Assignment in Optimization 2018 - Quasi-Newton methods

John Rapp Farnes, Erik Ris
Axel Sjöberg
971126-4771, 940309-8677
960314-3653
jo5113fa-s@student.lu.se, ine15erik@student.lu.se
ax3817sj-s@student.lu.se

January 30, 2019

Contents

1 Introduction

The purpose of this project was to implement and compare two different kinds of Quasi-Newton methods: BFGS (Broyden–Fletcher–Goldfarb–Shanno) and DFP (Davidon–Fletcher–Powell) for minimizing functions. The different methods are used to solve different optimization problems, including constained optimization via penalty- and barrier-functions. The project was based upon the course FMAN60 and the litterature used in the course.

2 Method

The basic idea behind both optimization algorithms is to perform iterations of the following routine:

1. Start at a point x_k, where x_0 is a given initial point
2. Determine a search direction d_k
3. Minimize $F(\lambda) = f(x_k + \lambda d_k)$ along d_k using a line search algorithm. Let $\lambda^* =$ optimum
4. Let $x_{k+1} = x_k + \lambda^* d_k$
5. Check x_{k+1} against the termination criterion

This algorithm was implemented in MatLab as *nonlinearmin*, with *linesearch* as a subroutine, with the following function signatures:

$$[lambda, No\_of\_iterations] = linesearch(func, x, d)$$
$$[x, No\_of\_iterations] = nonlinearmin(f, start, method, tol, printout)$$

2.1 linesearch

The Golden Section method was first implemented as the line search method but it was later discarded in favour of Armijo's rule. Armijo's rule is an inexact line search method that specifies two criteria that a λ solving the line search optimization problem should satisfy, based on two parameters $\alpha > 1$ and $0 < \varepsilon < 1$. The algorithm starts with an initial $\lambda = \lambda_0$ and iteratively checks the λ against the criteria of Armijo's rule. If the criteria that the λ is to large, it is divided by α, and multiplied by α if it is too small. The criteria are based on "overshooting" the tangent at $\lambda = 0$. For a further explanation of Armijo's rule see the course literature, and the schematic overview in figure 1.

The chosen parameters in the implementation were $\alpha = 2$, $\varepsilon = 0.1$ and $\lambda_0 = 1$.

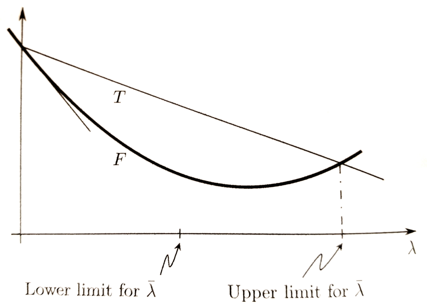

T
F
λ
Lower limit for $\bar{\lambda}$
Upper limit for $\bar{\lambda}$

Figure 1: Schematic overview of Armijo's rule, where T is the "overshooting" tangent at $\lambda = 0$

2.2 nonlinearmin

Both of the examined methods determine the direction d_k using the gradient of f as well as conjugate directions. The difference between the methods concerned the formula for calculating the matrix D, later used for determining $d_k = -D \cdot \nabla f$. The methods were implemented according to the course literature.

3 Optimization problems

The function *nonlinearmin* was used to solve four different optimization problems, referred to as (1), (2), (3), (4) below.

Problem (2) and (3) were solved using the penalty method which was implemented in MatLab, using the sequence $\mu = 0.01, 0.1, 1, 10, 100$. Problem (4) was solved with the barrier method with $\varepsilon = 1, 0.1, 0.01, 0.001, 0.0001$.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(\bar{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\ & \text{(Rosenbrock's function)} \end{aligned} \tag{1}$$

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(\bar{x}) = e^{x_1 x_2 x_3 x_4 x_5} \\ \text{subject to} \quad & x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 = 10 \\ & x_2 x_3 = 5 x_4 x_5 \\ & x_1^3 + x_3^3 = -1 \end{aligned} \tag{2}$$

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(\bar{x}) = e^{x_1} + x_1^2 + x_1 x_2 \\ \text{subject to} \quad & \frac{1}{2} x_1 + x_2 - 1 = 0 \end{aligned} \tag{3}$$

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(\bar{x}) = (x_1 - 5)^2 + (x_2 - 3)^2 \\ \text{subject to} \quad & x_1 + x_2 \leq 3 \\ & -x_1 + 2x_2 \leq 4 \end{aligned} \tag{4}$$

4 Results

4.1 Optimization problems

Table 1 and 2 shows the results from the numerical optimization of the four problems, from the starting points given in the assignment.

