131890-S Optimization methods **Winter 2023/2024**

Lecture 5: More local methods

Daniel Kaszyński *8 December 2023 r.*

5.1 Newton Descent

5.1.1 Theory behind Newton Descent

Newton descent is another iterative algorithm often used to minimize functions, similarly to gradient descent and steepest descent discussed in previous lectures. It uses second order approximations to find critical points of given function f. The basic idea of Newton's method in optimization is to iteratively update an initial guess for the optimal solution based on the function's first and second derivatives. The update rule is derived from the Taylor series expansion of the function around the current guess.

To better understand how this algorithm works, let's start with Taylor series approximation around point x_k going up to second degree derivative (second-order Taylor approximation of f):

$$f(x_k + h) \approx f(x_k) + f'(x_k)h + \frac{1}{2}f''(x_k)h^2 \tag{5.1}$$

The goal of this methodology is to find h for which the function around a given point x_k changes the fastest. We assume that x_k in each step is given and therefore constant. With this information we can make an observation that Taylor series is a poly-nominal formula for approximating values ($f(x_k)$, $f'(x_k)$ and $\frac{1}{2}f''(x_k)$ are constant leaving us with only h to work with).

Next up, we can apply First Order Conditions and equate the derivative of our approximation to zero:

$$0 = \frac{d}{dh}\left(f(x_k) + f'(x_k)h + \frac{1}{2}f''(x_k)h^2\right) = f'(x_k) + f''(x_k)h \tag{5.2}$$

Then, using a simple formula transformation, we can obtain the formula for the optimal value of h:

$$h = -\frac{f'(x_k)}{f''(x_k)} \tag{5.3}$$

This formula for h can be used to calculate steps taken in subsequent iterations of Newton descent algorithm for 1D functions.

Definition 1: Newton Descent for 1D function

Given that Newton descent is an iterative algorithm, by a point calculated at $(k-1)$-th step of the gradient descent of function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ we call:

$$x_{k+1} = x_k + h = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{5.4}$$

where $k \in \mathbb{N}$ is an iteration number, $f'(x_k)$ is the derivative of a function f in point x_k and $f''(x_k)$ is the second derivative of a function f in point x_k.

The Newton descent presented in formula above (5.4) can be generalized to more than one dimension by replacing the derivative with the gradient and the reciprocal of the second derivative with the inverse of the Hessian matrix (because as we know from previous lectures, Hessian matrix is a generalization of a second derivative).

Definition 2: Newton Descent

Given that Newton descent is an iterative algorithm, by a point calculated at $(k-1)$-th step of the gradient descent of function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ we call:

$$x_{k+1} = x_k - H_f(x_k)^{-1} \nabla_f(x_k) \tag{5.5}$$

where $k \in \mathbb{N}$ is an iteration number, $H_f(x_k)^{-1}$ is the inverse of the Hessian matrix and $\nabla_f(x_k)$ is the gradient of a function f calculated in point x_k.

Caution! When using Newton descent we do not know by default if we are optimizing functions towards minimum, maximum or other critical points of given function. Newton descent is steering towards stationarity point of a function. That's why it is good practice to use Newton descent as the last optimization step in order to more quickly reach the extremum of the function that we approached using another algorithm.

Example 1. Let $f(x) = ax^2 + bx + c$ and x_0 a starting point for Newton descent algorithm.

Using Newton descent formula for the given 1D function f, we get:

$$x_1 = x_0 - \frac{2ax_0 + b}{2a} \tag{5.6}$$

We can then divide the fraction obtained on the right side of the equation into two parts:

$$x_1 = x_0 - \frac{2ax_0}{2a} - \frac{b}{2a} \tag{5.7}$$

Thanks to this transformation, we can eliminate expressions containing x_0:

$$x_1 = -\frac{b}{2a} \tag{5.8}$$

The obtained formula for x_1 is the formula for a vertex of second order polynomial function (in this case minimum). So in one step we are able to find the extremum of a function. As we can see, this algorithm is very efficient whenever we face problems dealing with quadratic forms.

