HOMEWORK 7

Mathieu Gaillard

March 4, 2019

Problem 0: Homework checklist

I mainly used the class material, Google, Wikipedia and my previous knowledge. I also asked a question on Piazza.

Problem 1

Question 1:

Implement a backtracking line search routine to select the step length and add this to our simple gradient descent code from the lectures.

Following is the backtracking function that is later used by a script.

```
using Printf, LinearAlgebra, ApproxFun

function backtracking_gradient_descent(fgh,x0;
    maxiter=10000,tol=1.0e-8,quiet=false,hessian=false,histx=[],hista=[],mu=0.1)

    x = copy(x0)
    n = length(x)

    hist = zeros(2,maxiter)
    savehistx = eltype(histx) == Vector{Float64} ? true : false
    savehista = eltype(hista) == Float64 ? true : false

    f = Inf
    normg = Inf
    lastiter = 0
    g = Vector{Float64}()
    h = Matrix{Float64}(undef, 0, 0)

    if !quiet
        @printf("  %6s  %9s %9s %9s\n", "iter", "val", "normg", "fdiff");
    end

    for iter=1:maxiter
        if savehistx
            push!(histx, x);
        end

        if iter>1
            # Choose the search direction
            if hessian
                p = -h\g; # Newton
            else
                p = -g; # Gradient descent
            end

            # Choose alpha with the backtracking method
            alpha = 1.0;
            while (fgh(x + alpha*p)[1] > f + mu*alpha*p'*g)
                alpha /= 2;
            end

            # Step towards the descent direction
            x = x + alpha*p;

            # Save the value of alpha in the history
            if savehista
                push!(hista, alpha);
            end
        end

        flast = f
```

```
        f,g,h = fgh(x)
        normg = norm(g,Inf)

        fdiff = flast - f

        if !quiet
            @printf("  %6i  %9.2e %9.2e %9.2e\n",
                iter, f, normg, fdiff)
        end

        hist[:,iter] = [f; normg]
        lastiter = iter

        if normg <= tol
            break
        end
        if !isfinite(normg)
            break
        end
    end

    if lastiter < maxiter
        hist = hist[:,1:lastiter]
    end

    if normg > tol
        @warn "Did not converge"
    end

    return x,f,g,hist
end
```

Following is a script to use the backtracking function.

```
using OptimTestProblems;
using Plots;

plotly(size = (800, 600));

include("simple_gradient_descent.jl");

# The function we want to optimize
optim_function = MultivariateProblems.UnconstrainedProblems.examples["Rosenbrock"];

# To easily diplay a contour plot
ezcontour(x, y, f) = begin
    X = repeat(x', length(y), 1)
    Y = repeat(y, 1, length(x))
    # Evaluate each f(x, y)
    Z = map((x,y) -> f([x,y]), X, Y)
    plot(x, y, Z, st=:contour)
end

function opt_combine(x, f, g!, h!)
    gradient = Vector{Float64}(undef, length(x));
    hessian = Matrix{Float64}(undef, length(x), length(x));
    g!(gradient, x);
    h!(hessian, x);
    return (f(x), gradient, hessian);
end
function opt_problem(p)
    return x -> opt_combine(x, p.f, p.g!, p.h!);
end

# Here's an example
fgh = opt_problem(optim_function);

histx = Vector{Vector{Float64}}();
hista = Vector{Float64}();
# x,fx,gx,hist = backtracking_gradient_descent(fgh, [1.2,1.2];
#                   maxiter=16000, quiet=true, hessian=false,
#                   histx=histx, hista=hista, mu=0.01);
x,fx,gx,hist = backtracking_gradient_descent(fgh, [1.2,1.2];
                  maxiter=16000, quiet=true, hessian=true,
                  histx=histx, hista=hista, mu=0.01);

ezcontour(-1.5:0.05:1.5, -1.5:0.05:1.5, optim_function.f);
plot!(map(first,histx),map(x->x[2], histx), linewidth=2);

plot(hista,linewidth=2,title="Step lengths");
```

Here is how the script behave on the Rosenbrock function. Figure 1 shows the path taken using gradient descent and backtracking starting from $(1.2, 1.2)$. Figure 2 shows the path taken using Newton direction and backtracking starting from $(1.2, 1.2)$. Figure 3 shows the path taken using gradient descent and backtracking starting from $(-1.2, 1.0)$. Figure 4 shows the path taken using Newton direction and backtracking starting from $(-1.2, 1.0)$. With both starting points, a maximum of 16,000 iterations and $c_1 = 0.01$, only the method using Newton direction converges.

