Statistical computing MATH10093
Computer lab 2
Solutions

Finn Lindgren

23/1/2019

Summary

In this lab session you will explore optimisation methods, and practice writing R functions that can be supplied to the standard optimisation routine in R, in order to perform numerical parameter estimation. You will not hand in anything, but you should keep your code script file for later use. Note: This lab sheet contains supplementary course notes about optimisation and the R language.

1. Initialise:
   (a) Open `RStudio` with the project you created last week, and start a new code file to hold the code for this lab. During the lab, remember to save the script file regularly, to avoid losing any work.
   (b) Recall the options and keyboard shortcuts introduced in Lab 1.
   (c) Whenever you encounter a function you're not yet familiar with, use the help system to read the documentation!
2. Install the `FLtools` package from `https://bitbucket.org/finnlindgren/FLtools/` by, e.g., running

```
remotes::install_bitbucket("finnlindgren/FLtools", dep = TRUE)
```

The `remotes::install_bitbucket` syntax means that R will use the function `install_bitbucket` from the package `remotes`, without having to load all the functions into the global environment, as would be done with `library("remotes")`. In some cases, R/RStudio will detect that some related packages have newer versions, and ask if you want it to first upgrade those packages. For `FLtools`, keeping the existing versions of those packages should be fine.

The `FLtools` package includes a graphical interactive tool for exploring optimisation methods, based on the R interactive `shiny` system.

Note: If you're running on your own computer, you might not have the remotes package, in which case you need to install it first:

```
install.packages("remotes")
```

3. The optimisation methods discussed in the lecture (Gradient Descent, Newton and the Quasi-Newton method BFGS, see Lecture 2) are all based on starting from a current best estimate of a minimum, finding a search direction, and then performing a line search to find a new, better, estimate. The *Nelder-Mead Simplex* method works in a similar way, but doesn't use any derivatives of the target function; it only uses function evaluations, and keeps track of a set of local points ($m+1$ points, if m is the dimension of the problem). For 2D problems the method updates a triangle of points. In each iteration, it attempts to move away from the "worst" point, by performing a simple line search. In addition to the basis line search, it will reduce or expand the triangle. Expansion happens similarly to the adaptive step length in the Gradient Descent method.

   Start the `optimisation` shiny app:

   ```
   FLtools::optimisation()
   ```

   (a) For the "Simple (1D)" and "Simple (2D)" functions, familiarise yourself with the "Step", "Converge", and "Reset" buttons. Choose different optimisation starting points by clicking in the figure.

   (b) Explore the different optimisation methods and what they display in the figure for each optimisation step.

   - The simplex/triangle shapes are shown for each "Simplex" method step in blue. The "best" points for each simplex are connected (magenta).
   - The Newton methods display the true quadratic Taylor approximations (contours in red) as well as the approximations used to find the proposed steps (contours in blue).

   Also observe the diagnostic output box and how the number of function, gradient, and Hessian evaluations differ between the methods.

   (c) For the "Rosenbrock (2D)" function, observe the differencences in convergence behaviour for the four different optimisation methods.

   (d) For m-dimensional problems and derivatives approximated by finite differences, a gradient calculation costs at least m extra function evaluations, and a Hessian costs at least $2m^2$ extra function evaluations. For the 2D Rosenbrock function, count the total number of function evaluations required by each optimisation method (under the assumption that finite differences were used for the gradient and Hessian) until convergence is reached.

   Hint: find the "Evaluations:" information in the app. You may need to use the "Converge" button multiple times, since it will only do at most 500 iterations each time.

Also note the total number of iterations needed by each method. How do they compare?
Answer:
$m = 2$, so the number of function evaluations is given by
`#f + 2 * #gradient + 8 * #hessian`:

- Nelder-Mead: 202 (103 iterations)
- Gradient Descent: $14132+2 \cdot 9405 = 3.2942 \times 10^4$ (9404 iterations)
- Newton: $29 + 2 \cdot 23 + 8 \cdot 22 = 251$ (22 iterations)
- BFGS: $54 + 2 \cdot 41 + 8 \cdot 1 = 144$ (40 iterations)

BFGS uses the smallest number of function evaluations, which makes it faster than the more "exact" Newton method, despite needing almost twice as many iterations. Thus, we would normally only consider using the full Newton method if we have a more efficient way of obtaining the Hessian than finite differences. Gradient Descent does extremely badly on this test problem; clearly, using some approximate information about the second order derivatives can provide much better search directions and step lengths than using no higher order information. The Simplex method doesn't use higher order information, but due to its local "memory" it can still be competetive; in this test case, it outperforms the Newton method in terms of cost, but not the BFGS method.