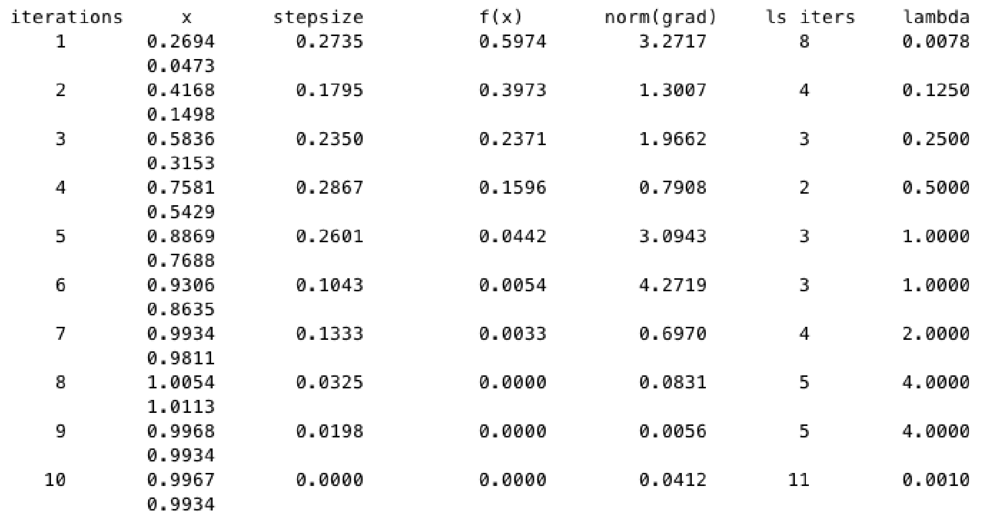
Problem	Optimal point x^*	$f(x^*)$	Iterations
Rosenbrock	$(0.9974, 0.9947)$	$6.9089 \cdot 10^{-6}$	37
Penalty problem	$(-1.7170, 1.8275, 1.5956, -0.7637, -0.7637)$	0.0539	13
Exercise 9.3	$(-1.2785, 1.6393)$	-0.1828	17
Exercise 9.5	$(2.4977, 0.4979)$	12.5224	19

Table 1: Results from *nonlinearmin* using DFP

Problem	Optimal point x^*	$f(x^*)$	Iterations
Rosenbrock	$(0.9974, 0.9948)$	$6.6996 \cdot 10^{-6}$	37
Penalty problem	$(-1.7165, 1.8284, 1.5950, -0.7637, -0.7637)$	0.0539	13
Exercise 9.3	$(-1.2785, 1.6393)$	-0.1828	14
Exercise 9.5	$(2.4975, 0.4980)$	12.5224	19

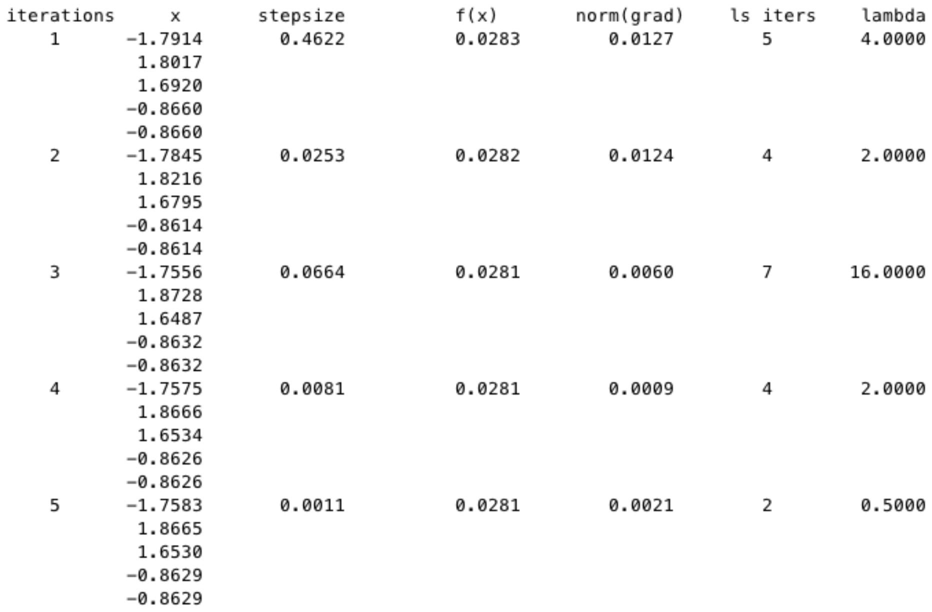
Table 2: Results from *nonlinearmin* using BFGS