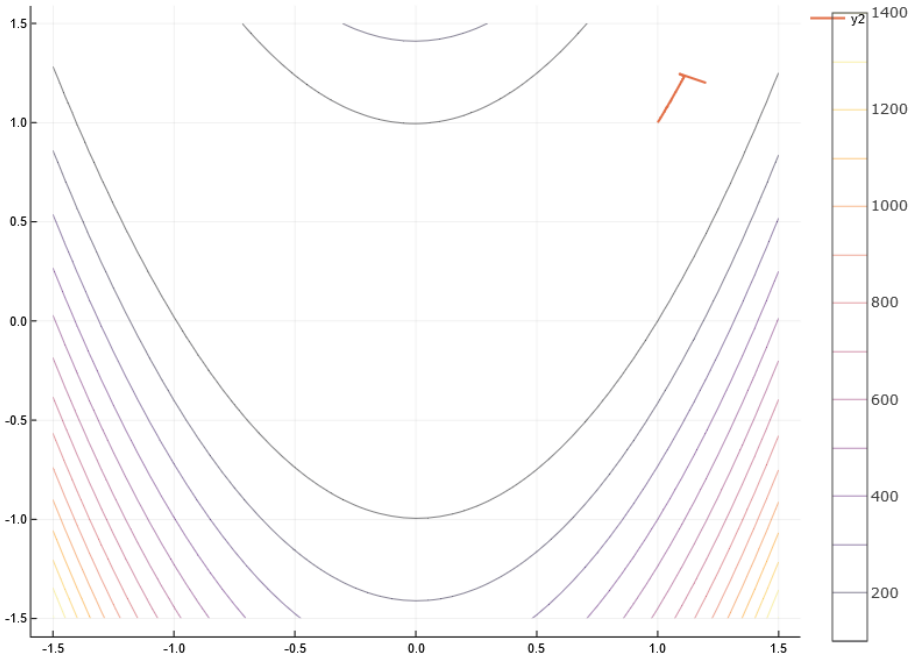

Figure 1: Rosenbrock function starting from $(1.2, 1.2)$ using Gradient Descent and backtracking.

Question 2:

Prepare a plot of the step lengths (values of α_k that were accepted) when we run gradient descent with this line search on the Rosenbrock function staring from the point $(1.2, 1.2)$ and also the point $(-1.2, 1)$.

Step lengths with gradient descent and backtracking are in Figure 5 and 6.

Question 3:

Implement Newton's method using line search as well: just use the search direction $\boldsymbol{H}(\mathbf{x}_k)\mathbf{p} = -\mathbf{g}_k$.

In the code, we add this line of code:

```
p = -h\g; # Newton direction
```

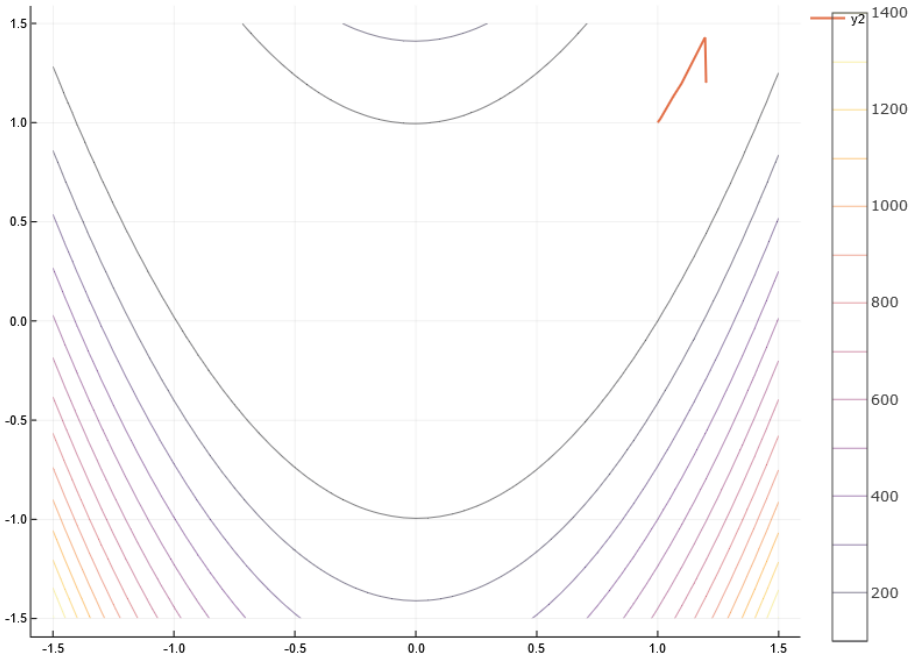

Figure 2: Rosenbrock function starting from $(1.2, 1.2)$ using Newton direction and backtracking.

