MTH3045: Statistical Computing

Dr. Ben Youngman
b.youngman@exeter.ac.uk
Laver 817; ext. 2314

24/3/2025

Week 11 lecture 1

Root finding

- Consider some function $f(x)$ for $x \in \mathbb{R}$ and wanting to find the value of x, $\tilde{x}$ say, such that $f(\tilde{x}) = 0$
- Sometimes we can analytically find $\tilde{x}$, but sometimes not
- We'll just consider the latter case where we'll need to find $\tilde{x}$ numerically, such as through some iterative process
- We won't go into the details of root-finding algorithms; instead we'll just look at `R`'s function `uniroot()`
- This is `R`'s go-to function for root finding
- This chapter will just demonstrate its use by example

Example: Root-finding in R I

- Use `uniroot()` in `R` to find the root of

$$f(x) = (x + 3)(x - 1)^2$$

  i.e. to find $\tilde{x}$, where $f(\tilde{x}) = 0$, given that $\tilde{x} \in [-4, 4/3]$

Example: Root-finding in R II

- We'll start by writing a function to evaluate $f()$, which we'll call `f`

```
f <- function(x) (x + 3) * (x - 1)^2
```

- Then we'll call `uniroot()`

```
uniroot(f, c(-4, 4.3))
```

```
## $root
## [1] -2.999997
##
## $f.root
## [1] 4.501378e-05
##
## $iter
## [1] 7
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 6.103516e-05
```

Example: Root-finding in R III

- We see that its output includes various details
- Most important are
  - `root`, its estimate of $\tilde{x}$, which is $\tilde{x} \simeq -2.9999972$
  - `f.root`, the value of $f()$ at the root, i.e. $f(\tilde{x})$, which is $f(\tilde{x}) \simeq 4.5013782 \times 10^{-5}$
- We note that $f(\tilde{x})$ is sufficiently close to zero that we should be confident that we've reached a root

Example: Root-finding in R IV

- *Remark*: We can ask `uniroot()` to extend the search range for the root through its argument `extendInt`
- Options are `'no'`, `'yes'`, `'downX'` and `'upX'`, which correspond to not extending the range (the default) or allowing it to be extended to allow upward and downward crossings, just downward or just upward, respectively
- (If we want to extend the search interval for the root, `extendInt = 'yes'` is usually the best option. Otherwise, we need to think about how $f()$ behaves at the roots, i.e. whether it's increasing or decreasing. See the help file for `uniroot()` for more details.)
- If we return to the above example and consider the search range $[-2, -1]$ instead, then by issuing

```
uniroot(f, c(-2, -1), extendInt = 'yes')$root
```

```
## [1] -2.999991
```

we do still find the root, even though it's outside of our specified range.

Challenges I

- Go to Challenges I of the week 11 lecture 1 challenges at https://byoungman.github.io/MTH3045/challenges

One-dimensional optimisation in R

- We'll only briefly look at how we can perform one-dimensional optimisation in R, which is through its `optimize()` function
- As described by its help file, `optimize()` uses '*a combination of golden section search and successive parabolic interpolation, and was designed for use with continuous functions*'
- We can instead use `optimize()` by calling `optim(..., method = 'Brent')`
  - the two are equivalent
  - the only reason I can see for using `optim(..., method = 'Brent')` over `optimize()` is that `optim()` is R's preferred numerical optimisation function, and hence users my benefit from familiarity with its output, as opposed to that of `optimize()`
- By default `optim()` uses the Nelder-Mead polytope method, which we'll cover in Section 5.4.7, which doesn't usually work well in one dimension

Example: Numerical maximum likelihood estimation I

- Consider a sample of data $y_1, \ldots, y_n$ as independent realisations from the Exp(λ) distribution with pdf

$$f(y \mid \lambda) = \lambda \exp(-\lambda y) \qquad \text{for } y > 0$$

  where $\lambda > 0$ is an unknown parameter that we want to estimate
- Its mle is $1/\bar{y}$, where $\bar{y} = n^{-1} \sum_{i=1}^{n} y_i$
- Confirm this numerically in `R` using `optimize()` by assuming that the sample of data

$$0.4, 0.5, 0.8, 1.8, 2.1, 3.7, 8.2, 10.6, 11.6, 12.8$$

  are independent Exp(λ) realisations

Example: Numerical maximum likelihood estimation II

- By default `optimize()` will find the minimum, so we want to write a function that will evaluate the exponential distribution's log-likelihood

$$\log f(\mathbf{y} \mid \lambda) = n \log \lambda - \lambda \sum_{i=1}^{n} y_i$$

  and then negate it
- We'll call this `negloglik(lambda, y)`

```
negloglik <- function(lambda, y) {
  # Function to evaluate Exp(lambda) neg. log likelihood
  # lambda is a scalar
  # y can be scalar or vector
  # returns a scalar
  -n * log(lambda) + lambda * sum(y)
}
```