As we can see in the formula for Newton descent (5.5), to calculate the subsequent steps taken in each iteration of the algorithm, we must be able to get the Hessian of the function. To be able to calculate the Hessian of a function one must derive a partial derivative over x_i and x_j.

$$H_f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

To calculate the Hessian numerically, we need to use approximations of partial derivatives - finite differences. They are denoted using the Δ character. Therefore, the Hessian matrix using said approximations would look like this:

$$H_f(x) = \begin{bmatrix} \frac{\Delta^2 f}{\Delta x_1^2} & \cdots & \frac{\Delta^2 f}{\Delta x_1 \Delta x_n} \\ \vdots & \ddots & \vdots \\ \frac{\Delta^2 f}{\Delta x_n \Delta x_1} & \cdots & \frac{\Delta^2 f}{\Delta x_n^2} \end{bmatrix}$$

Keeping the formula for central difference in mind, we can write the single finite difference over x_i as:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{\Delta f}{\Delta x_i}(x) = \frac{f(x + e_i h) - f(x - e_i h)}{2h} \tag{5.9}$$

Similarly, each entry of the Hessian matrix, being finite difference over x_i and x_j, would follow this formula:

$$\frac{\Delta^2 f}{\Delta x_i \Delta x_j}(x) = \frac{f(x + e_i h + e_j h) - f(x + e_i h - e_j h) - f(x - e_i h + e_j h) + f(x - e_i h - e_j h)}{4h^2} \tag{5.10}$$

It is worth noting that the order in which approximations of partial derivatives are applied is not important. If function $f \in C^2$ (it is at least twice-differentiable), using Schwartz theorem we can prove the following:

$$\frac{\Delta^2 f}{\Delta x_i \Delta x_j} = \frac{\Delta^2 f}{\Delta x_j \Delta x_i} \tag{5.11}$$

which also implies that:

$$H_f(x) = H_f^T(x) \tag{5.12}$$

Another thing to remember is that not from every function Hessian is easily obtainable (for example functions that do not have curvature). To ensure correct flow of work of the algorithm, a certain trick must be used. This trick comes down to adjusting Hessian of a function by adding to it diagonal matrix multiplied by small λ value. This change should be applied whenever determinant of $abs(H_f(x))$ is smaller that some arbitrary threshold t. Described method is often called **Ridge Regression** or **Levenberg-Marquardt adjustment**.

5.1.2 Newton-Raphson method

There is also another algorithm, similar to Newton descent, that is worth mentioning. It is called the Newton-Raphson method (and sometimes just Newton's method). This algorithm is a root-finding algorithm which produces successively better approximations to the roots (or zeroes) of a given function. The formula used by this algorithm is as follows:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{5.13}$$

Caution! This is a very similar formula to the formula used by Newton descent, but it differs in the degree of derivatives of the function f. The two mentioned algorithms are trying to solve two different problems. However, they share a common relation. The Newton-Raphson method finds the roots of the function that will be given to it. Assuming that a given function is already a derivative of another function (considered by the Newton descent algorithm), finding these roots basically gives us the formula for Newton descent and solves the problems it addresses.

5.1.3 Implementation of the Newton descent algorithm

Using programming language **R** we can write a Newton descent implementation. First of all, we need a function to calculate numerical Hessian matrix:

```r
num_hessian <- function(f, x, h = 10^-3){
  #' Numerical Hessian
  #'
  #' @description Function resposible for determining the numerical Hessian
  #' by calculating the matrix of partial derivatives.
  #'
  #' @param f function. The function which Hessian we determine
  #' @param x numerical vector. The point at which the numerical Hessian
  #' is determined
  #' @param h scalar. Finite difference value
  #'
  #' @usage num_hessian(f, x)
  #'
  #' @return Hessian matrix of partial derivatives of function f.

  n <- length(x)
  H <- matrix(NA, nrow = n, ncol = n)
  E <- diag(n)

  for(i in 1 : n) { # Rows
      for(j in 1 : n) { # Columns
          H[i, j] <- (
                f(x+E[i,]*h+E[j,]*h) - f(x+E[i,]*h-E[j,]*h)
                - f(x-E[i,]*h+E[j,]*h) + f(x-E[i,]*h-E[j,]*h)
            ) / (4*h^2)
      }
  }

  return(H)
}
```

Listing 1: Numerical Hessian implementation

The formula used in subsequent iterations of the Newton descent algorithm requires us to calculate the inverse of the Hessian matrix. Fortunately, language **R** provides a ready-made solution to this problem in the form of the *solve()* method.

```r
my_fun <- function(x) 1/8*x[1]^2+x[2]^2
x0 <- c(3, 4)
solve(num_hessian(my_fun, x0)) # Returns the inverse of the Hessian matrix
```

