Continuous optimization: Descent Methods

Maximilian Zebhauser

12 11 2018

Given is the following funciton with $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{m} \in \mathbb{R}^n$ as a fixed vector and $A \in \mathbb{S}^n_{++}$ as a fixed positive definite matrix:

$$f(x) = \frac{1}{2}(x-m)^\top A(x-m) - \sum_{i=1}^{n} \log\left(x_i^2\right)$$

Implementations

Function:

```
given_function <- function(x,A,m) (0.5 * t((x - m)) %*% A %*% (x - m) - sum(log(x^2)))
given_gradient <- function(x,A,m) A %*% (x - m) - 2/(x)
given_hessian <- function(x,A,m) A + diag(diag( 2 / (x %*% t(x))))
```

Gradient descent function:

```
gradient_descent <- function(x_start, A, m, precis = 0.01, iter = 10000){
    i <- 1; step_length <- precis + 1; step_size <- 1
    trace <- list(list("x" = x_start, "f_x" = given_function(x_start,A,m)))
    while (precis < step_length) {
        gradient <- given_gradient(trace[[i]][["x"]],A,m)
        x_next <- trace[[i]][["x"]] - step_size * gradient
        step_length <- sqrt(sum(gradient^2))
        i = i + 1
        trace[i] <- list(list("x" = x_next,"f_x" = given_function(x_next,A,m)))
        if (trace[[i-1]][["f_x"]] < trace[[i]][["f_x"]]){
            i = i - 1
            step_size <- step_size/2
        }
        if (i == iter) break
    }
    trace
}
```

Coordinate descent function, as proposed by Stephen J. Wright (https://arxiv.org/pdf/1502.04759.pdf):

```
coord_descent <- function(x_start, A, m, precis = 0.01, iter = 10000){
    i <- 1; step_length <- precis + 1; step_size <- 1
    trace <- list(list("x" = x_start, "f_x" = given_function(x_start,A,m)))
    space <- length(trace[[i]][["x"]])
    k <- rep(1:space,iter/space)
    while (precis < step_length) {
        gradient <- given_gradient(trace[[i]][["x"]],A,m)
        basis_vector <- rep(0,space)
        basis_vector[k[i]] <- gradient[k[i]]
        x_next <- trace[[i]][["x"]] - step_size * basis_vector
        step_length <- sqrt(sum(gradient^2))
        i = i + 1
        trace[i] <- list(list("x" = x_next,"f_x" = given_function(x_next,A,m)))
        if (trace[[i-1]][["f_x"]] < trace[[i]][["f_x"]]){
            i = i - 1
            step_size <- step_size/2
        }
        if (i == iter) break
    }
    trace
}
```

Newton's method:

```
newton_descent <- function(x_start, A, m, precis = 0.01, iter = 10000){
    i <- 1; step_length <- precis + 1; step_size <- 1
    trace <- list(list("x" = x_start, "f_x" = given_function(x_start,A,m)))
    while (precis < step_length) {
        gradient <- given_gradient(trace[[i]][["x"]],A,m)
        hessian <- given_hessian(trace[[i]][["x"]], A, m)
        x_next <- trace[[i]][["x"]] - step_size * (solve(hessian) %*% gradient)
        step_length <- sqrt(sum(gradient^2))
        i = i + 1
        trace[i] <- list(list("x" = x_next,"f_x" = given_function(x_next,A,m)))
        if (trace[[i-1]][["f_x"]] < trace[[i]][["f_x"]]){
            i = i - 1
            step_size <- step_size/2
        }
        if (i == iter) break
    }
    trace
}
```

Convergence with variation in A for the two-dimensional case

$\mathfrak{m} = (0.5, 0)$ and $A = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$ where $\rho \in (-1, 1)$.

Having $\rho = |1|$ means, that one of the eigenvalues is 0 and A is positive semi-definite. This indicates that the function has a linear subspace rather than a minimum. This special case is excluded from any further investigation. All three approaches use the basic backtracking method and have the same precision threshold. To compare the performance of the three methods I will compare the number of iterations needed. I will average the number of iterations needed for 100 different random starting points. Starting points will be constant and only ρ will be varyied. The outcomes of the analysis is seen in Table 1.

