University of Mannheim
Scientific Computing, B6 26, C306, 68131 Mannheim
Dr. Andreas Sommer (ansommer@mail.uni-mannheim.de)

UNIVERSITY OF MANNHEIM

Nonlinear Optimization (FSS 2023)

Exercise Sheet #8

Due on 30.04.2023 (before 13:00).

A naïve implementation of Quasi-Newton methods as introduced in the lecture requires the (explicit) calculation and construction of the (inverse) Hessian approximation $B_k \in \mathbb{R}^{n\times n}$. For large-scale problems, i.e. n is large, the required space $O(n^2)$ might be infeasible.

Limited-memory Quasi-Newton methods store only a limited subset of curvature information (i.e. s^k and y^k), that is, only the last m vectors. Furthermore, the B_k matrices are never explicitly formed. Instead, the matrix-vector products $B_k \cdot x$ are directly computed, reducing the required memory to $O(m \cdot n)$.

In this exercise, you will implement a *limited memory inverse BFGS algorithm* (see exercise 1) without explicit formation of the inverse Hessian matrix approximations, and test its effectiveness on the extended Rosenbrock function.

General remarks to the exercise

- **Test and helper functions can be downloaded from the course homepage:**
  i) Extended Rosenbrock function: `extRosenbrock.m` and `extRosenbrockGradient.m`
  ii) Powell-Wolfe step size strategy: `powellwolfe.m`
  iii) Iteration output function: `outputIteration.m`
  iv) Explicit DFP and BFGS update functions: `updateBFGS.m` and `updateDFP.m`
  v) The full-matrix inverse BFGS method (see below) for comparison: `bfgsinverse.m`

- **Inverse BFGS Method**
  For comparison, the globalized inverse BFGS method (algorithm 12) with Powell-Wolfe step size strategy for globalization is implemented in `bfgsinverse.m`. The algorithm stops if either $\|\nabla f(x^k)\| \leq tol$ or a maximum number of iterations is exceeded.
  The function header is given as

  ```
  function X = bfgsinverse(f, gradf, B0, x0, tol, maxit, gamma, eta, output)
  ```

  where `f` is a function handle to the objective, `gradf` a function handle to the objective's gradient, `B0` an initial inverse Hessian approximation, `x0` the initial guess, `tol` the stopping criterion tolerance, `maxit` the maximum number of iterations, and `gamma` and `eta` are the parameters of the Powell-Wolfe step size strategy. The result matrix `X` contains all generated iterates.
  When `output` is set to `true` , the algorithm prints in every iteration in a single line: the iteration number, the norm of the gradient, the contraction rate, the function value, the step size, and the first 10 components of the current iterate x^k.

- Test your program with the *extended Rosenbrock function*:

$$f(x) := \sum_{i=1}^{n/2} \left(\alpha(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2\right) \quad \text{for } x \in \mathbb{R}^n \text{ with } n \text{ even}$$

  The larger the parameter α, the higher the non-linearity of the problem (and the more difficult it is to solve). The solution is always $\tilde{x} = (1, 1, ..., 1)^\top$ with $f(\tilde{x}) = 0$.

- The initial guesses can be generated by `x0 = random('normal', 0.0, 20.0, [n 1]);`
  Hint: For testing purposes, it is useful to always initialize the random number generator with a fixed seed, e.g. 1234, by calling `rng(1234)` *before generating the initial guess.*

- Time measurements can be done using the function pair `tic()` and `toc()`.
  When measuring the time, make sure that the output in every iteration is *disabled* (!)

1. Implement the Limited Memory Inverse BFGS Method (L-BFGS) [6+6 points]

In this exersice, you first implement an algorithm for efficiently calculating a Quasi-Newton direction without explicitly building the inverse Hessian approximation B_k. In a second step, you implement a limited-memory BFGS method with Powell-Wolfe step-size strategy for globalization. Finally, you test the program on an extended (high-dimensional) Rosenbrock function

We use the following notations:

$$s^k := x^{k+1} - x^k \ , \qquad y^k := \nabla f(x^{k+1}) - \nabla f(x^k) \ , \qquad \rho_k := \frac{1}{(y^k)^\top s^k}$$

Given the current iterate number k, the current gradient $\nabla f(x^k)$, and the "history" of curvature information up to now, i.e. all ρ_i, s^i, y^i for $i = 0, ..., k$, the following algorithm can be used to calculate the product $B_k \nabla f(x^k)$ (note: no minus sign!):

Algorithm (X1): L-BFGS two-loop recursion

Input: current gradient $\nabla f(x^k)$, memory size m, history ρ_i, s^i, y^i $(i = 0, ..., k)$.