Listing 2: Hessian solve() method example

Having the ability to calculate the inverse of the matrix, we can finally create the code necessary for the Newton descent algorithm. Example of it's implementation can look like this:

```r
newton_descent <- function(f, x, K = 100){
  #' NEWTON DESCENT
  #'
  #' INPUT:
  #'       - f: target function
  #'       - x: starting point
  #'       - K: maximum iteration limit
  #'
  #' OUTPUT:
  #'       - x_opt: found solution
  #'       - f_opt: value of target function in the found solution
  #'       - x_hist: history of explored solutions
  #'      - f_hist: history of target function values
  #'      - t_eval: time elapsed during algorithm calculations

  start_time <- Sys.time()
  results <- list(x_opt = x,
                  f_opt = f(x),
                  x_hist = matrix(NA, nrow = K, ncol = length(x)),
                  f_hist = rep(NA, K),
                  t_eval = NA)

  results$x_hist[1,] <- x
  results$f_hist[1] <- f(x)

  for(k in 2:  K){
    # calculating gradient and hessian of a function f
    G <- grad(f, x)
    H <- num_hessian(f, x)

    # using Ridge Regression to help solve situations
    # where Hessian can not be calculated
    if(abs(det(H)) < 10^(-3)) H <- H + diag(n)*10^(-3)

    # description of the transition from point x_k to x_k+1
    x_new <- x - solve(H) %*% G

    # checking whether the new solution
    # is the best so far
    if(f(x_new) < results$f_opt){
      results$x_opt <- x_new
      results$f_opt <- f(x_new)
    }

    results$x_hist[k,] <- x_new
    results$f_hist[k] <- f(x_new)

    x <- x_new
  }
  # time difference between the end and start of the algorithm
  results$t_eval <- Sys.time() - start_time
  return(results)
}
```

Listing 3: Example Newton descent implementation

Example 2. Let $f(x_1, x_2) = \frac{1}{8} \cdot (x_1)^2 + (x_2)^2$ and $x_{start} = (3, 4)$.

How Newton descent converges can be seen on the plot below (alongside with paths created by other similar algorithms):