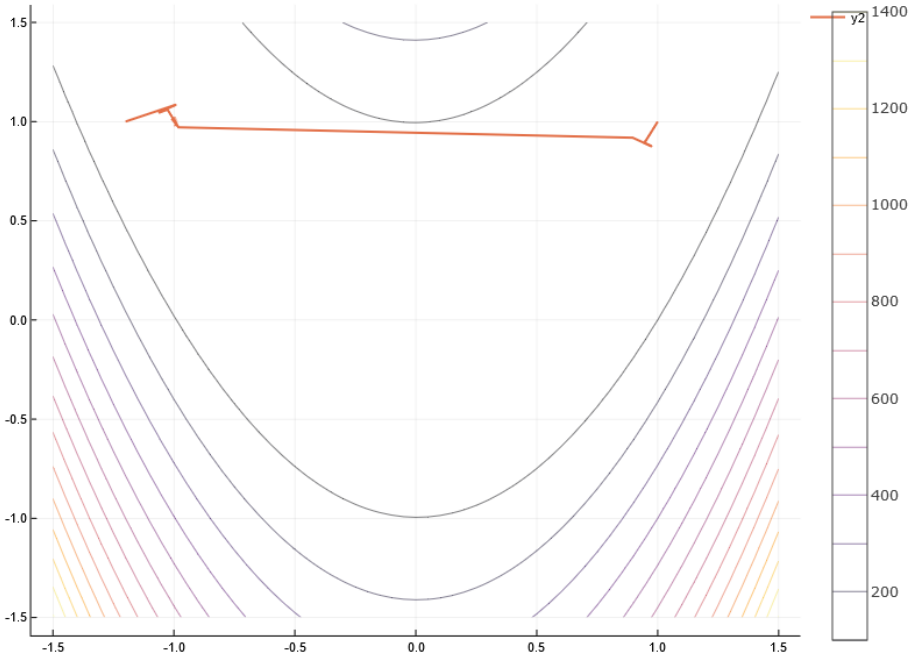

Figure 3: Rosenbrock function starting from $(-1.2, 1.0)$ using Gradient Descent and backtracking.

Question 4:

Prepare a plot of the step lengths as in part 2.

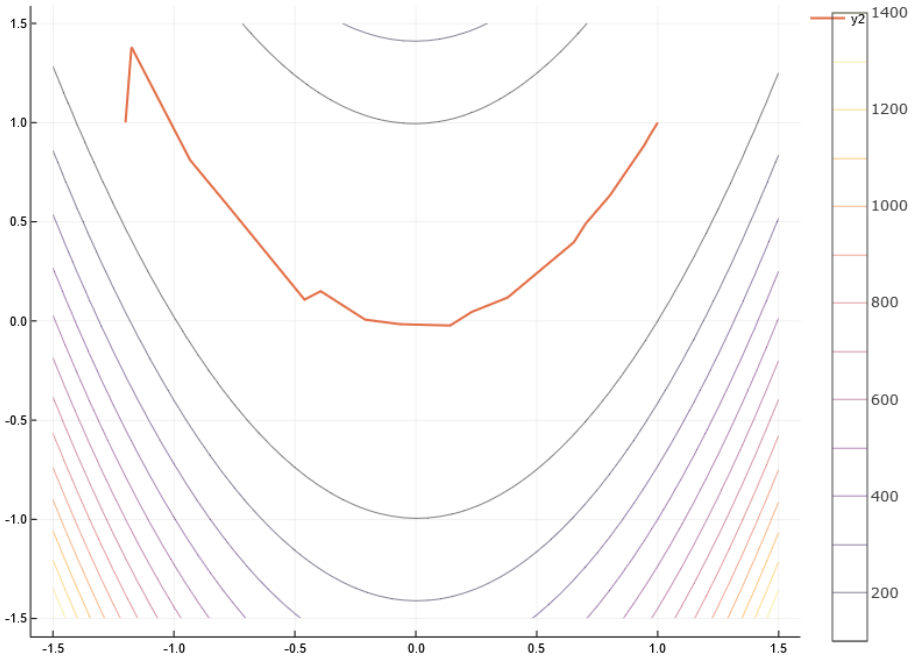

Figure 4: Rosenbrock function starting from $(-1.2, 1.0)$ using Newton direction and backtracking.