Example: Numerical maximum likelihood estimation III

- We then pass this on to `optimize()` with our sample of data, which we'll call `y`

```
y <- c(0.4, 0.5, 0.8, 1.8, 2.1, 3.7, 8.2, 10.6, 11.6, 12.8)
optimize(negloglik, lower = .1, upper = 10, y = y)
```

```
## $minimum
## [1] 0.1904839
##
## $objective
## [1] 26.58228
```

- We see that R's numerical maximum likelihood estimate of λ is 0.1904839, and the true value is $1/5.25 \simeq 0.1904762$
  - so the two agree to five decimal places
- *Remark 1*: We can ask `optimize()` to be more precise through its `tol` argument, which has default `tol = .Machine$double.eps^0.25`
  - smaller values of `tol` will give more accurate numerical estimates
- *Remark 2*: Calling `optimise()` is equivalent to calling `optimize()`, for those that don't like American spellings of English words.

Challenges I

- Go to Challenges I of the week 11 lecture 2 challenges at https://byoungman.github.io/MTH3045/challenges

Bibliographic notes

- By far the best resource for reach up on numerical optimisation is Nocedal and Wright (2006)
  - Chapter 3 covers Newton's method and line search
  - Chapter 6 covers quasi-Newton methods
  - Chapter 8 covers derivative-free optimisation, including the Nelder-Method in Section 9.5
- Optimisation is also covered in Monahan (2011, chap. 8) and in Wood (2015, sec. 5.1)
- Simulated annealing is covered in Press et al. (2007, sec. 10.12)
- Root-finding is covered in Monahan (2011, sec. 8.3) and Press et al. (2007, chap. 9)

Quasi-Newton methods I

- Between Newton's method and steepest descent lie **quasi-Newton** methods
- These essentially employ Newton's method, but with some approximation to the Hessian matrix
- Instead of the Newton's method search direction

$$\mathbf{p}_i = -\left[\nabla^2 f(\boldsymbol{\theta}_i)\right]^{-1} \nabla f(\boldsymbol{\theta}_i)$$

consider the search direction

$$\tilde{\mathbf{p}}_i = -\mathbf{H}_i^{-1} \nabla f(\boldsymbol{\theta}_i)$$

where $\mathbf{H}_i$ is an approximation to the Hessian matrix $\nabla^2 f(\boldsymbol{\theta}_i)$ at the ith iteration

Quasi-Newton methods II

- We might, for example, want to avoid explicitly calculating $\nabla^2 f(\boldsymbol{\theta}_i)$ because it's a matrix that's sufficiently more difficult to calculate than the gradient (e.g. mathematically, or just in terms of time)
  - so that using an approximation to the Hessian matrix (provided it is an adequate approximation) gives a more efficient approach to optimisation than using the Hessian matrix itself
- We should typically expect quasi-Newton methods to converge slower than Newton's method, but provided that convergence isn't too much slower or less reliable, then we may prefer this over analytically forming the Hessian matrix

BFGS (Broyden–Fletcher–Goldfarb–Shanno) I

- In MTH3045 we shall consider the so-called **BFGS** (shorthand for Broyden–Fletcher–Goldfarb–Shanno) quasi-Newton algorithm
- Put simply, at iteration i, the BFGS algorithm assumes that

$$\nabla^2 f(\boldsymbol{\theta}_i) \simeq \mathbf{H}_i = \mathbf{H}_{i-1} + \frac{\mathbf{y}_i \mathbf{y}_i^{\mathrm{T}}}{\mathbf{y}_i^{\mathrm{T}} \mathbf{s}_i} - \frac{(\mathbf{H}_{i-1})^{-1} \mathbf{s}_i \mathbf{s}_i^{\mathrm{T}} (\mathbf{H}_{i-1})^{-T}}{\mathbf{s}_i^{\mathrm{T}} (\mathbf{H}_{i-1})^{-1} \mathbf{s}_i},$$

