Unconstrained Optimization Assignment

Andrea Cognolato, s281940@studenti.polito.it

January 25, 2021

1 Introduction

Descent methods are a vast family of iterative optimization algorithms for finding a local minimum of a differentiable function f. They operate by repeatedly moving in a direction where f decreases.

More rigorously, descent methods aim to solve the following unconstrained optimization problem:

$$\min_{x} f(x)$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{R}^n$.

To do so, a direction p_k is chosen such that $\nabla f(x_k)^T p_k < 0$ and the next iterate x_{k+1} is generated by moving along p_k with a step of length α_k. That is: $x_{k+1} = x_k + \alpha_k p_k$.

There are several ways to choose p_k. In this assignment, we will consider three methods: Steepest Descent, Fletcher Reeves, and Polak–Ribière.

Likewise, we have many possible choices for choosing the step length α_k, which result in various *line search* strategies. In our case we will use backtracking line search, a strategy based on repeatedly shrinking our step length α until the Armijo conditions are satisfied.

We implement these three methods in Julia[1], and analyze their computation time on problems of size $n = 10^4, 10^5$ and using both the exact gradient, forward finite differences, and centered finite differences.

1.1 Steepest Descent

Steepest descent (or gradient descent) is one of the simplest methods. At every iteration the steepest direction, i.e. the direction along with f decreases the most, is picked.

The gradient of a function at a point is denoted by $\nabla f(x_k)$, and is a vector indicating the direction of most rapid ascent, by taking its opposite, we obtain the direction of steepest descent.

$$p_k^{SD} = -\nabla f(x_k)$$

Steepest descent unfortunately can display pathological behaviour on functions with very narrow valleys. In this case, the algorithm assumes a "zig-zagging" behaviour where a lots of tiny steps are made towards the solution, drastically decreasing its convergence speed.

1.2 Fletcher-Reeves

To solve steepest descent's the issue with very narrow valleys, nonlinear conjugate gradient methods, such as Fletcher-Reeves were developed. To compute their descent directions, they combine the direction of steepest descent with the descent directions of the previous step, weighted by a factor β_k.

[1] Full codes available at github.com/mrandri19/polito-numerical-optimization

$$d_k = -\nabla f(x_k)$$

$$p_{k+1} = d_k + \beta_k p_k$$

In particular, in Fletcher-Reeves, the coefficient β_k is computed as follows:

$$\beta_k^{FR} = \frac{\nabla f(x_{k+1})^T \nabla f(x_{k+1})}{\nabla f(x_k)^T \nabla f(x_k)}$$

1.3 Polak–Ribière

Polak–Ribière is an alternative nonlinear conjugate gradient method, where β_k is computed as follows:

$$\beta_k^{PR} = \frac{\nabla f(x_{k+1})^T (\nabla f(x_{k+1}) - \nabla f(x_k))}{\nabla f(x_k)^T \nabla f(x_k)}$$

1.4 Backtracking Line Search

In our case, nonlinear optimization, finding the best step length α is itself a nonlinear optimization problem. To avoid solving it exactly at every iteration, we use backtracking.

In backtracking we start with $\alpha = \alpha_0$, and iteratively shrink it by $\alpha \leftarrow \rho\alpha$, $\rho \in (0, 1)$, until the Armijo condition is met:

$$f(x_k + \alpha_k d_k) < f(x_k) + c_1 \alpha \nabla f(x_k)^T d_k$$

2 Problem analysis

In our assignment we are tasked with applying the three optimization algorithms to the following function:

$$\min_{x \in \mathbb{R}^n} \sum_{i=1}^{n} (\frac{1}{4} x_i^4 + \frac{1}{2} x_i^2 + x_i) = \min_{x \in \mathbb{R}^n} f(x)$$

First of all we compute symbolically the gradient, since we are going to be comparing the exact derivatives with the ones computed via finite differences.

$$(\nabla f(x))_i = x_i^3 + x_i + 1$$

Then, since this problem is additively separable, we can write it as

$$\min_{x \in \mathbb{R}^n} f(x) = \min_{x \in \mathbb{R}^n} \sum_{i=1}^{n} g(x_i)$$

where $g(x) = x^4/4 + x^2/2 + x$. Notice how since g is convex, f is convex due to being a sum of gs.

