithm

First we must import some data to work with. We will use the cars dataset from base R.

```
s <- cars$speed; d <- cars$dist
data_length <- length(s)
```

Then we define our objective function with we choose as the MSE. Furthermore we will provide the gradient.

```
f <- function(x) 1/data_length * sum(((x[1] + x[2] * s) - d)^2)
g <- function(x) {
  gradient.a <- 2/data_length * sum((x[1] + x[2] * s) - d)
  gradient.b <- 2/data_length * sum(s * ((x[1] + x[2] * s) - d))
  return(c(gradient.a, gradient.b))
}
```

Next we start to implement a basic gradient descent algorithm.

```
# Usign the norm as a criteria for convergence
norm2 <- function(x) norm(as.matrix(x), type = "2")

basic_gd <- function(output = FALSE, x_0 = c(-19,3.6), alpha = 1e-4, tolerance = 1e-4,
                     k_max = 100000) {
  x_k <- x_0
  f_iterates <- rep(0, k_max + 1)
  keep_going <- TRUE
  k <- 0
  f_iterates[1] <- f(x_k)
  while(keep_going) {
    k <- k + 1
    g_k <- g(x_k)
    p_k <- -g_k
    x_k <- x_k + alpha * p_k
    keep_going <- ((norm2(g_k) >= tolerance) & (k < k_max))
    f_iterates[k + 1] <- f(x_k)
  }
  if((norm2(g_k) <= tolerance)) {
    cat('Converged in', k, 'steps', '\n')
  } else {
    cat('Iteration limit reached', '\n')
  }
```

```
  cat('f(x_k) = ',f(x_k),'\t','x_k = ',x_k,'\n')
  if(output == TRUE) {
    return(f_iterates)
  }
}
basic_gd()
```

```
## Iteration limit reached
## f(x_k) =  227.0737    x_k =  -17.75634 3.94273
```

We did not converge in 100.000 steps, so we can assume that this algortihm is not very fast.

2.2.2 The wolfe conditions

the best choice of step length we can make is the global minimizer of the function

$$\phi(\alpha) = f(x_k + \alpha p_k), \quad \alpha > 0. \tag{1}$$

However it is not practical to calculate this every time as it would require to many evaluations of the objective function and/or gradient. We can however use some conditions to determine desireable step size at minimal cost.

Sufficient decrease: This condition ensure that the step insures a sufficient decrease in the objective function and can be formulated as

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k, \quad c_1 \in (0, 1). \tag{2}$$

In other words the condtion ensures that the reduction in the objective function is proportional to the step size and the directional derivative $\nabla f_k^T p_k$.

Curvature condition: This conditions ensures that when the objective function is decreasing at a rate beyond a certain point we will continue in that direction until gradient is not as steep anymore. The condition is formulated as follows

$$\nabla f(x_k + \alpha p_k)^T p_k \geq c_2 \nabla f_k^T p_k, \quad c_2 \in (c_1, 1). \tag{3}$$

These two conditions are known collectively as the *Wolfe conditions*, and if we furthermore pose a restriction on how positive the gradient can be we have what are called the *strong Wolfe conditions*

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k, \qquad c_1 \in (0, 1) \tag{4}$$

$$|\nabla f(x_k + \alpha p_k)^T p_k| \geq c_2 |\nabla f_k^T p_k|, \qquad c_2 \in (c_1, 1). \tag{5}$$