  where $\mathbf{s}_i = \boldsymbol{\theta}_i - \boldsymbol{\theta}_{i-1}$ and $\mathbf{y}_i = \nabla f(\boldsymbol{\theta}_i) - \nabla f(\boldsymbol{\theta}_{i-1})$
- Hence the BFGS algorithm uses differences in the gradients of successive iterations to approximate the Hessian matrix
- We now note that we use $\mathbf{H}_i$ in $\mathbf{p}_i = -[\mathbf{H}_i]^{-1} \nabla f(\boldsymbol{\theta}_i)$
- We can avoid solving this system of linear equations by instead directly obtaining $[\mathbf{H}_i]^{-1}$ through

$$[\mathbf{H}_i]^{-1} = \left(\mathbf{I}_p - \frac{\mathbf{s}_i \mathbf{y}_i^{\mathrm{T}}}{\mathbf{y}_i^{\mathrm{T}} \mathbf{s}_i} \right) [\mathbf{H}_{i-1}]^{-1} \left(\mathbf{I}_p - \frac{\mathbf{y}_i \mathbf{s}_i^{\mathrm{T}}}{\mathbf{s}_i^{\mathrm{T}} \mathbf{y}_i} \right) + \frac{\mathbf{s}_i \mathbf{s}_i^{\mathrm{T}}}{\mathbf{y}_i^{\mathrm{T}} \mathbf{y}_i}$$

BFGS (Broyden–Fletcher–Goldfarb–Shanno) II

- The following R function updates $[\mathbf{H}_{i-1}]^{-1}$ to $[\mathbf{H}_i]^{-1}$ given $\boldsymbol{\theta}_i$, $\boldsymbol{\theta}_{i-1}$, $\nabla f(\boldsymbol{\theta}_{i-1})$ and $\nabla f(\boldsymbol{\theta}_i)$, which are the arguments x0, x1, g0 and g1, respectively

```
iH1 <- function(x0, x1, g0, g1, iH0) {
  # Function to update Hessian matrix
  # x0 and x1 are p-vectors of second to last and last estimates, respectively
  # g0 and g1 are p-vectors of second to last and last gradients, respectively
  # iH0 is previous estimate of p x p Hessian matrix
  # returns a p x p matrix
  s0 <- x1 - x0
  y0 <- g1 - g0
  denom <- sum(y0 * s0)
  I <- diag(rep(1, 2))
  pre <- I - tcrossprod(s0, y0) / denom
  post <- I - tcrossprod(y0, s0) / denom
  last <- tcrossprod(s0) / denom
  pre %*% iH0 %*% post + last
}
```

Example: Weibull maximum likelihood: BFGS I

- Repeat Example 5.5 using the BFGS method
- Comment on how it compares to Newton's method

Example: Weibull maximum likelihood: BFGS II

- The following code implements five iterations of the BFGS method

```
## This build of rgl does not include OpenGL functions.  Use
##  rglwidget() to display results, e.g. via options(rgl.printRglwidget = TRUE)
iterations <- 5
xx <- matrix(0, iterations + 1, 2)
dimnames(xx) <- list(paste('iter', 0:iterations), c('lambda', 'k'))
xx[1, ] <- c(1.6, .6)
g <- iH <- list()
for (i in 2:(iterations + 1)) {
  g[[i]] <- weib_d1(xx[i - 1, ], y0, mult = -1)
  if (sqrt(sum(g[[i]]^2)) < 1e-6)
    break
  if (i == 2) {
    iH[[i]] <- diag(1, 2)
  } else {
    iH[[i]] <- iH1(xx[i - 2, ], xx[i - 1, ], g[[i - 1]], g[[i]], iH[[i - 1]])
  }
  search_dir <- -(iH[[i]] %*% g[[i]])
 alpha <- line_search(xx[i - 1, ], search_dir, weib_d0, y = y0, mult = -1)
 xx[i, ] <- xx[i - 1,] + alpha * search_dir
}
```