Since all elements of the sum in f have the same minimum, then

$$\min_{x \in \mathbb{R}^n} \sum_{i=1}^{n} g(x_i) = \sum_{i=1}^{n} \min_{x \in \mathbb{R}^n} g(x_i)$$

Using WolframAlpha we can calculate the minimum for g, which is $x^* \approx -0.682328....$ Thus the optimal $x \in \mathbb{R}^n$ will be $(-0.682328, ..., -0.682328)$. We will use this to verify the correctness of our algorithms, but we will not exploit separability for minimization, which would allow us to compute the complete solution using just one dimension.

2.1 Exploiting separability in the gradient approximation

A naïve implementation of finite differences would require $O(n)$ evaluations of f, one for each $\frac{\partial f(x)}{\partial x_i}$. In turn, each evaluation of f requires $O(n)$ evaluations of g, because of the summation from 1 to n.

By exploiting separability we can rewrite the finite differences to require only $O(n)$ evaluations of g for the whole gradient, rather than $O(n^2)$.

Firstly, we define $F(x)$, $F : \mathbb{R}^n \to \mathbb{R}^n$:

$$F(x) = [g(x_1), g(x_2), ..., g(x_n)]^T$$

We can get back to our original f by multiplying with a vector of ones

$$f(x) = 1_n^T F(x)$$

The first finite difference, where $e_1 = [1, 0, ..., 0]$ is the first basis vector, then is

$$\begin{aligned}
f(x + he_1) - f(x) &= 1_n^T F(x + he_1) - 1_n^T F(x) \\
&= 1_n^T (F(x + he_1) - F(x)) \\
&= 1_n^T (\begin{bmatrix} g(x_1 + h) \\ g(x_2) \\ \vdots \\ g(x_n) \end{bmatrix} - \begin{bmatrix} g(x_1) \\ g(x_2) \\ \vdots \\ g(x_n) \end{bmatrix}) \\
&= 1_n^T (\begin{bmatrix} g(x_1 + h) - g(x_1) \\ 0 \\ \vdots \\ 0 \end{bmatrix} \\
&= g(x_1 + h) - g(x_1)
\end{aligned}$$

From this, we can obtain the whole gradient in a single pass with:

$$\nabla^{FD} f(x) = \begin{bmatrix} g(x_1 + h) - g(x_1) \\ g(x_2 + h) - g(x_2) \\ \vdots \\ g(x_n + h) - g(x_n) \end{bmatrix}$$

3 Results

3.1 Comparing optimization methods

We begin by comparing the computation time and number of iterations for the three optimization methods and the three gradient computation methods. For this comparison we fix the finite difference step size k to -8, as well as the number of dimensions n to 10^5. The computation time is plotted in figure 1 while the number of iterations can be found in figure 2.

We can observe how all three algorithms converge in a similar number of iterations. Steepest descent being on average the fastest with ≈ 45 iterations while Polak–Ribière the slowest requiring ≈ 65. Additionally we see that, as expected, the fastest method for calculating the gradient is the exact method. Finite difference methods, despite being optimized to exploit separability, still require double the amount of floating point operations compared to the exact one. This effect is particularly noticeable for Fletcher-Reeves where we see how there is almost a 4x difference in runtime, despite a slightly increase in iterations. Analogous considerations apply for $n = 10^4$ as seen in tables 1, 2.

