Practical 4: a BFGS optimizer

This practical is to be completed in groups of 3. What is submitted must be solely the work of the members of the submitting group. Code must not be shared between groups. We have automatic systems available for checking for this, but in addition students tend to make distinctive errors in coding, or use distinctively convoluted solutions to problems: these tend to stand out even if you do the obvious things to try and hide the code sharing. You can use code from the lecture notes without citation, but if you use any other code that you did not write yourself you should cite where it came from. All group members are expected to make a substantive contribution - email me if you have problems in this regard (work will never be completely equal, but there should be proper engagement).

Task: to write an R function, `bfgs`, implementing the BFGS quasi-Newton minimization method.

Specification: Your `bfgs` optimization function should operate broadly in the same way as `optim` or `nlm`, but there are some differences of detail. Note that the purpose is to have an independent implementation: you must code the optimization yourself, not simply call optimization code written by someone else. The implementation can use base R, and any of the packages included with base R, but should not use any other packages. The function arguments should be as follows:

```
bfgs(theta,f,...,tol=1e-5,fscale=1,maxit=100)
```

`theta` is a vector of initial values for the optimization parameters.

`f` is the objective function to minimize. Its first argument is the vector of optimization parameters. Its second argument is a logical indicating whether or not gradients of the objective w.r.t. the parameters should be computed. Remaining arguments will be passed from `bfgs` using '`...`'. The scalar value returned by `f` will have a `gradient` attribute if the second argument to `f` is `TRUE`.

`...` any arguments of `f` after the first two (the parameter vector, and gradient logical indicator) are passed using this.

`tol` the convergence tolerance.

`fscale` a rough estimate of the magnitude of f at the optimum - used in convergence testing.

`maxit` the maximum number of BFGS iterations to try before giving up.

Your `bfgs` function should return a (named) list containing:

`f` the scalar value of the objective function at the minimum.

`theta` the vector of values of the parameters at the minimum.

`iter` the number of iterations taken to reach the minimum.

`g` the gradient vector at the minimum (so the user can judge closeness to numerical zero).

`H` the approximate Hessian matrix (obtained by finite differencing) at the minimum.

The function should issue errors or warnings (using `stop` or `warning` as appropriate) in at least the following cases. 1. If the objective or derivatives are not finite at the initial `theta`; 2. If the step fails to reduce the objective but convergence has not occurred 3. If `maxit` is reached without convergence.

If the supplied `f` does not supply a `gradient` attribute when requested, then you should compute the gradient by finite differencing. Make sure that the design for dealing with this is clean and simple.

Other considerations:

1. You will need to reduce the step length if a step leads to a non-finite objective value. You will also need to ensure that your step length both reduces the objective function and satisfies the second Wolfe condition (use $c_2 = 0.9$). For this practical you don't have to worry about the first Wolfe condition (any decrease will suffice). You will need to figure out a sensible way of searching for the right step length (but a perfect method is not required for this practical).

2. To judge whether the gradient vector is close enough to zero, you will need to consider the magnitude of the objective (you can't expect gradients to to be down at 10^{-10} if the objective is of order 10^{10}, for example). So the gradients are judged to be zero when they are smaller than `tol` multiplied by the objective. But then there is a problem it the objective is zero at the minimum - we can then never succeed in making the magnitude of the gradient less than the magnitude of the objective. So `fscale` is provided. Then if `f0` is the current value of the objective and `g` the current gradient,

   ```
   max(abs(g)) < (abs(f0)+fscale)*tol
   ```

   is a suitable condition for convergence.

3. The final approximate Hessian matrix can be generated by finite differencing the gradient vector. Such an approximate Hessian will be asymmetric: `H <- 0.5 * (t(H) + H)` fixes that.

4. If p is the number of parameters, then the update of the approximate inverse Hessian should have cost $O(p^2)$, not $O(p^3)$.

Here is one example objective function (actual minimum at $1, 1$):

```
rb <- function(theta,getg=FALSE,k=10) {
## Rosenbrock objective function, suitable for use by 'bfgs'
  z <- theta[1]; x <- theta[2]
  f <- k*(z-x^2)^2 + (1-x)^2 + 1
  if (getg) {
    attr(f,"gradient") <- c(2*k*(z-x^2),
    -4*k*x*(z-x^2) -2*(1-x))
  }
  f
} ## rb
```

Starting from $-1, 2$ optimization of `rb` should take around 20 steps. You will almost certainly need to use a debugger at some point in this project. If you have not managed to get the `debug` package to work, then you can use the `debug` function in R instead. e.g. `debug(bfgs)` then call `bfgs` and it will be run in debug mode (rather like with `mtrace`, but less convenient). `Q` to quit. `undebug(bfgs)` to turn off.

What to submit: One text file of carefully commented code should be submitted on Learn. This should contain no code outside of functions, and should load no libraries except those supplied with base R. The file should be called WGxxP4.R where 'xx' is replaced by your workgroup number (I don't care about the case: wgxxp4.r is also fine). The first line of the code should include your group number and the names of the people in your group. The second line should include the address of your github repo. Do not modify the repo after submission. Do not send me an invite to your repo — I will ask you for one if I need to see it. The deadline is 16:00 Friday 19th November. It is strongly recommended that you submit well in advance of this, as last minute computer glitches will not be taken as an excuse for late submission[1].

Marking Scheme: Full marks will be obtained for code that:

1. is clearly commented, well laid out and well structured.
2. is reasonably clean concise and efficient, and avoids significant inefficiencies, such as very inefficient matrix computation.
3. passes a variety of automated optimization task checks.
4. sticks exactly to the specification.
5. is coded only using what is available in base R.
6. avoids significant faults not covered by the above!

[1] This is for reasons of fairness - otherwise it becomes too easy to game the system for an extension.