Example: Weibull maximum likelihood: BFGS III

- Our estimates at each iteration are

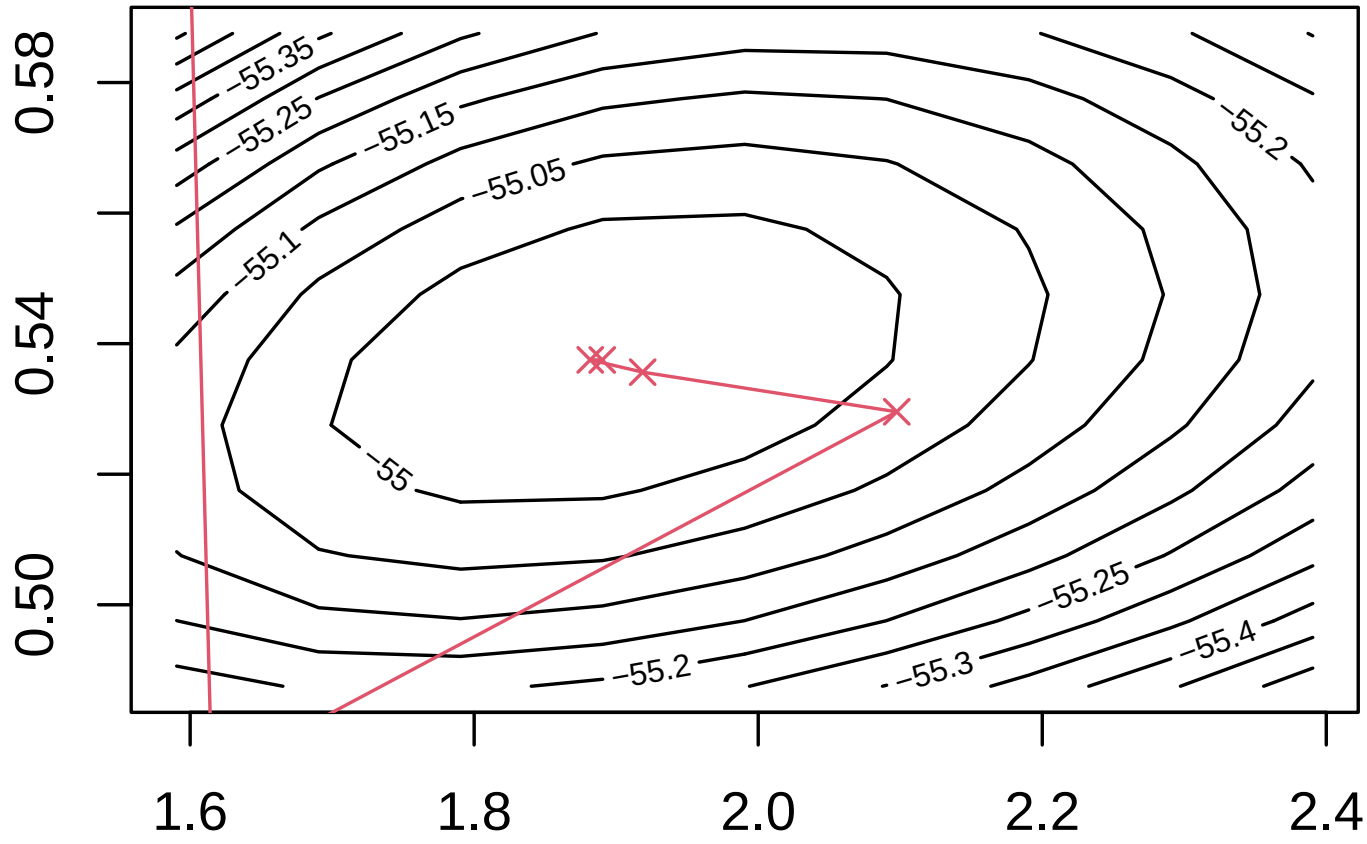
```
xx
```

```
##          lambda         k
## iter 0 1.600000 0.6000000
## iter 1 1.615241 0.4736904
## iter 2 2.097571 0.5295654
## iter 3 1.918661 0.5356343
## iter 4 1.881726 0.5375145
## iter 5 1.890167 0.5374986
```

  and we see that we need two more iterations than Newton's method to reach convergence to three decimal places

Example: Weibull maximum likelihood: BFGS IV

- Finally, we'll plot the course of the iterations



and we can see the slightly less direct route we've taken

Quasi-Newton methods in R I

- There are various options for performing quasi-Newton methods in R
- For these, we just need to supply the function to be minimised and its gradient
- The first option is to use `nlminb()` again
  - if we don't supply a function to evaluate the Hessian, then `nlminb()` uses a quasi-Newton approach
- The alternative, and possibly preferred option, is to use `optim()` with option `method = 'BFGS'`
- We'll repeat Example 5.5 using BFGS instead