(e) For the "Multimodal" functions, explore how the optimisation methods behave for different starting points.

(f) How far out can the optimisation start for the "Spiral" function? E.g., try the "Newton" method, starting in the top right corner of the figure.

4. A function in R can have a `name`, `parameters`, and a return `value`. A function is defined through

```
thename <- function(parameters) {
  expressions
}
```

where `expressions` is one or more lines of code. The result of the last evaluated expression in the function is the value returned to the caller.

Each input parameter can have a *default value*, so that the caller doesn't have to specify it unless they want a different value. Sometimes, a `NULL` default value is used, and the parameter checked internally with `is.null(parameter)` to determine if the user supplied something.

It is good practice to refer to the parameters by name when calling a function (instead of relying on the order of the parameters; the first parameter is a common exception to this practice), especially for parameters that have default values. Example:

```r
my_function <- function(param1, param2 = NULL, param3 = 4) {
  # 'if': run a block of code if a logical statement is true
  if (is.null(param2)) {
    param1 + param3
  } else {
    # 'else', companion to 'if':
    #    run this other code if the logical statement wasn't true
    param1 + param2 / param3
  }
}
my_function(1)

## [1] 5

my_function(1, param3 = 2)

## [1] 3

my_function(1, param2 = 8, param3 = 2)

## [1] 5

my_function(1, param2 = 8)

## [1] 3
```

The main optimisation function in R is `optim()`, which has the following call syntax:

```r
optim(par, fn, gr = NULL, ...,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN",
                 "Brent"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE)
```

Here, the `method` parameter appears to have a whole vector as its default value. However, this is merely a way to show the user all the permitted values. If the user does not supply anything specific, the first value, `"Nelder-Mead"` will be used.

You may have briefly encountered the special `...` parameter syntax in an earlier lab. It means that *there may be additional parameters specified here by the caller, that should be passed on to another function that is called internally.* For `optim()`, those extra parameters are passed on to the target function that the user supplies. A call to `optim()` to minimise the function defined by `myTargetFunction()` might take this form:

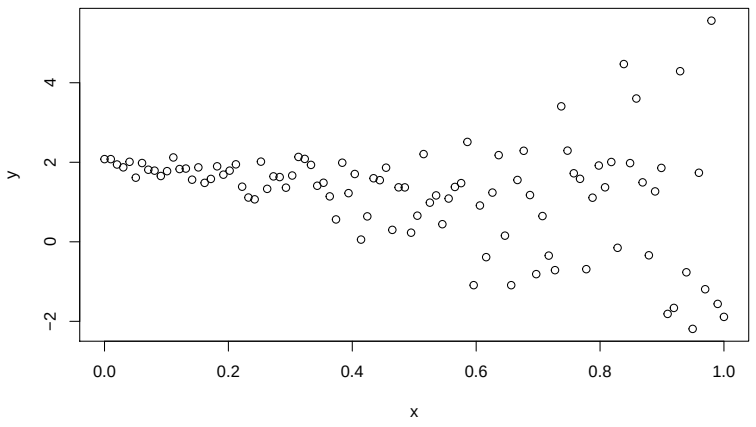
```
opt <- optim(par = start_par,
             fn = myTargetFunction, extra1 = value1, extra2 = value2)
```

When `optim()` uses the function `myTargetFunction()`, it will be called like this:

```
myTargetFunction(current_par, extra1 = value1, extra2 = value2)
```

(a) Use the following code to simulate synthetic random data from a model of the type mentioned in Lecture 2, where the standard deviation of the observation depends on two model parameters via covariates.

```
n <- 100
X <- cbind(1, seq(0, 1, length.out = n))
theta_true <- c(2, -2, -2, 3)
y <- rnorm(n = n,
           mean = X %*% theta_true[1:2],
           sd = exp(X %*% theta_true[3:4]))
plot(X[, 2], y, xlab="x")
```