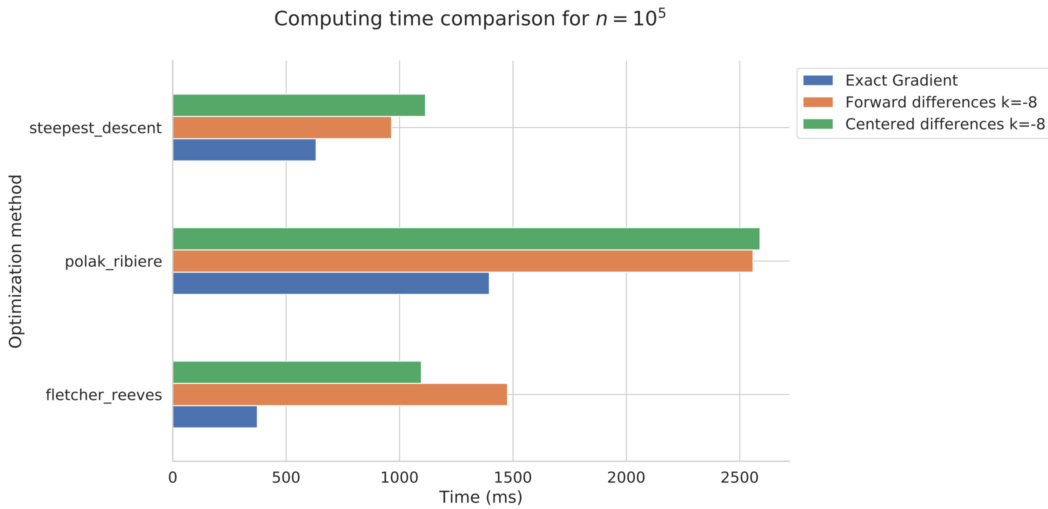

Figure 1: Computing time comparison

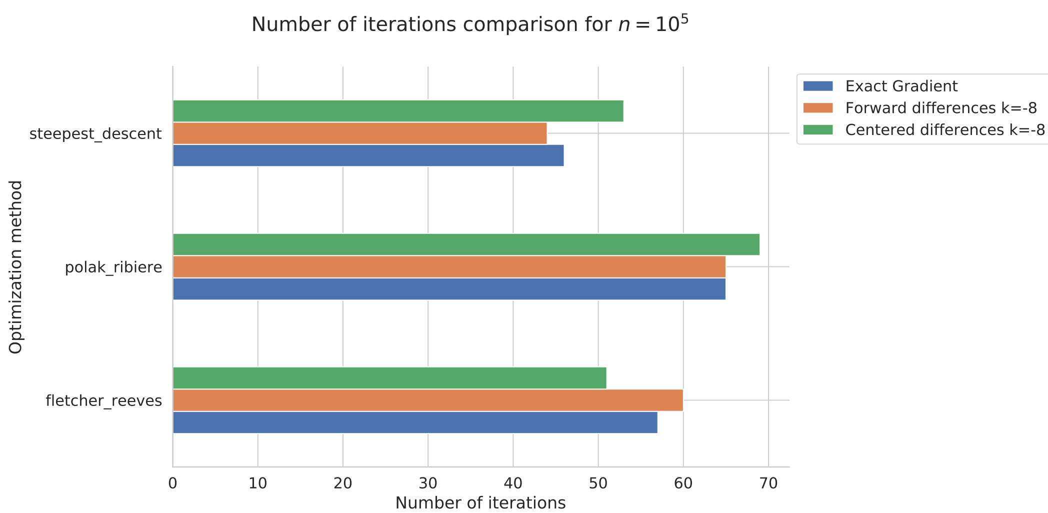

Figure 2: Number of iterations comparison

3.2 Effect of finite differences step size

We move on to comparing how different values of k affect the performance and convergence of the algorithms. The coefficient k is used to compute the step size $h = 10^{-k}||\hat{x}||$, where $\hat{x}$ is the point at which the derivative approximation is computed.