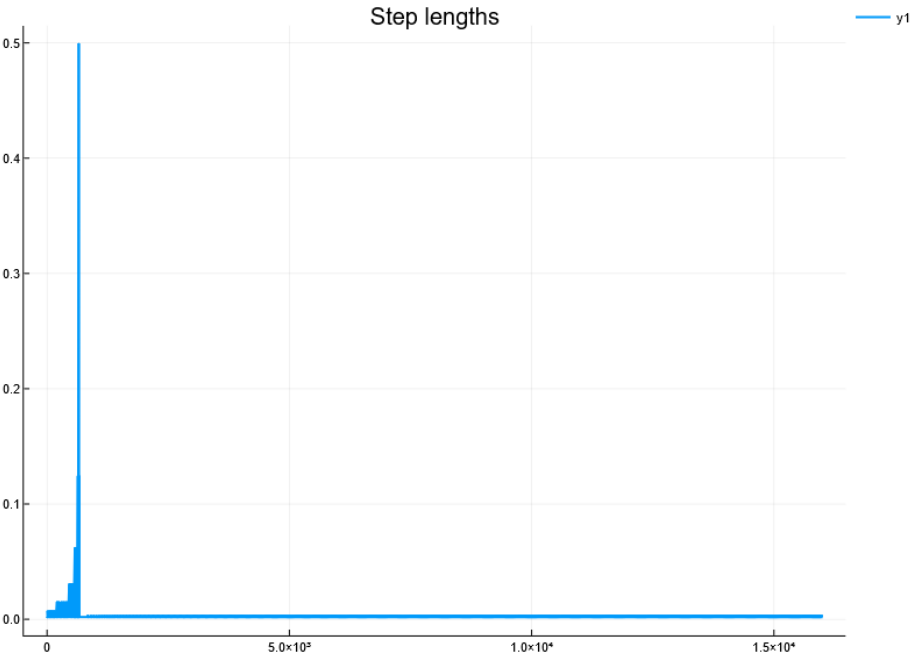

Figure 5: Step length with Rosenbrock function starting from $(1.2, 1.2)$ using Gradient Descent and backtracking.

Step lengths with Newton direction and backtracking are in Figure 7 and 8.

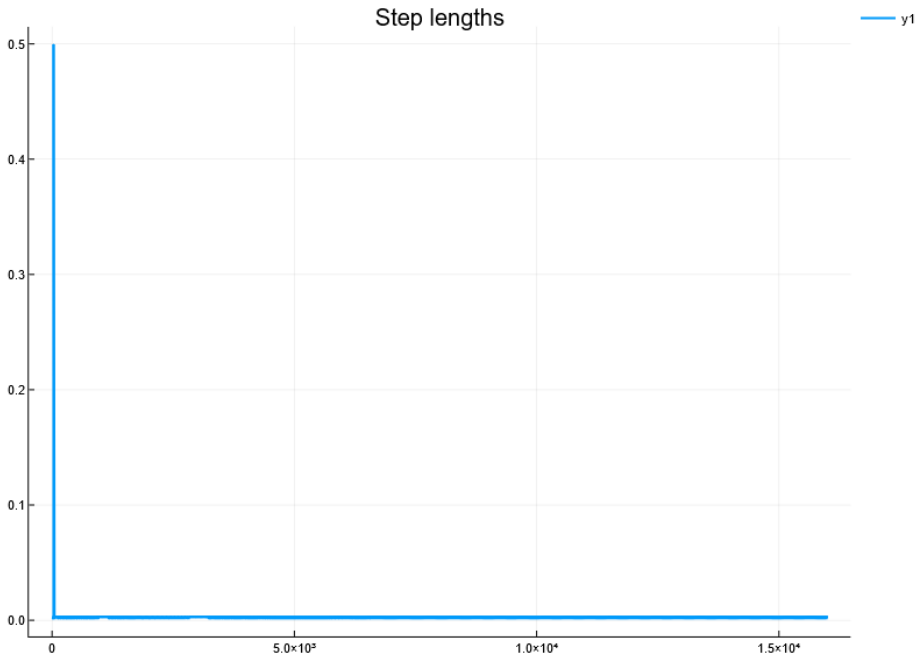

Figure 6: Step length with Rosenbrock function starting from $(-1.2, 1.0)$ using Gradient Descent and backtracking.