4.2 Example of printouts



```
iterations      x       stepsize          f(x)      norm(grad)    ls iters    lambda
     1       0.2694       0.2735         0.5974        3.2717         8       0.0078
             0.0473
     2       0.4168       0.1795         0.3973        1.3007         4       0.1250
             0.1498
     3       0.5836       0.2350         0.2371        1.9662         3       0.2500
             0.3153
     4       0.7581       0.2867         0.1596        0.7908         2       0.5000
             0.5429
     5       0.8869       0.2601         0.0442        3.0943         3       1.0000
             0.7688
     6       0.9306       0.1043         0.0054        4.2719         3       1.0000
             0.8635
     7       0.9934       0.1333         0.0033        0.6970         4       2.0000
             0.9811
     8       1.0054       0.0325         0.0000        0.0831         5       4.0000
             1.0113
     9       0.9968       0.0198         0.0000        0.0056         5       4.0000
             0.9934
    10       0.9967       0.0000         0.0000        0.0412        11       0.0010
             0.9934
```

Figure 2: Print out from nonlinearmin.m, f = rosenbrock.m, method = ”BFGS”



```
iterations      x         stepsize          f(x)        norm(grad)    ls iters      lambda
     1       -1.7914       0.4622          0.0283         0.0127         5         4.0000
              1.8017
              1.6920
             -0.8660
             -0.8660
     2       -1.7845       0.0253          0.0282         0.0124         4         2.0000
              1.8216
              1.6795
             -0.8614
             -0.8614
     3       -1.7556       0.0664          0.0281         0.0060         7        16.0000
              1.8728
              1.6487
             -0.8632
             -0.8632
     4       -1.7575       0.0081          0.0281         0.0009         4         2.0000
              1.8666
              1.6534
             -0.8626
             -0.8626
     5       -1.7583       0.0011          0.0281         0.0021         2         0.5000
              1.8665
              1.6530
             -0.8629
             -0.8629
```

Figure 3: Print out from nonlinearmin.m, f = (2) + penalty func., method = "DFP", $\mu = 0.01$

4.3 Comparison of different initial point and methods

Method	Intial point	Optimal point x^*	$f(x^*)$	Iterations
DFP	$(0, 2, 2, -1, -1)$	$(-1.7171, 1.8274, 1.5956, -0.7637, -0.7637)$	0.0539	13
DFP	$(-1, 2, 2, -1, -1)$	$(-1.7170, 1.8275, 1.5956, -0.7637, -0.7637)$	0.0539	15
BFGS	$(0, 2, 2, -1, -1)$	$(-1.7180, 1.8256, 1.5967, -0.7636, -0.7636)$	0.0539	12
BFGS	$(-2, 2, 2, -1, -1)$	$(-1.7165, 1.8284, 1.5950, -0.7637, -0.7637)$	0.0539	13

Table 3: Comparison of results from *nonlinearmin* for problem (2) with different initial points and methods

5 Discussion

5.1 Line search

5.1.1 Choice of line search algorithm

At first, the Golden Section method was chosen as the line search method because of its reliable convergence properties. The problem with the method however was that in order to find optimum located far from the initial point, the initial interval had to be very large. This led to computations that did not finish in a reasonable amount of time. In order to obtain flexibility regarding the location of the optimum, while not sacrificing performance, Armijo's rule was chosen instead.

5.1.2 Numerically calculating $F'(0)$

Armijo's rule requires the derivative to be computed at $\lambda = 0$. This derivative must be calculated numerically, approximating the limit in the definition of the derivative, in order to achieve reasonable performance. The problem with this approach is to determine which h to use in the limit. For some functions, such as the *test_func* in the assignment, a h that is very small ($\approx 10^{-60}$) must be used in order to avoid numerical errors (which will result in an *Inf* or *NaN* derivative). The problem with such a small h is that it numerically "kills" functions that are less steep, resulting in a derivative of 0. As such, h must be chosen dynamically depending on the local behavior of the function.