2.2.3 A strong Wolfe gradient descent algorithm

```
norm2 <- function(x) norm(as.matrix(x), type = "2")

# Implementation of Algorithm 3.5
# (Line Search Algorithm)
alpha <- function(a_0, x_k, c1, c2, r_k) {
  a_max <- 4*a_0
  f_k <- f(x_k)
  phi_k <- f_k
  a_1 <- a_0
  a0 <- 0
  a_k <- a_1
  a_k_old <- a0
```

```
  k <- 0
  k_max <- 10000
  done <- FALSE
  while(!done) {
    k <- k + 1
    f_k <- f(x_k)
    g_k <- g(x_k)
    p_k <- -g_k
    phi_k_old <- f(x_k + a_k_old * p_k)
      phi_k <- f(x_k + a_k * p_k)
      dphi_k_0 <- t(g(x_k)) %*% p_k
      l_k <- f_k + c1 * a_k * dphi_k_0
      if ((phi_k > l_k) || ((k > 1) && (phi_k >= phi_k_old))) {
        return(zoom(a_k_old, a_k, x_k, c1, c2))
      }
    dphi_k <- t(g(x_k + a_k * p_k)) %*% p_k
      if (abs(dphi_k) <= -c2*dphi_k_0) {
        return(a_k)
      }
    if (dphi_k >= 0) {
      return(zoom(a_k, a_k_old, x_k, c1, c2))
    }
    a_k_old <- a_k
    a_k <- r_k*a_k + (1 - r_k)*a_max
    done <- (k > k_max)
  }
  return(a_k)
}

# Implementation of Algorithm 3.6
# (Zoom Algorithm)
zoom <- function(a_lo, a_hi, x_k, c1, c2) {
  f_k <- f(x_k)
  g_k <- g(x_k)
  p_k <- -g_k
  k <- 0
  k_max <- 10000   # Maximum number of iterations.
  done <- FALSE
  while(!done) {
    k <- k + 1
    phi_lo <- f(x_k + a_lo * p_k)
      a_k <- 0.5*(a_lo + a_hi)
      phi_k <-  f(x_k + a_k * p_k)
      dphi_k_0 <- t(g(x_k)) %*% p_k
      l_k <-  f_k + c1 * a_k * dphi_k_0
      if ((phi_k > l_k) || (phi_k >= phi_lo)) {
        a_hi <- a_k
      } else {
        dphi_k <-  t(g(x_k + a_k * p_k)) %*% p_k
          if (abs(dphi_k) <= -c2*dphi_k_0) {
            return(a_k)
          }
        if (dphi_k*(a_hi - a_lo) >= 0) {
```

```
          a_hi <- a_lo
        }
        a_lo <- a_k
      }
    done <- (k > k_max)
  }
  return(a_k)
}
```

We can then create a function that uses the line search and zoom algortihm to actually minimize the objective function.

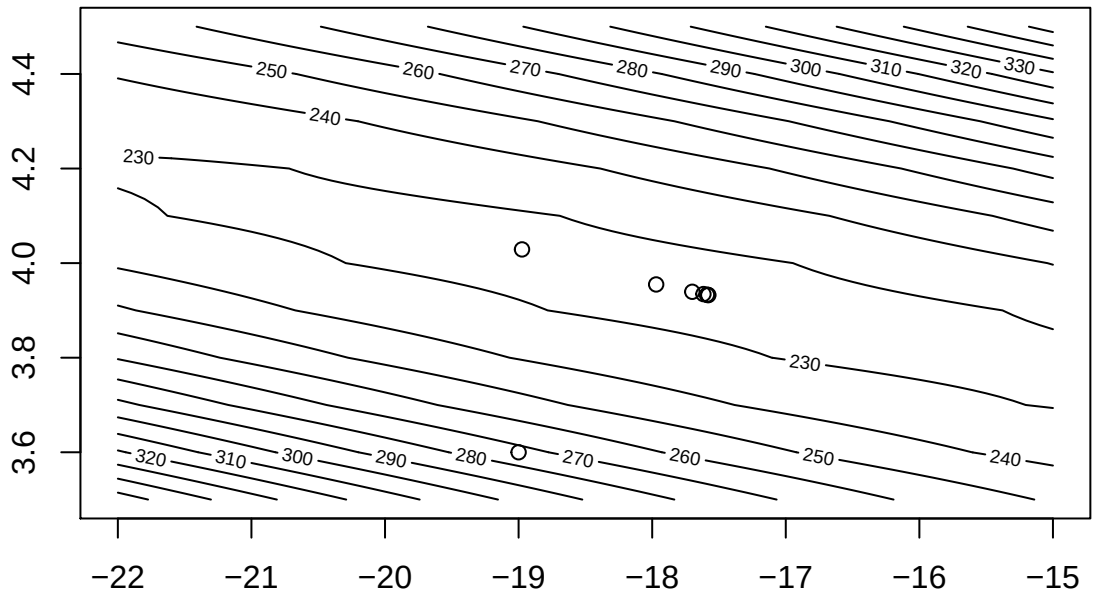
```
strong_wolfe_gd <-
  function(f, g, x_0, a_0 = 1, r_k = 0.5,
           c1 = 1e-4, c2 = 4e-1, tol_grad_f = 1e-4,
           k_max = 100000, output = FALSE, plot = FALSE) {
  f_iterates <- rep(0,k_max + 1)
  keep_going <- TRUE

  if (plot == TRUE) {
    agrid <- seq(-22,-15,.1)
    bgrid <- seq(3.5,4.5,.1)
    zvalues <- matrix(NA_real_,length(agrid),length(bgrid))
    for (i in 1:length(agrid)) {
      for (j in 1:length(bgrid)) {
        zvalues[i,j] <- f(c(agrid[i],bgrid[j]))
      }
    }
    my_levels <- seq(150, 400, 10)
    contour(agrid, bgrid, zvalues, levels = my_levels)
    points(x_0[1], x_0[2])
  }

  x_k <- x_0
  k <- 0
  f_iterates[1] <- f(x_k)
  while (keep_going) {
    k <- k + 1
    a_k <- alpha(a_0, x_k, c1, c2, r_k)
    g_k <- g(x_k)
    p_k <- -g_k
    x_k <- x_k + a_k*p_k
    keep_going <- (norm2(g_k) >= tol_grad_f) & (k < k_max)
    f_iterates[k + 1] <- f(x_k)

    #plot iterates
    if (plot == TRUE) {
      if (k == 1 || k %% 100 == 0) {
        points(x_k[1], x_k[2])
      }
    }
  }
  if (!output == TRUE || !plot == TRUE) {
    if((norm2(g_k) <= tol_grad_f)) {
```