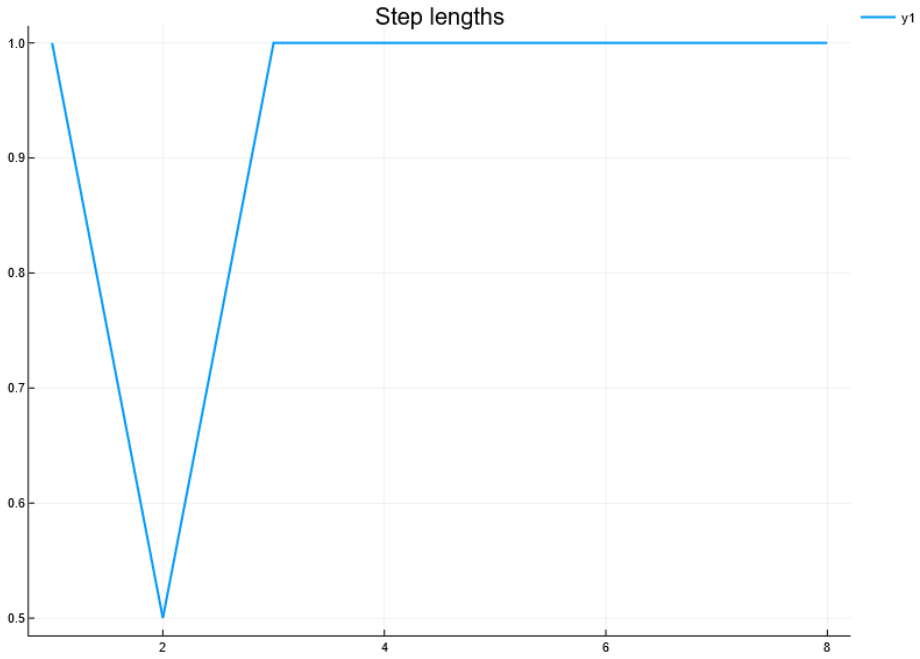

Figure 7: Step length with Rosenbrock function starting from $(1.2, 1.2)$ using Newton direction and backtracking.

Question 5:

Discuss any notable differences or similarities.

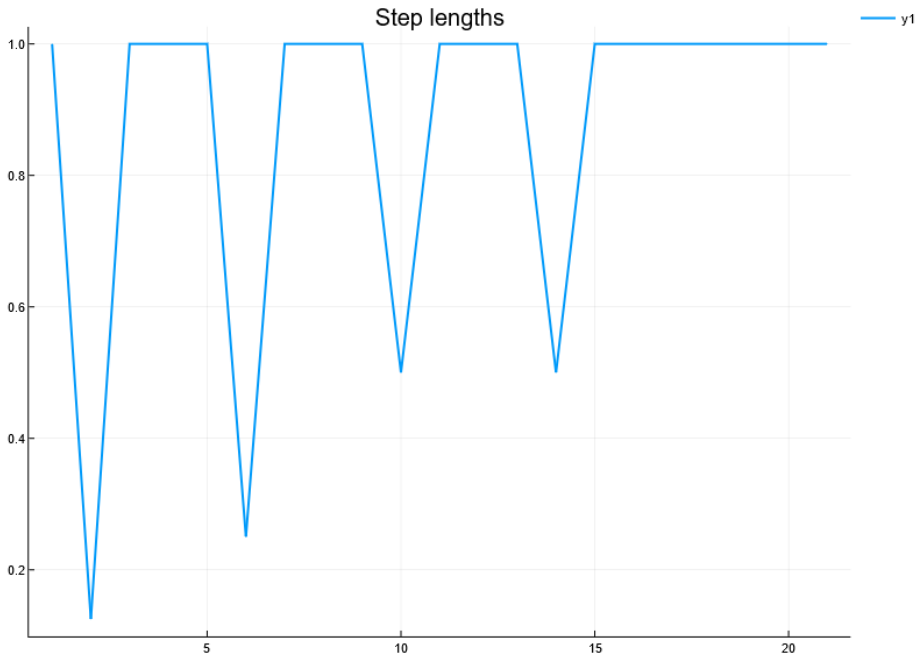

Figure 8: Step length with Rosenbrock function starting from $(-1.2, 1.0)$ using Newton direction and backtracking.

