MTH3045: Statistical Computing

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

19/3/2024

Week 10 lecture 1

Week 10 lecture 1

5.4 Newton's multi-dimensional method

Taylor's theorem (multivariate) I

- Taylor's theorem extends to the multivariate case, so that

$$f(\boldsymbol{\theta}) \simeq f(\boldsymbol{\theta}_0) + \left[\nabla f(\boldsymbol{\theta}_0)\right]^{\mathsf{T}} (\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^{\mathsf{T}} \left[\nabla^2 f(\boldsymbol{\theta}_0)\right] (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

can be re-written as

$$f(\boldsymbol{\theta} + \boldsymbol{\Delta}) \simeq f(\boldsymbol{\theta}) + \left[\nabla f(\boldsymbol{\theta})\right]^{\mathsf{T}} \boldsymbol{\Delta} + \frac{1}{2}\boldsymbol{\Delta}^{\mathsf{T}} \left[\nabla^2 f(\boldsymbol{\theta})\right] \boldsymbol{\Delta} \qquad (5.1)$$

As similar argument to that of one dimension, i.e. finding $\boldsymbol{\Delta}$ that minimises equation (5.1), gives

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

which leads us to the iterative equation

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

for which we require that $\nabla^2 f(\boldsymbol{\theta}_i)$ is positive semi-definite in order to ensure that $f(\boldsymbol{\theta}_{i+1}) \leq f(\boldsymbol{\theta}_i)$

Taylor's theorem (multivariate) II

- *Remark 1*: For the multi-dimensional case of Newton's method, we will refer to $\mathbf{p}_i = -\left[\nabla^2 f(\boldsymbol{\theta}_i)\right]^{-1} \nabla f(\boldsymbol{\theta}_i)$ as the **Newton step**
- *Remark 2*: Sometimes it may turn out that $\nabla^2 f(\boldsymbol{\theta}_i)$ is not positive semi-definite
- Fortunately, this does not prohibit use of Newton's method because we can *perturb* $\nabla^2 f(\boldsymbol{\theta}_i)$ so that it is positive semi-definite, which will then guarantee that $f(\boldsymbol{\theta}_{i+1}) \leq f(\boldsymbol{\theta}_i)$
- There are various options for perturbation, but a common choice is to use $\nabla^2 f(\boldsymbol{\theta}_i) + \gamma \mathbf{I}_p$, where $\mathbf{I}_p$ is the $p \times p$ identity matrix, and we choose γ very small, and sequentially increase its value until $\nabla^2 f(\boldsymbol{\theta}_i) + \gamma \mathbf{I}_p$ is positive semi-definite
- For example, we might proceed through $\gamma = 10^{-12}, 10^{-10}, \ldots$

Example: Weibull maximum likelihood: Newton's method I

- The Weibull distribution is sometimes used to model wind speeds
- For a wind speed y its pdf is given by

$$f(y \mid \lambda, k) = \frac{k}{\lambda} \left(\frac{y}{\lambda} \right)^{k-1} e^{-(y/\lambda)^k} \quad \text{for } y > 0$$

  and where $\lambda, k > 0$ are its parameters
- (Note that this is the scale parameterisation of the Weibull distribution)
- For observed independent wind speeds $y_1, \ldots, y_n$ its corresponding log-likelihood is therefore

$$\log f(\mathbf{y} \mid \lambda, k) = n \log k - nk \log \lambda + (k-1) \sum_{i=1}^{n} \log y_i - \sum_{i=1}^{n} \left(\frac{y_i}{\lambda} \right)^k$$

Example: Weibull maximum likelihood: Newton's method II

- To implement Newton's method, we need to find the first and second derivatives of $\log f(\mathbf{y} \mid \lambda, k)$ w.r.t. λ and k
- The first derivatives are

$$\begin{pmatrix} \dfrac{\partial \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda} \\ \dfrac{\partial \log f(\mathbf{y} \mid \lambda, k)}{\partial k} \end{pmatrix} = \begin{pmatrix} \dfrac{k}{\lambda}\left(\sum_{i=1}^{n}\left(\dfrac{y_i}{\lambda}\right)^k - n\right) \\ \dfrac{n}{k} - n\log\lambda + \sum_{i=1}^{n}\log y_i - \sum_{i=1}^{n}\left[\left(\dfrac{y_i}{\lambda}\right)^k \log\left(\dfrac{y_i}{\lambda}\right)\right] \end{pmatrix}$$