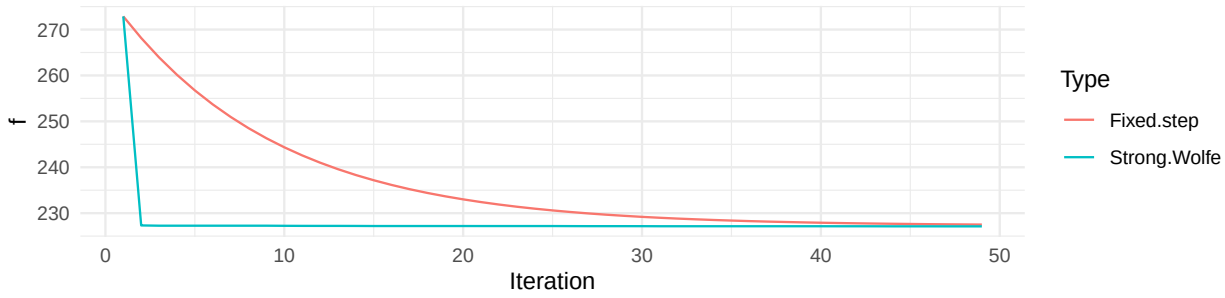
```
      cat('Converged in', k, 'steps', '\n')
    } else {
      cat('Iteration limit reached', '\n')
    }
    cat('f(x_k) = ',f(x_k),'\t', 'x_k = ', x_k, '\n')
  }
  if (output == TRUE) {
    return(f_iterates)
  }
}
strong_wolfe_gd(f,g, c(-19, 3.6), output = FALSE, plot = TRUE)
```




```
## Converged in 662 steps
## f(x_k) =  227.0704    x_k =  -17.57954 3.932433
```

We can conclude that his algorithm converges must faster for the given starting value. This algortihm converged in 662 steps whereas the simpler algorithm did not converge in 100.000 iterations.

To see the difference in the two algorithms for just the first 50 iterations the following plot can be considered. Notice how much difference progress there is the two algorithms in just the first step.



I suspect that the reason for the little reduction in the objective function after the first step is due to poor scaling.

3 Calculating derivatives

Many numerical optimization algorithms use the gradient to minimize the respective objective function. However sometimes it can be time-consuming if the objective function is complicated. Therefore it would beneficial to have the algortihm calculate the derivative automatically. There are several different approaches that could be taken. The first approach we will consider is called finite differencing.

3.1 Finite differencing

The idea is to estimate the derivatives by observing the change in function values in response to small perturbations of the unknowns near a given point. In the case of forward-difference our estimate of the gradient for a function is given by

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + \varepsilon e_i)}{\varepsilon}.$$

In the case of central-difference our estimate is given as

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + \varepsilon e_i) - f(x - \varepsilon e_i)}{2\varepsilon}.$$