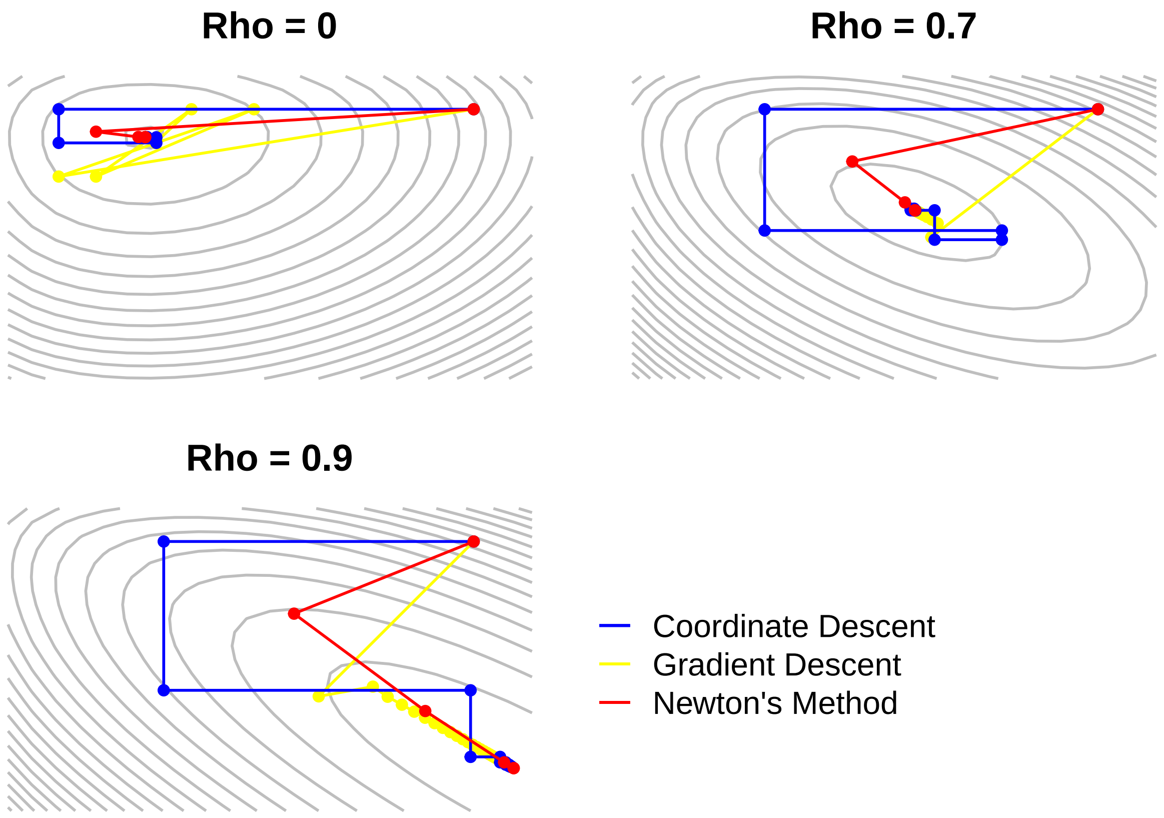
The Newton Method requires on average between 6 to 7 iterations no matter what values the matrix A consists of. It is the most stable approach and has the best performance for all cases. In the cases, where the function is less elipsoid shaped (lower ratio of the two eigenvalues) the gradient descent method performs better than the coordinate-wise descent. It is clear that in the extreme cases the gradient descent method struggles.

Table 1: Performance as average number of iterations

Rho	Gradient Descent	Coordinate Descent	Newton's Method	Eigenvalue ratio
-0.9	100.23	28.76	6.25	19.00
-0.6	26.18	34.36	6.78	4.00
-0.3	10.80	15.18	6.94	1.86
0.0	15.66	24.04	6.99	1.00
0.3	12.65	18.67	6.73	1.86
0.6	26.01	19.63	6.57	4.00
0.9	120.01	25.95	6.14	19.00

Graphical analysis of the two-dimensional case

Let's have a look at three cases graphically. First I want to explore the case where the gradient descent method performs well (e.g. $\rho = 0$). Then I explore two cases where the function is more elipsoid shaped (e.g. $\rho = 0.7$ and $\rho = 0.9$). For the graphical analysis the start values are picked manually. This allows to "zoom in" and we can focus the analysis on one quadrant. This is done with no loss of generality, as the function behaves almost the same as $\rho = 0$ in the two quadrants I and III while varying positive ρ. The shape changes in the II and IV quandrant mirrowed around the origin. The behaviour is vice versa regarding the quadrants for negative ρ. To observe the convergence of the different methods in a higher detail, I will focus on the IV quadrant and therefore pick a start value in this quadrant: $x = (4.5, -1)$.



All methods show a path as expected. The gradient decent heads always in the direction of the gradient solely and in these special cases converges for increasing ρ within the following number of iterations: 9, 13 and 37. The coordinate-wise descent is always heading in direction of the partial derivative and shows a zig-zag path when converging to the minimum. It performs simliar as the gradient descent, but has lower number of iterations when the functions is more elipsoid shaped: 9, 13 and 14. The Newton method performs best with respect to number of iterations and finds its path by taking the the gradient and the hessian into account. This means compared to the gradient descent method it respects the curvature of the function. Following iterations needed for the seen three cases: 5, 5 and 6.

Higher dimensional cases

In this step I will compare how the methods perform if a higher the number of dimensions is observed, e.g. $\mathbb{R}^n$ with $n \in [100, 1000]$. This time I average over 25 random starting points for computational reasons. Matrix A and vector m are randomly simulated, which is also true for the starting points. The results are seen in Table 2 and Table 3 below.

Table 2: Performance as average number of iterations

Dimension	Gradient Descent	Coordinate Descent	Newton's Method
50	23.68	3140.68	7.16
250	563.64	10000.00	10.48
500	2816.12	10000.00	11.64
750	5773.20	10000.00	12.52
1000	6418.48	10000.00	13.64

Table 3: Performance as running time

Dimension	Gradient Descent	Coordinate Descent	Newton's Method
50	0.02	2.57	0.05
250	2.84	52.60	4.49
500	48.30	174.31	34.55
750	226.36	395.75	122.83
1000	434.09	718.75	324.76

In higher dimensions the coordinate descent method doesn't converge within the iteration threshold. This also the reason for the high running time. The convergence performance of the gradient descent method gets worse with increasing number of dimensions, but within the given range of dimensions the method converges. The Newton's method is by far the most performant method considering the number of iterations. When it comes to running time the Newton's method only slightly outperforms the gradient descent method. The reason for this are the two additional calculation steps, e.g. the calculation of the hessian and the calculation of the inverse of the hessian.