Quasi-Newton methods in R II

- To use `nlminb()` we can use the following.

```
nlminb(c(1.6, .6), weib_d0, weib_d1, y = y0, mult = -1)
```

```
## $par
## [1] 1.8900689 0.5375279
##
## $objective
## [1] 54.95316
##
## $convergence
## [1] 0
##
## $iterations
## [1] 7
##
## $evaluations
## function gradient
##        9        8
##
## $message
## [1] "relative convergence (4)"
```

Quasi-Newton methods in R III

- To use optim() we can use the following.

```
optim(c(1.6, .6), weib_d0, weib_d1, y = y0, mult = -1, method = 'BFGS')
```

```
## $par
## [1] 1.8900632 0.5375283
##
## $value
## [1] 54.95316
##
## $counts
## function gradient
##       14        6
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Finite-difference checking I

```
fd <- function(x, f, delta = 1e-6, ...) {
  # Function to evaluate derivative w.r.t. vector by finite-differencing
  # x is a p-vector
  # fn is the function for which the derivative is being calculated
  # delta is the finite-differencing step, which defaults to 10^{-6}
  # returns a vector of length x
  f0 <- f(x, ...)
  p <- length(x)
  f1 <- numeric(p)
  for (i in 1:p) {
    ei <- replace(numeric(p), i, 1)
    f1[i] <- f(x + delta * ei, ...)
  }
  (as.vector(f1) - f0) / delta
}
```

Finite-difference checking II

- Finite-differencing can help us check a gradient function we've written

```
# Check a function we've written
weib_d1(c(1.6, .6), y = y0, mult = -1)
```

```
## [1] -1.950898 16.167624
```

```
fd(c(1.6, .6), weib_d0, y = y0, mult = -1)
```

```
## [1] -1.950895 16.167729
```

Finite-difference checking III

- Finite-differencing can also help us check whether we've reached a local minimum

```
# Check optima
## Nelder-Mead
p_nm <- optim(c(1.6, .6), weib_d0, y = y0, mult = -1)$par
weib_d1(p_nm, y = y0, mult = -1)
```

```
## [1]  0.0004811097 -0.0116608005
```

```
fd(p_nm, weib_d0, y = y0, mult = -1)
```

```
## [1]  0.0004823661 -0.0115609069
```

```
## BFGS
p_bfgs <- optim(c(1.6, .6), weib_d0, weib_d1, y = y0, mult = -1, method = 'BFGS
weib_d1(p_bfgs, y = y0, mult = -1)
```

```
## [1] -1.677935e-05  1.121945e-04
```

```
fd(p_bfgs, weib_d0, y = y0, mult = -1)
```

```
## [1] -1.552536e-05  2.120899e-04
```

Quasi-Newton methods in R IV

- We note `nlminb()` and `optim()` have essentially given the same value of `par`, i.e. for the $\hat{\lambda}$ and $\hat{k}$, which is reassuring
- Note that `nlminb()` has used fewer function evaluations than `optim()`
  - we won't go into the details of the cause of this, but it is worth noting that the functions use different stopping criteria, and slightly different variants of the BFGS algorithm
- Note also that `nlminb()` has used three more function evaluations with the BFGS method than with Newton's method
  - this is is typically the case, and reflects the improved convergence achieved by using the actual Hessian matrix with Newton's method, as opposed to the approximation that's used with the BFGS approach

Exam tips

1. If you're unsure whether your derivative function is correct, why not use finite-differencing to check it? Having generic finite-differencing functions to hand, such as `fd()` in the lecture notes, could be very helpful. Finite-differencing can also be used to approximate Hessian matrices.
2. Balance correcting code with moving on to the next question. If your code has a small error that's causing it to not run properly, then you may still pick up marks for a partially correct answer.

References

Monahan, John F. 2011. *Numerical Methods of Statistics*. 2nd ed. Cambridge University Press. https://doi.org/10.1017/CBO9780511977176.

Nocedal, J., and S. Wright. 2006. *Numerical Optimization*. 2nd ed. Springer Series in Operations Research and Financial Engineering. Springer New York. https://books.google.co.uk/books?id=VbHYoSyelFcC.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge University Press. https://books.google.co.uk/books?id=1aAOdzK3FegC.

Wood, Simon N. 2015. *Core Statistics*. Institute of Mathematical Statistics Textbooks. Cambridge University Press. https://doi.org/10.1017/CBO9781107741973.