Example: Weibull maximum likelihood: Newton's method III

- ...and the second derivatives are stored in the matrix

$$\begin{pmatrix} \dfrac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda^2} & \dfrac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda \partial k} \\ \dfrac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial k \partial \lambda} & \dfrac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial k^2} \end{pmatrix}$$

where

$$\frac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda^2} = \frac{k}{\lambda^2} \left(n - (1+k) \sum_{i=1}^{n} \left(\frac{y_i}{\lambda} \right)^k \right)$$

$$\frac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda \partial k} = \frac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial k \partial \lambda}$$

$$= \frac{1}{\lambda} \left(\sum_{i=1}^{n} \left(\frac{y_i}{\lambda} \right)^k - n + k \sum_{i=1}^{n} \left[\left(\frac{y_i}{\lambda} \right)^k \log \left(\frac{y_i}{\lambda} \right) \right] \right)$$

$$\frac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial k^2} = -\frac{n}{k^2} - \sum_{i=1}^{n} \left(\frac{y_i}{\lambda} \right)^k \left[\log \left(\frac{y_i}{\lambda} \right) \right]^2$$

Example: Weibull maximum likelihood: Newton's method IV

- Consider the following wind speed measurements (in m/s) for the month of March

```
y0 <- c(3.52, 1.95, 0.62, 0.02, 5.13, 0.02, 0.01, 0.34, 0.43, 15.5,
        4.99, 6.01, 0.28, 1.83, 0.14, 0.97, 0.22, 0.02, 1.87, 0.13, 0.01,
        4.81, 0.37, 8.61, 3.48, 1.81, 37.21, 1.85, 0.04, 2.32, 1.06)
```

- Use five iterations of Newton's method to estimate $\hat{\lambda}$ and $\hat{k}$, assuming the above wind speeds are independent from one day to the next and follow a Weibull distribution

Example: Weibull maximum likelihood: Newton's method V

- We'll start by plotting the log-likelihood surface
- We wouldn't normally do this, but it can be useful to quickly judge whether the log-likelihood surface is well-behaved, such as being unimodal and approximately quadratic about its maximum

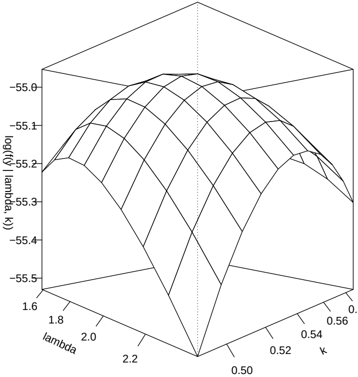


Figure 1: Log-likelihood surface of Weibull distribution model for wind speed data with different λ and k values.

Example: Weibull maximum likelihood: Newton's method VI

- Then we'll write functions `weib_d1` and `weib_d2` to evaluate the first and second derivatives of the Weibull distribution's log-likelihood w.r.t. λ and k
- We'll create these as functions of...
  - `pars`, the vector of parameters
  - `y` the vector of data
  - `mult`, a multiplier of the final log-likelihood, which defaults to `1`
- Introducing `mult` makes it much simpler when we later need the negative log-likelihood, as we don't have to write separate functions
- Always make sure that these functions take the same arguments

Example: Weibull maximum likelihood: Newton's method VII

- Often when calculating derivatives w.r.t. multiple parameters, we find that calculations are repeated
- It is worth avoiding repetition, and instead storing the results of any calculations that are used multiple times as objects. We'll do this in the next few lines of code, storing the re-used objects as z1, z2, etc.

```
weib_d1 <- function(pars, y, mult = 1) {
  # Function to evaluate first derivative of Weibull log-likelihood
  # pars is a vector
  # y can be scalar or vector
  # mult is a scalar defaulting to 1; so -1 returns neg. gradient
  # returns a vector
  n <- length(y)
  z1 <- y / pars[1]
  z2 <- z1^pars[2]
  out <- numeric(2)
  out[1] <- (sum(z2) - n) * pars[2] / pars[1] # derivative w.r.t. lambda
  out[2] <- n * (1 / pars[2] - log(pars[1])) +
    sum(log(y)) - sum(z2 * log(z1)) # w.r.t k
  mult * out
}
```

Example: Weibull maximum likelihood: Newton's method VIII

```
weib_d2 <- function(pars, y, mult = 1) {
  # Function to evaluate second derivative of Weibull log-likelihood
  # pars is a vector
  # y can be scalar or vector
  # mult is a scalar defaulting to 1; so -1 returns neg. Hessian
  # returns a matrix
  n <- length(y)
  z1 <- y / pars[1]
  z2 <- z1^pars[2]
  z3 <- sum(z2)
  z4 <- log(z1)
  out <- matrix(0, 2, 2)
  out[1, 1] <- (pars[2] / pars[1]^2) * (n - (1 + pars[2]) * z3) # w.r.t. (lambd
  out[1, 2] <- out[2, 1] <- (1 / pars[1]) * ((z3 - n) +
    pars[2] * sum(z2 * z4)) # w.r.t. (lambda, k)
  out[2, 2] <- -n/pars[2]^2 - sum(z2 * z4^2) # w.r.t. k^2
  mult * out
}
```

