Practical 4 (group-work): Newton optimizer

Programming Task: in your work group of 3, to write an R function, `newt`, implementing Newton's method for minimization of functions. Only functions in base R and its packages (including `recommended` packages) can be used and your code must not install any packages. Do not use functions `attach` or `solve`. Your submitted code should only define functions, and not have any code outside functions except for loading permitted package (it can of course contain functions other than `newt`).

Specification: Your `newt` optimization function should operate broadly in the same way as `nlm` and `optim`, but the purpose is to have an independent implementation: you must code the optimization yourself, not simply call optimization code written by someone else. The function arguments should be as follows:

```
newt(theta,func,grad,hess=NULL,...,tol=1e-8,fscale=1,maxit=100,max.half=20,eps=1e-6)
```

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

`func` is the objective function to minimize. Its first argument is the vector of optimization parameters. Remaining arguments will be passed from `newt` using '`...`'.

`grad` is the gradient function. It has the same arguments as `func` but returns the gradient vector of the objective w.r.t. the elements of parameter vector.

`hess` is the Hessian matrix function. It has the same arguments as `func` but returns the Hessian matrix of the objective w.r.t. the elements of parameter vector. If not supplied then `newt` should obtain an approximation to the Hessian by finite differencing of the gradient vector (hint: `(t(A)+A)/2` is exactly symmetric for any matrix `A`).

`...` any arguments of `func`, `grad` and `hess` after the first (the parameter vector) are passed using this.

`tol` the convergence tolerance.

`fscale` a rough estimate of the magnitude of `func` near the optimum - used in convergence testing.

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

`max.half` the maximum number of times a step should be halved before concluding that the step has failed to improve the objective.

`eps` the finite difference intervals to use when a Hessian function is not provided.

`newt` should return a list containing:

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

`theta` the value 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).

`Hi` the inverse of the Hessian matrix at the minimum (useful if the objective is a negative log likelihood).

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 despite trying `max.half` step halvings; 3. If `maxit` is reached without convergence; 4. If the Hessian is not positive definite at convergence.

Convergence should be judged by seeing whether all elements of the gradient vector have absolute value less than `tol` times the absolute value of the objective function plus `fscale`. Don't forget to deal with the case in which the Hessian is not positive definite[1]. Also make sure that your step halving procedure deals with the possibility of a trial step leading to a non-finite objective value (not only an increased objective). One appropriate set of test functions are these (minimum at 1,1)...

```
rb <- function(th,k=2) {
  k*(th[2]-th[1]^2)^2 + (1-th[1])^2
}

gb <-  function(th,k=2) {
  c(-2*(1-th[1])-k*4*th[1]*(th[2]-th[1]^2),k*2*(th[2]-th[1]^2))
}
```

[1] Hint: Cholesky decomposition fails for matrices that are not positive definite; function `try` allows you to run R code that may fail.

```
hb <- function(th,k=2) {
  h <- matrix(0,2,2)
  h[1,1] <- 2-k*2*(2*(th[2]-th[1]^2) - 4*th[1]^2)
  h[2,2] <- 2*k
  h[1,2] <- h[2,1] <- -4*k*th[1]
  h
}
```

You should also design your own tests (of different dimensions for `theta` for example). The tests are not to be submitted, but simply to check that the function does what is intended in general, and will pass marking tests.

As a group of 3 you should aim to produce well commented[2], clear and efficient code for the task. The code should be written in a plain text file (no special characters) called `proj4.r` and is what you will submit. Your solutions should use only the functions available in base R (or packages supplied with base R - no other packages). The work must be completed in your work group of 3, which you must have arranged and registered on Learn, and collaborative code development must be done using a github repo set up for this purpose.

The first comment in your code should list the names and university user names of each member of the group. The second comment should give the address of your github repo, which should be made public *after* the hand in deadline. The third comment **must** give a brief description of what each team member contributed to the project, and roughly what proportion of the work was undertaken by each team member. Contributions never end up completely equal, but you should aim for rough equality, with team members each making sure to 'pull their weight', as well as not unfairly dominating[3]. Any team member that did not take an active part in actually writing code for earlier group projects, should ensure they do so this time.

One piece of work - the text file containing your commented R code - is to be submitted for each group of 3 on Learn by 12:00 Friday 18th November 2022. Format the code and comments tidily, so that they display nicely on an 80 character width display, without line wraps (or only occasional ones). No extensions are available on this course, because of the frequency with which work has to be submitted. So late work will automatically attract a hefty penalty (of 100% after work has been marked and returned). Technology failures will not be accepted as a reason for lateness (unless it is provably the case that Learn was unavailable for an extended period), so aim to submit ahead of time.

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

1. does what it is supposed to do, without using `solve`, so that it correctly optimizes the function given on the sheet and a variety of unseen test cases.
2. is carefully commented, so that someone reviewing the code can easily tell exactly what it is for, what it is doing and how it is doing it without having read this sheet, or knowing anything else about the code. Note that *easily tell* implies that the comments must also be as clear and *concise* as possible. You should assume that the reader knows basic R, but not that they know exactly what every function in R does.
3. is well structured and laid out, so that the code itself, and its underlying logic, are easy to follow.
4. is efficiently coded, with matrix operations in particular coded to take the minimum leading order cost possible. e.g. code that could easily be done in $O(n^2)$ operations should not be coded to take $O(n^3)$, for $n \times n$ matrices.
5. was prepared collaboratively using git and github in a group of 3.
6. contains no evidence of having been copied, in whole or in part, from other students on this course, students at other universities (there are now tools to help detect this, including internationally), previous students, online sources etc.
7. includes the comment stating team member contributions.

Highest marks will be awarded for concise, efficient, well structured, well commented code behaving according to the specification. Individual marks may be adjusted within groups if contributions are widely different.

[2]Good comments give an overview of what the code does, as well as line-by-line information to help the reader understand the code. Generally the code file should start with an overview of what the code in that file is about, and a high level outline of what it is doing. Similarly each function should start with a description of its inputs outputs and purpose plus a brief outline of how it works. Line-by-line comments aim to make the code easier to understand in detail.

[3]all team members must have git installed and use it