In figures 3 and 4 we see the results for $n = 10^5$ and using steepest descent. The algorithms fail to converge with step sizes too long i.e. $k = 2, 4, 6$, as the number of iterations has reached $K_{max} = 1000$. Notice as

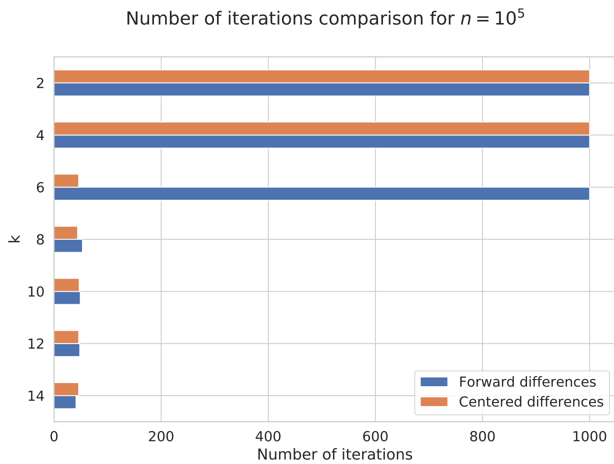

Figure 3: Number of iterations for different values of k, using steepest descent

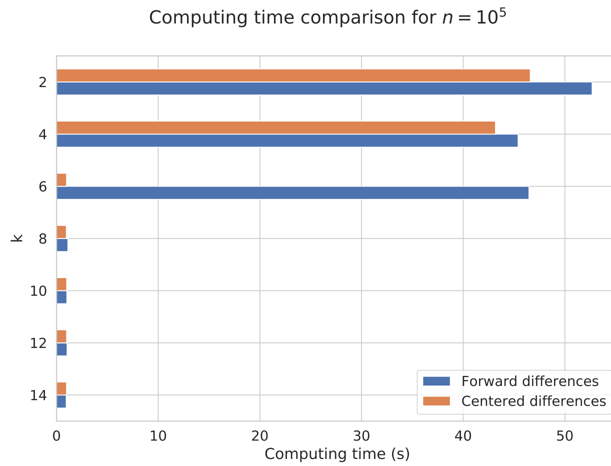

Figure 4: Computing time

well as for $k = 6$ the centered differences algorithm maintains enough precision to make steepest descent converge, while forward differences, with their $O(h)$ error, can not. Similar behaviour is observed for the other optimization methods, as seen in tables 1, 2.

Optimization Method	Computing time Steepest Descent		Fletcher-Reeves		Polak–Ribière	
Problem size	4	5	4	5	4	5
Gradient method						
Centered k=10	0.104	0.998	0.135	1.318	0.210	2.588
Centered k=12	0.105	0.991	0.113	1.329	0.207	2.811
Centered k=14	0.091	0.983	0.095	1.270	0.222	2.733
Centered k=2	4.271	46.597	2.550	109.914	3.775	49.870
Centered k=4	0.114	43.175	0.078	0.671	0.274	3.196
Centered k=6	0.107	0.985	0.112	1.304	0.210	2.587
Centered k=8	0.109	0.965	0.116	1.476	0.212	2.559
Exact	0.048	0.632	0.047	0.373	0.137	1.396
Forward k=10	0.102	1.014	0.136	1.268	0.204	2.430
Forward k=12	0.106	1.036	0.120	1.499	0.208	2.534
Forward k=14	0.089	0.951	0.091	1.254	0.220	2.866
Forward k=2	3.840	52.684	3.567	114.210	3.902	51.761
Forward k=4	4.264	45.400	2.369	32.173	0.207	1.529
Forward k=6	0.086	46.468	0.073	0.736	0.231	2.913
Forward k=8	0.121	1.114	0.105	1.096	0.204	2.589

Table 1: Computing times (s) for all combinations of methods and parameters

Optimization Method	Iterations Steepest Descent		Fletcher-Reeves		Polak–Ribière	
Problem size	4	5	4	5	4	5
Gradient method						
Centered k=10	49	47	69	56	65	69
Centered k=12	47	46	58	56	65	71
Centered k=14	43	46	48	50	69	69
Centered k=2	1000	1000	709	1000	1000	1000
Centered k=4	51	1000	38	27	87	87
Centered k=6	48	46	58	56	65	69
Centered k=8	45	44	60	60	65	65
Exact	47	46	69	57	65	65
Forward k=10	48	49	70	57	65	65
Forward k=12	49	48	63	56	65	67
Forward k=14	43	41	45	47	70	79
Forward k=2	1000	1000	1000	1000	1000	1000
Forward k=4	1000	1000	602	556	65	41
Forward k=6	42	1000	38	30	73	73
Forward k=8	57	53	55	51	65	69