At first, this problem was solved by starting with a small h and increasing it as long as the resulting derivative was 0. This approach however proved to be either slow (if too many different values of h were tried) or unstable (if too few values were tried). After consulting other groups having the same problem, we implemented a solution where first a $\lambda = \lambda'$ is found that has the property that $F(\lambda') \approx F(0)$. This λ' was found by starting with $\lambda = \lambda_0$ and dividing λ with α as long as $F(\lambda) > F(0)$, or multiplying it with α as long as $F(\lambda) < F(0)$. The resulting λ' was then used in calculating h, with $h = h_0\lambda'$ where h_0 is a constant ($h_0 = 10^{-6}$ in the implementation). This gives a good value of h since λ' is a measure of the steepness of the function at $\lambda = 0$. If it is steep, the λ' is small resulting in a small h (required for calculating the derivative) and the opposite if the function is flat (not "killing" the derivative). This method is implemented as *find_suitable_lambda*, which returns λ'. This λ' is then used as λ_0 in Armijo's rule.

In addition, we added a condition where we set the derivative to 0 if it is calculated to be a positive number, in order to terminate earlier from Armijo's rule.

5.2 Termination criterion

For the termination criterion, we combined criteria regarding the changes in $\bar{x}_k$ and $f(\bar{x})$ using (5) and (6), where ϵ is the tolerance.

$$|\bar{x}_{k+1} - \bar{x}_k| \leq \epsilon \tag{5}$$

$$|f(\bar{x}_{k+1}) - f(\bar{x}_k)| < \epsilon \tag{6}$$

The algorithm was terminated if any of the above conditions were met. The motivation behind doing this is that the criterion then considers both when changes in $\bar{x}$ is declining and $f(\bar{x})$ is. The inequalities and equalities in (5) and (6) have been taken from the course literature.

This termination criterion was not only applied to outer loop of the *nonlinearmin* but also the inner loop, as changes that were too small during the inner iterations may result in elements of D that were *NaN*, since $q = 0$ if $y_{k+1} - y_k$ was too small. In addition, the algorithm terminated if $\nabla f < \epsilon$ as close to zero gradients may have resulted in zero directions and thereby line search optimization problems that had no solution.

5.3 Performance on problems (1) - (4)

5.3.1 Rosenbrock (1)

Depending on the initial point, sometimes the DFP method outperformed the BFGS method and sometimes the BFGS outperformed the DFP. For both the DFP- and the BFGS method, more iterations were required as the initial point was further away from the optimal minimum point. Choosing an initial point that was very far away form the optimal point sometimes meant that the problem converged to a local minimum that was not the global minimum and sometimes lead to errors in the line search. This was the case for both the DFP- and BFGS method. For instance setting the initial point to $\bar{x} = (10000, 50000)$ and using the DFP method, the problem converged to the optimal point $\bar{x} = (-1100, 1.3282)$, which gave the minimum value $1.3305 \cdot 10^6$. When testing initial points that were close to the global min, the linesearch always worked and always converged towards the same optimal solution. In order to make the algorithm converge to the global minimum, several different initial points that are far away from each other should be tested. If only one initial point is tested, the algorithm could converge towards a local minimum that is not a global minimum, which was the case above.

5.3.2 Penalty function problems (2), (3)

The minimization of (2) gave the same minimal value for DFP and BFGS, however the minimal value differed depending on the initial points. Depending on the initial point the BFGS algorithm and DFP algorithm required a different amount of iterations before terminating the optimization due to the termination criterion. For all the tested initial points the BFGS method required less than or the equal amount iterations as the DFP method. These points were however arbitrarily chosen and this may not hold in general.

For minimization problem (2) there was no apparent relation between the required number of iterations and the distance from the initial point and the optimal point, as can be seen in table 3 above. For both the DFP method and BFGS method there existed an initial point with greater distance to the optimal point than another initial point, at the same time as it required fewer iterations before terminating the optimization.

The minimization problem (3) converged to the same solution for all the tested initial points, except for initial points that were far away from the optimal point. The initial points far away from the optimal solution generated an error in the linesearch as it required too many iterations in the linesearch. As with minimization problem (2), depending on the initial point, the BFGS algorithm and DFP algorithm required a different amount of iterations before terminating the optimization. Setting different initial points, the BFGS method always required less than or the equal amount of iterations as the DFP method required. None the less, the initial points were still chosen arbitrarily as above and analogously we cannot determine if the BFGS method requires less than or the equal amount of iterations as the DFP method for all initial points.