These estimate arises from Taylor's formula that states

$$f(x + h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) - O(h^4),$$

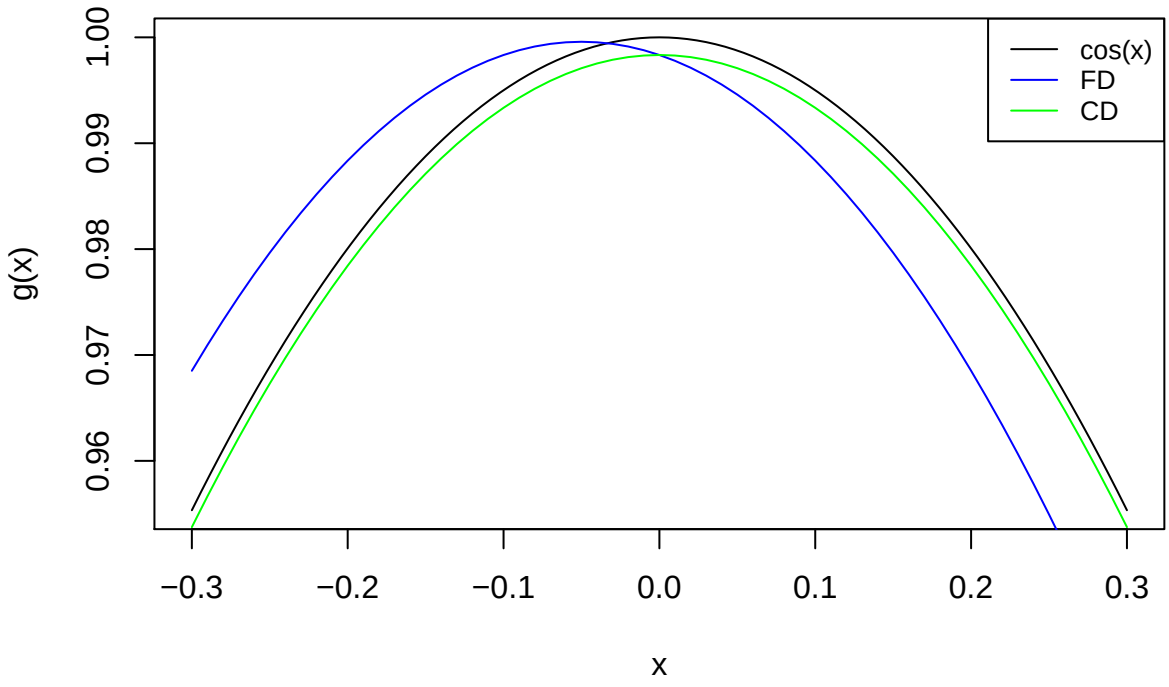
rearranging gives that

$$f'(x) = \frac{f(x + h) - f(x)}{h} + \frac{1}{2}hf''(x) + \frac{1}{6}h^2 f'''(x) - O(h^3).$$

By considering only the first term after the equality we give rise to a truncation error. We are only interested in $0 < h < 1$, such that the truncation error becomes $O(h)$. Furthermore the truncation error decreases as h decreases. The could lead us to believe that we can then choose a very small h a go on our merry way. However this is not the case due to floating point arithmetic producing a round-off error. It can be shown that the round-off error bound is $O(\varepsilon_M/h)$, i.e. the round-off error increases with h. Thus we have established that there is a trade-off between truncation and round-off error.

The difference between the estimates for the two methods can be seen in the following plot.

```
f <- function(x) sin(x)
g <- function(x) cos(x)
fd <- function(x){
  h <- 1e-1
  (f(x + h) - f(x))/h
}
cd <- function(x){
  h <- 1e-1
  (f(x + h) - f(x - h))/(2*h)
}
int <- seq(-0.3,0.3, 0.01)
curve(g, min(int), max(int))
lines(int, fd(int), col = 'blue')
lines(int, cd(int), col = 'green')
legend('topright', legend = c('cos(x)', 'FD', 'CD'),
       col = c('black', 'blue', 'green'),
       lty = 1, cex = 0.8)
```



3.1.1 What would happen if we extend the central-difference to also use $f(x-2h)$ and $f(x+2h)$? Hint: consider the Taylor series up to a sufficiently high power of h. Hint: "five-point stencil".

First we deduce the five point stencil rule by Taylor expansion

$$f(x \pm h) = f(x) \pm hf'(x) + \frac{h^2}{2!}f''(x) \pm \frac{h^3}{3!}f'''(x) + O(h^4)$$

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{3}f'''(x)O(h^4)$$

$$f(x+2h) - f(x-2h) = 4hf'(x) + \frac{8h^3}{3}f'''(x) + O(h^4)$$

To get rid of the third term we write

$$8f(x+h) - 8f(x-h) - f(x+2h) + f(x-2h) = 12hf'(x) + O(h^4)$$

Thus we have

$$f'(x) = \frac{8f(x+h) - 8f(x-h) - f(x+2h) + f(x-2h)}{12h}$$

where our truncation error is $O(h^4)$, which all else equal is better than for both forward and central differencing. These are $O(h)$ and $O(h^2)$ respectively.

3.1.2 Analyse this method in comparison with FD and CD (theoretically and practically on specific examples)

3.1.3 What are the advantages and disadvantages of the different finite difference methods?

When we use more point to calculate the derivative we get a smaller truncation error thus allowing us to use a smaller value for h. The disadvantage is that the function needs to be evaluated in more points which could be expensive. Therefore the choice of method depends on the specific problem.

3.2 Exercise 2

3.3 Exercise 3

4 Quasi Newton

5 Least Squares

6 Constrained Optimization