1: $q := \nabla f(x^k)$
2: **for** $i = k-1, k-2, \ldots, k-m$ **do**
3: $\quad \alpha_i := \rho_i (s^i)^\top q$
4: $\quad q := q - \alpha_i y^i$
5: **end for**
6: $d := B_k^0 q$
7: **for** $i = k-m, k-m+1, \ldots, k-1$ **do**
8: $\quad \beta := \rho_i (y^i)^\top d$
9: $\quad d := d + s^i(\alpha_i - \beta)$
10: **end for**

The initial inverse Hessian approximation B_k^0 can be freely chosen in every iteration. An effective choice is to choose a scaled version of the unit matrix $I \in \mathbb{R}^{n \times n}$:

$$B_k^0 := \frac{(s^{k-1})^\top y^{k-1}}{(y^{k-1})^\top y^{k-1}} \cdot I$$

However, the matrix B_k^0 is also not explicitly formed (as it would require $O(n^2)$ storage). Instead, in the above formulation, only the scaling factor in front of the unit matrix is used.

In a first step, implement algorithm (X1) in a Matlab function with header

```
function d = lbfgsinverse_evalBkd(gradfxk, k, m, rho, s, y)
```

that gets as input the current gradient $\nabla f(x^k)$ in `gradfxk`, the current iteration number `k`, the memory size `m`, and the *full* curvature history in cell arrays `rho`, `s`, `y`. Here, *full* means that these arrays store *all* generated scalings ρ_k and vectors s^k, y^k, although only the last m are used during optimization. The result variable `d` contains the product $B_k \nabla f(x^k)$.

Hint: You may use the given function `lbfgsinverse_buildBk.m` to explicitly build the matrix B_k from s^k, y^k and ρ_k. This might prove useful to verify whether the product $B_k d^k$ is computed correctly by your implementation of algorithm (X1).

In a second step, implement the limited memory inverse BFGS algorithm:

Algorithm (X2): L-BFGS method

Input: initial guess x^0, memory size $m > 0$, tolerance *tol*

STOP, if $\|\nabla f(x^0)\| \leq tol$
for $k = 0, 1, 2, ...$ **do**
  if $k = 0$ **then**
    Use $d^0 = -\nabla f(x^0)$ as initial search direction
  else
    Determine search direction $d^k = -B_k \nabla f(x^k)$ using algorithm (X1)
  end if
  Compute step size $\sigma_k > 0$ with Powell-Wolfe step size strategy
  $x^{k+1} := x^k + \sigma_k d^k$
  STOP, if $\|\nabla f(x^{k+1})\| \leq tol$
  Compute and store curvature information:

$$s^k := x^{k+1} - x^k \ , \quad y^k := \nabla f(x^{k+1}) - \nabla f(x^k) \ , \quad \rho_k := \frac{1}{(y^k)^\top s^k}$$

end for

Test your program with the *extended Rosenbrock function*: For $\alpha = 1$ as well as $\alpha = 10$, choose all combinations of history sizes $m = 1$, $m = 5$, and $m = 20$, and dimensions $n = 100$, $n = 1000$, and $n = 10000$.

Measure the required time, number of iterations, and function evaluations.

As initial guess, choose a random vector $x_0 \in \mathbb{R}^n$ with normally distributed components with mean zero and standard deviation twenty. Make sure that you are always using the same initial guess by fixing the random seed.

Use the following parameters: `maxiter=10000`, `tol=1.0d-6`, `eta=0.9`, `gamma=1.0d-4`.

Measure the required time and the count the number of iterations and function evaluations.

Hint: To verify your code, you can compare your results to the full-matrix inverse BFGS implementation in `bfgsinverse.m` (possible only for the smaller n).

Further reading: Nocedal & Wright, Numerical Optimization, Chapter 9: L-BFGS method. Note that there the inverse Hessian approximation is denoted as H_k.

Put all files into a single zip-archive, named by the lexicographically ordered family names of all group members separated by a hyphen, and send the zip-file to ansommer@mail.uni-mannheim.de.
Add printouts (PDFs) from code and output to your submissions.
Comment your code intensely. Use a complete header that describes input and output arguments and also comment the implementation where appropriate (see the examples at the course web site).
Avoid obvious inefficiencies like repeated evaluation of identical/unchanged expressions.