Table 2: Number of iterations for all combinations of methods and parameters

4 Code

```
# The function f we need to optimize
function f(x)
    val = 0.0
     for i = eachindex(x)
        val += 1/4 * x[i]^4 + 1/2 * x[i]^2 + x[i]
    end
    return val
end

# The gradient of f, computed symbolically
∇f(x) = [x[i]^3 + x[i] + 1 for i = eachindex(x)]

# Compute the optimal step length via backtracking line
# search, using Armijo-Goldstein conditions
function backtrack(xk, dk, f, ∇fk, α0, c1, ρ, btmax)
    α = α0
    bt = 1
    fk = f(xk)
    ∇f_dk = ∇fk' * dk
    while bt < btmax
         if (f(xk + α * dk) < fk + c1 * α * ∇f_dk)
            break
        end
        α = ρ * α
        bt += 1
    end
    return bt, α
end

```

```julia
# Nonlinear steepest descent method.
function steepest_descent(
        x0, # starting point
        f, ∇f, # function handles for the objective and its gradient
        rel_diff=1e-8, kmax=1000, # solver hyperparameters
        α0=5.0, c1=1e-4, ρ=0.8, btmax=50 # backtracking hyperparameters
    )
    k = 0
    xk = x0

    while k < kmax
        ∇fk = ∇f(xk)
        dk = -∇fk

        bt, αk = backtrack(xk, dk, f, ∇fk, α0, c1, ρ, btmax)

        xk_new = xk + αk * dk
        # relative movement stopping criterion
        if norm(xk_new - xk) / norm(xk) < rel_diff
            break
        end
        xk = xk_new

        k += 1
    end

    return xk, k
end

function fletcher_reeves(
        x0,
        f, ∇f,
        rel_diff=1e-8, kmax=1000,
        α0=5.0, c1=1e-4, ρ=0.8, btmax=50
    )
    xk = x0
    ∇fk = ∇f(xk)

    pk = -∇fk
    k = 0

    while k < kmax
        bt, αk = backtrack(xk, pk, f, ∇fk, α0, c1, ρ, btmax)

        xkp1 = xk + αk * pk
        if norm(xkp1 - xk) / norm(xkp1) < rel_diff
            break
        end

        ∇fkp1 = ∇f(xkp1)
        βkp1 = (∇fkp1' * ∇fkp1) / (∇fk' * ∇fk)

        pk = -∇fkp1 + βkp1 * pk

```

```
        xk = xkp1
        ∇fk = ∇fkp1

        k = k + 1
    end

    return xk, k
end

function polak_ribiere(
        x0,
        f, ∇f,
        rel_diff=1e-8, kmax=1000,
        α0=5.0, c1=1e-4, ρ=0.8, btmax=50
    )
    xk = x0
    ∇fk = ∇f(xk)

    pk = -∇fk
    k = 0

    while k < kmax
        bt, αk = backtrack(xk, pk, f, ∇fk, α0, c1, ρ, btmax)

        xkp1 = xk + αk * pk
        if norm(xkp1 - xk) / norm(xkp1) < rel_diff
            break
        end

        ∇fkp1 = ∇f(xkp1)
        βkp1 = (∇fkp1' * (∇fkp1 - ∇fk)) / (∇fk' * ∇fk)

        pk = -∇fkp1 + βkp1 * pk

        xk = xkp1
        ∇fk = ∇fkp1

        k = k + 1
    end

    return xk, k
end

g(x) = 1/4 * x^4 + 1/2 * x ^ 2 + x

function ∇f_fwd_diff(x; k=8)
    h = 10.0^(-k) * norm(x)
    return [(g(x[i] + h) - g(x[i])) / h for i = eachindex(x)]
end

function ∇f_cnt_diff(x; k=8)
    h = 10.0^(-k) * norm(x)
    return [(g(x[i] + h) - g(x[i] - h)) / 2h for i = eachindex(x)]
end
```