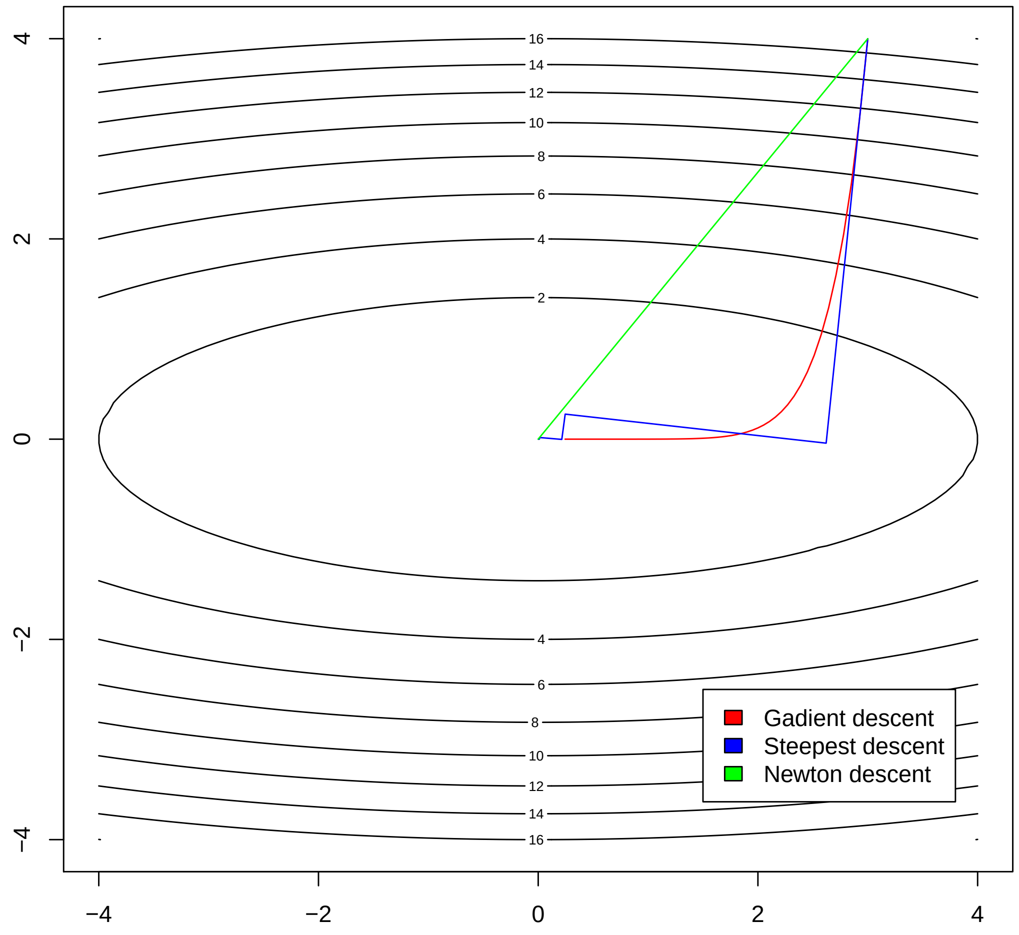

Figure 5.1: Comparison of the paths obtained by the Newton descent and other similar algorithms on function f in 2D space

We can see that Newton descent has a path that is significantly different from the other algorithms. Steepest descent and gradient descent algorithms are moving orthogonal with regards to counter-plot. Newton descent takes a much more direct route towards the global minimum of the function. It is no longer going towards locally best direction. This behavior is caused by the fact that the function f is quadratic.

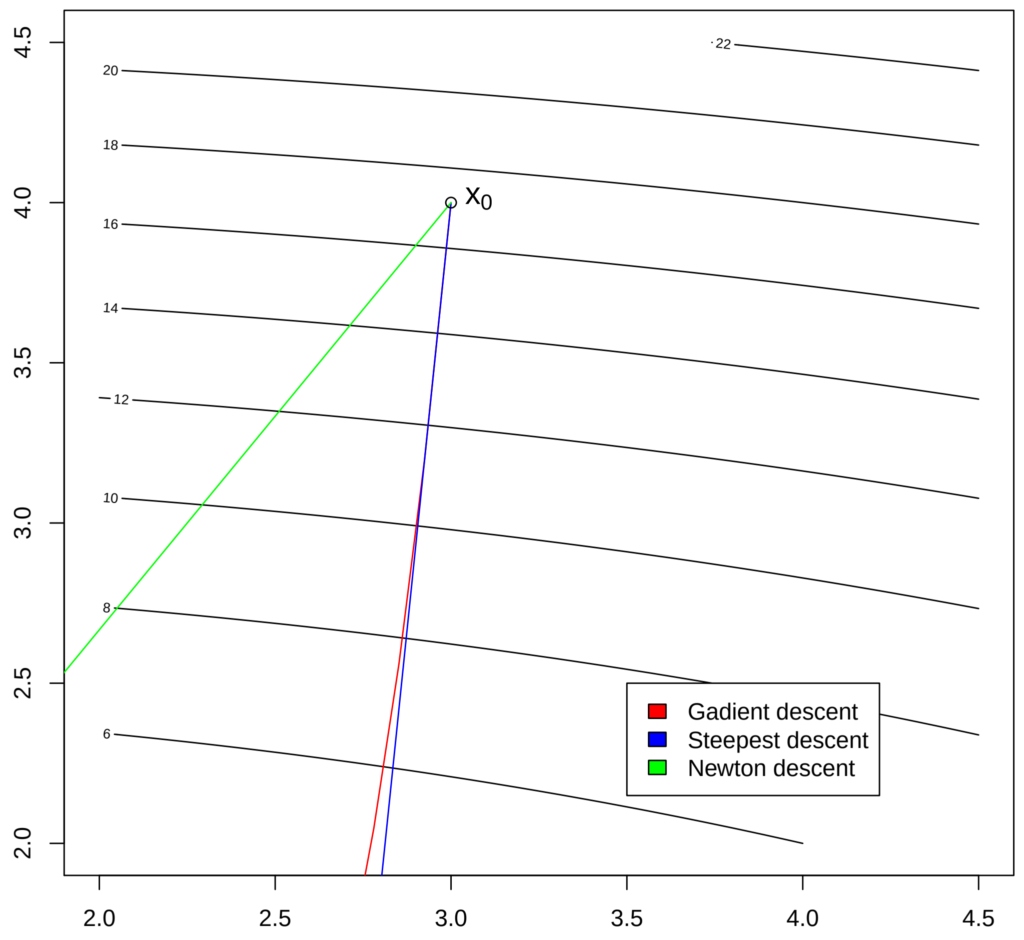

Figure 5.2: Closeup of the paths obtained by the Newton descent and other similar algorithms