Choosing an initial point that is very far away from the optimal point resulted in an error in the lineserch as it required too many iterations in the linesearch. Setting an initial point somewhat close to the optimal value however always meant that the problem converged to same solution for (2) and (3). Choosing a suitable initial point proved to be quite the difficult task. No one size fits all method that works in all cases were apparent. The most successful strategy was choosing plenty of initial points that were far away from each other and compare the solutions to each other. A few different points close to each other were tried out to test the consistency of the program behaviour.

5.3.3 Barrier Function Problem (4)

For the initial point $\bar{x} = (0, 0)$ the BFGS and DFP converged to the same solution and the optimization was terminated in the same number of iterations. Changing the initial point to $\bar{x} = (0, -10)$ gave the same minimimum value, however it only required 14 iterations for the DFP method and 17 for the BFGS method. For all feasible points, the DFP required less than or the equal amount of iterations as the BFGS method. As with minimization problem (2) and (3) the initial points were chosen arbitrarily and we cannot determine whether or not the DFP method for all initial points require less or the equal amount of iterations as the BFGS method, in problem (4).

For all the feasible points tested, the barrrier function converged to the same solution. The strategy concerning choosing a suitable initial point in this problem is thereby reduced to finding feasible points.

5.4 Comparison of DFP and BFGS

For penalty problems as problem (2) and (3) the BFGS method performed better than the DFP method in the sense that it required less iterations for the same minimal value. For the barrier problem (4) the DFP performed better than the BFGS in the same regard. As such, we are not able to draw a clear conclusion regarding which if DFP or BFGS is ”better”. As only two barrier functions with a few different initial points and one barrier function with a few different initial points were tested, there is a lack of foundation to build strong conclusions. For the Rosenbrock function depending on the initial point whichever of the DFP and BFGS method outperformed the other method varied.

Appendix - Matlab implementation

linesearch.m

```
function [lambda, No_of_iterations] = linesearch(func, x, d)

    F = @(lambda) func(x + lambda .* d);

    [lambda, No_of_iterations] = armijo(F);

    if isnan(func(x+lambda*d))
        error('Line search is NaN!')
    elseif func(x+lambda*d) > func(x)
        error('Line search did no good!')
    end
end

function [lambda, No_of_iterations] = armijo(F)

    alpha = 2;
    epsilon = 0.1;
    lambda_0 = 1;

    max_iterations = 500;

    F_0 = F(0);
    [lambda, No_of_iterations] = find_suitable_lambda(F, lambda_0, F_0, alpha, ...
        max_iterations);

    derivative_0 = derivative(F, 0, lambda);

    T = @(lambda) F_0 + epsilon*lambda*derivative_0;

    while No_of_iterations < max_iterations
        No_of_iterations = No_of_iterations + 1;

        below_tangent = F(lambda) <= T(lambda);
        too_small = F(alpha*lambda) <= T(alpha*lambda);

        if below_tangent && not(too_small)
            break;
        end

        if not(below_tangent)
            lambda = lambda / alpha;
        elseif too_small
            lambda = lambda * alpha;
        end
    end

    if No_of_iterations == max_iterations
        error('Too long time in line search!')
    end
end

function [lambda, No_of_iterations] = find_suitable_lambda(F, lambda, F_0, ...
    alpha, max_iterations)
    No_of_iterations = 0;

    was_bigger = F(lambda) > F_0;

    while (isnan(F(lambda)) || F(lambda) > F_0) && No_of_iterations < ...
        max_iterations
       lambda = lambda / alpha;
       No_of_iterations = No_of_iterations + 1;
    end

    if was_bigger
        return;
    end

    while F(lambda) <= F_0 && No_of_iterations < max_iterations
        lambda = lambda * alpha;
        No_of_iterations = No_of_iterations + 1;
    end
end

function dy = derivative(F, x, suitable_lambda)
    h_0 = 1e-6;
    h = suitable_lambda * h_0;

    dy = (F(x + h) - F(x - h))/(2*h);

    if (dy > 0)
        dy = 0;
    end
end
```