The mathematical definition can be written

$$
\begin{aligned}
i &\in \{1, 2, \ldots, n\} \\
x_i &= \frac{i-1}{n-1} \\
\boldsymbol{X} &= \begin{bmatrix} \mathbf{1} & \boldsymbol{x} \end{bmatrix} \\
\boldsymbol{\mu} &= \boldsymbol{X} \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \\
\log(\sigma_i) &= \theta_3 + x_i \theta_4 \\
y_i &= \mathrm{N}(\mu_i, \sigma_i^2), \quad \text{(independent)}.
\end{aligned}
$$

(b) Write a function

```
neg_log_lik <- function(theta, y, X) {
  # Code goes here
}
```

that evalues the negative log-likelihood for the model. See `?dnorm`.

```
## Solution:
neg_log_lik <- function(theta, y, X) {
  -sum(dnorm(x = y,
             mean = X %*% theta[1:2],
             sd = exp(X %*% theta[3:4]),
             log = TRUE))
}
```

(c) With the aid of the help text for `optim()`, find the maximum likelihood parameter estimates for our statistical model using the BFGS method with numerical derivatives. Use $(0, 0, 0, 0)$ as the starting point for the optimisation.

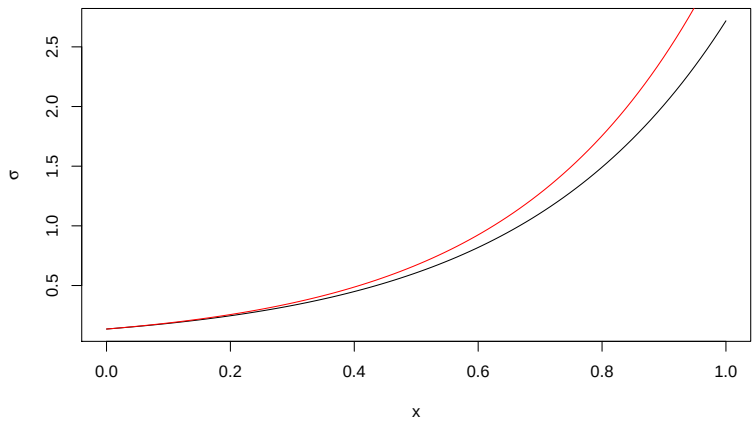
```
## Solution:
opt <- optim(par = c(0, 0, 0, 0),
             fn = neg_log_lik, y = y, X = X,
             method = "BFGS")
```

Check the `optim` help text for information about what the result object contains. Did the optimisation converge?

Answer:
The optimisation converged if `opt$convergence` is `0`. Or at least `optim()` *thinks* it coverged, since it triggered it's convergence tolerances.

The estimates of σ_i as functions of x_i should look similar to this:

```
plot(X[, 2], exp(X %*% theta_true[3:4]), type = "l",
     xlab = "x", ylab = expression(sigma))
lines(X[, 2], exp(X %*% opt$par[3:4]), col = 2)
```



The `expression(sigma)` y-axis label is a way of making R try to interpret an R expression and format it more like a mathematical expression, with greek letters, etc. For example,
`ylab = expression(theta[1] + x[i] * theta[2])`
would be formatted as $\theta_1 + x_i\theta_2$.

(d) Rerun the optimisation with the extra parameter `hessian = TRUE`, to obtain a numeric approximation to the Hessian of the target function at the optimum, and compute its inverse (see `?solve`), which is an estimate of the covariance matrix for the error of the parameter estimates. In the next Computer lab we will use this to evaluate approximate confidence and prediction intervals.

```
## Solution:
opt <- optim(par = c(0, 0, 0, 0),
             fn = neg_log_lik, y = y, X = X,
             method = "BFGS",
             hessian = TRUE)
covar <- solve(opt$hessian)
covar

##               [,1]         [,2]          [,3]          [,4]
## [1,]  0.0022827772 -0.007597505 -0.0004132743  0.0008264823
## [2,] -0.0075975049  0.050828380  0.0027647341 -0.0055289285
## [3,] -0.0004132743  0.002764734  0.0231597168 -0.0363194612
## [4,]  0.0008264823 -0.005528929 -0.0363194612  0.0726389957
```