Starting from both points, we can see that a lot less steps are required with the Newton direction than the Gradient Descent. Using the Newton direction, the step size is almost always 1. On the contrary, with the Gradient Descent direction, it is always close to zero. Therefore, Gradient Descent converges slower.

Problem 2

Describe what would happen if you used backtracking line search on a strongly convex quadratic:

$$f(\mathbf{x}) = 1/2\mathbf{x}^T\boldsymbol{Q}\mathbf{x} - \mathbf{x}^T\mathbf{c}.$$

Be as precise as possible and think about comparing with *exact* line search.

The only difference between the two methods is how we choose the step length. Exact line search looks for the minimum of $f(\mathbf{x} - \alpha\mathbf{g})$ and can go as far as needed. Conversely, backtracking tries different discrete values of α, starting with 1, $1/2$, $1/4\ldots$ Thus it explores only a limited set of possible solutions. Although it is always going to find a step that decreases the function f, the vast majority of time, it is not going to be the optimal step as found with exact line search. So first conclusion, backtracking cannot be better than exact line search, it's only an approximation. However, it is not significantly worse, because of the sufficient decrease condition, it will converge linearly. Simply, the rate of convergence is not going to be as fast as exact linear search. Usually it is not possible to do an exact line search, that's why backtracking is a good approximation.

Problem 3

Consider the function

$$f(\mathbf{x}) = -\mathbf{q}^T\mathbf{x} + \sum_j y_j \log(\boldsymbol{C}^T\mathbf{x})_j$$

where $\boldsymbol{C}$ is a very large matrix and computing $\boldsymbol{C}^T\mathbf{x}$ is expensive. Show how we can use the structure of the function to reduce the number of matrix vector products $\boldsymbol{C}^T\mathbf{x}$ that would be required for checking the Wolfe conditions when searching in a direction $\mathbf{p}$. (Hint, compute $\mathbf{w} = \boldsymbol{C}^T\mathbf{p}$ once, and see how to re-use it.)

The Wolfe conditions are:

$$f(\mathbf{x}_k + \alpha\mathbf{p}_k) \leq f(\mathbf{x}_k) + \alpha c_1 \mathbf{p}_k^T \mathbf{g}_k$$

$$g(\mathbf{x}_k + \alpha\mathbf{p}_k) \geq c_2 \mathbf{g}_k^T \mathbf{p}_k$$

The gradient of our function is:

$$\frac{\partial f}{\partial x_i} = -q_i + \sum_j y_j \frac{c_{ij}}{(\boldsymbol{C}^T\mathbf{x})_j}$$

In the case of our function, the evaluation of these conditions needs to be cheap when only α is varying. For that we compute some expressions in advance in order to avoid the computation of $\boldsymbol{C}^T\mathbf{x}$. At the beginning of the iteration k, we compute $\mathbf{v} = \boldsymbol{C}^T\mathbf{x}_k$, $\mathbf{w} = \boldsymbol{C}^T\mathbf{p}$, $f(\mathbf{x}_k)$, $\mathbf{g}_k$ and we reuse them every time we check the Wolfe conditions. It works because $\boldsymbol{C}^T\mathbf{x}$ is linear.

When we check the two conditions, we compute:

$$f(\mathbf{x}_k + \alpha\mathbf{p}_k) = -\mathbf{q}^T(\mathbf{x}_k + \alpha\mathbf{p}_k) + \sum_j y_j \log(\boldsymbol{C}^T(\mathbf{x}_k + \alpha\mathbf{p}_k))_j$$

and

$$\frac{\partial f}{\partial x_i} = -q_i + \sum_j y_j \frac{c_{ij}}{(\boldsymbol{C}^T(\mathbf{x}_k + \alpha\mathbf{p}_k))_j}$$

The most expensive term to compute is: $\boldsymbol{C}^T(\mathbf{x}_k + \alpha\mathbf{p}_k)$, which is equal to $\mathbf{v} + \alpha\mathbf{w}$ (simply the sum of 2 vectors).