nonlinearmin.m

```
function [x, No_of_iterations] = nonlinearmin(f, start, method, tol, printout)

    update_matrix = update_matrix_fn(method);

    max_iterations = 500;
    n = numel(start);
    x = start;
    No_of_iterations = 0;

    print_header(printout);

    while No_of_iterations < max_iterations
        No_of_iterations = No_of_iterations + 1;

        y = x;
        D = eye(n);
        for j = 1:n
            gf = grad(f, y);

            if grad_too_small(gf, tol)
                x = y;
                return;
            end

            d = -D * gf;
            [lambda, ls_iter] = linesearch(f, y, d);
            next_y = y + lambda*d;

            if should_stop(f, y, next_y, tol)
                break;
            end

            D = update_matrix(f, D, lambda, d, y, next_y);
            y = next_y;
        end
        x_next = next_y;

        print_row(printout, No_of_iterations, x, x_next, f, gf, ls_iter, lambda);

        if should_stop(f, x, x_next, tol)
            break;
        end

        x = x_next;
        if No_of_iterations == max_iterations
            error('Too long time in minimization!')
        end
    end

    x = x_next;
    if isnan(f(x)) || f(x) > f(start)
        error('Bad job of the unlinear search!')
    end

end

function update_matrix = update_matrix_fn(method)
    if method == "DFP"
```

```
        update_matrix = @update_DFP;
    elseif method == "BFGS"
        update_matrix = @update_BFGS;
    end
end

function [p, q] = get_pq(f, lambda, d, y, next_y)
    p = lambda * d;
    q = grad(f, next_y) - grad(f, y);

    if all(p == 0)
        error("p == 0");
    end

    if all(q == 0)
        error("q == 0");
    end
end
function D = update_DFP(f, D, lambda, d, y, next_y)
    [p, q] = get_pq(f, lambda, d, y, next_y);
    D = D + (p*p')/(p'*q) - (D*(q*q')*D)/(q'*D*q);
end
function D = update_BFGS(f, D, lambda, d, y, next_y)
    [p, q] = get_pq(f, lambda, d, y, next_y);
    D = D + (1 + q'*D*q/(p'*q))*(p*p')/(p'*q)-(p*q'*D+D*q*p')/(p'*q);
end
function stop = grad_too_small(gf, tol)
    stop = norm(gf) < tol;
    if stop
        disp("Gradient too small!");
    end
end
function stop = should_stop(f, x, x_next, tol)
    stop = all(abs(x_next - x) ≤ tol) || abs(f(x_next) - f(x)) < tol;
end

function print_header(printout)
    if (printout)
        disp("iterations     x       stepsize          f(x)       norm(grad) ...
                ls iters    lambda");
    end
end

function print_row(printout, No_of_iterations, x, x_next, f, gf, ls_iter, lambda)
    if (printout)
        step_size = norm(x_next - x);
        fprintf('%5.0f %12.4f %12.4f %15.4f %13.4f %8.0f %13.4f\n', ...
            No_of_iterations, ...
            x_next(1), ...
            step_size, ...
            f(x_next), ...
            norm(gf), ...
            ls_iter, ...
            lambda ...
        );
        for k = 2:numel(x_next)
          fprintf('%18.4f\n', x_next(k));
        end
    end
end
```

penalty.m

```
function penalty = penalty(f, g_k, h_k, mu)

    function q = q(x)
        q = f(x) + mu * alpha(g_k, h_k, x);
    end
    penalty = @q;
end

function val = values(g_k, x)
    val(1) = 0;
    for i = 1:numel(g_k)
        g = g_k{i};
        val(i) = g(x);
    end
end

function alpha = alpha(g_k, h_k, x)
    alpha_g = max(values(g_k, x), 0).^2;
    alpha_h = values(h_k, x).^2;

    alpha = sum(alpha_g) + sum(alpha_h);
end
```

barrier.m

```
function barrier = barrier(f, g_k, epsilon)

    function q = q(x)
        q = f(x) + epsilon * beta(g_k, x);
    end

    barrier = @q;
end

function val = values(g_k, x)
    val(1) = 0;
    for i = 1:numel(g_k)
        g = g_k{i};
        val(i) = g(x);
    end
end

function beta = beta(g_k, x)
    val = values(g_k, x);

    not_feasible = any(val ≥ 0);

    if not_feasible
        beta = 1e200;
    else
        val = 1./val;
        beta = -sum(val);
    end
end
```