Example: Weibull maximum likelihood: Newton's method IX

- Next we'll perform five iterations of Newton's method, although we see that reasonable convergence is achieved after two or three iterations

```
iterations <- 5
xx <- matrix(0, iterations + 1, 2)
dimnames(xx) <- list(paste('iter', 0:iterations), c('lambda', 'k'))
lk0 <- c(1.6, .6)
xx[1, ] <- lk0
for (i in 2:(iterations + 1)) {
  gi <- weib_d1(xx[i - 1, ], y0)
  Hi <- weib_d2(xx[i - 1, ], y0)
  xx[i, ] <- xx[i - 1,] - solve(Hi, gi)
}
xx
```

```
##          lambda         k
## iter 0 1.600000 0.6000000
## iter 1 1.712945 0.5328618
## iter 2 1.866832 0.5375491
## iter 3 1.889573 0.5375304
## iter 4 1.890069 0.5375279
## iter 5 1.890069 0.5375279
```

Example: Weibull maximum likelihood: Newton's method X

- Finally we can plot the course of the iterations, and see that Newton's method quickly homes in on the log-likelihood surface's maximum

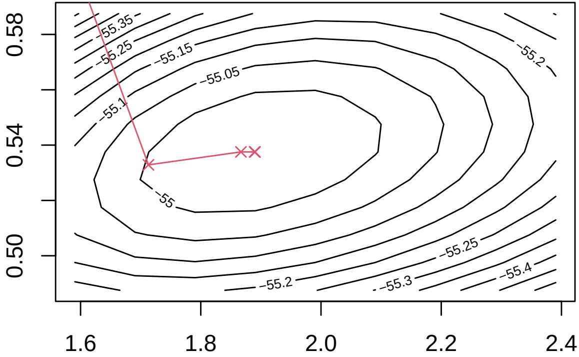


Figure 2: Five iterations of Newton's method to find Weibull maximum likelihood estimates.

Newton's method in `R` I

- We've just put together some simple code that implemented Newton's method
- There are various ways of performing variants of Newton's method in `R`, but not Newton's method itself
- So here we'll look at `nlminb()`, which is described as 'Unconstrained and box-constrained optimization using PORT routines'
- We can use our functions `weib_d1` and `weib_d2` from earlier for the first and second derivatives of the negative log-likelihood w.r.t. λ and k

Newton's method in R II

- We now just need a function to evaluate the negative log-likelihood itself
- We'll call this `weib_d0`
- Note, though, that it's important to ensure that invalid parameters, i.e. $\lambda \leq 0$ and/or $k \leq 0$, are avoided
- Below we achieve this by setting the log-likelihood to be extremely low (-10^8) for such parts of parameter space

```
weib_d0 <- function(pars, y, mult = 1) {
  # Function to evaluate Weibull log-likelihood
  # pars is a vector
  # y can be scalar or vector
  # mult is a scalar defaulting to 1; so -1 returns neg. log likelihood
  # returns a scalar
  n <- length(y)
  if (min(pars) <= 0) {
    out <- -1e8
  } else {
    out <- n * (log(pars[2]) - pars[2] * log(pars[1])) +
    (pars[2] - 1) * sum(log(y)) - sum((y / pars[1])^pars[2])
  }
  mult * out
}
```

Newton's method in R II

```r
nlminb(c(1.6, .6), weib_d0, weib_d1, weib_d2, y = y0, mult = -1)
```

```
## $par
## [1] 1.8900689 0.5375279
##
## $objective
## [1] 54.95316
##
## $convergence
## [1] 0
##
## $iterations
## [1] 5
##
## $evaluations
## function gradient
##        6        6
##
## $message
## [1] "relative convergence (4)"
```

Newton's method in R III

- We see that nlminb's output is a list comprising...
  - par, the parameter estimates
  - objective, the final value of the negative log-likelihood
  - convergence, whether the algorithm has converged (where 0 indicates successful convergence)
  - iterations, the number iterations before convergence was achieved
  - evaluations, how many times the function and gradient were evaluated
  - message provides further details on the type of convergence achieved

Challenges I

- Go to Challenges I of the week 10 lecture 1 challenges at https://byoungman.github.io/MTH3045/challenges

Week 10 lecture 2

Gradient descent

- If we consider small $\boldsymbol{\Delta}$ in (5.1) then we get the first order approximation

$$f(\boldsymbol{\theta} + \boldsymbol{\Delta}) \simeq f(\boldsymbol{\theta}) + [\nabla f(\boldsymbol{\theta})]^{\mathsf{T}} \boldsymbol{\Delta},$$

  which is appropriate for small $\boldsymbol{\Delta}$
- The concept behind gradient descent is simple: we want to minimise $[\nabla f(\boldsymbol{\theta})]^{\mathsf{T}} \boldsymbol{\Delta}$, which requires that we follow the direction of $-\nabla f(\boldsymbol{\theta})$
- To allow for different magnitudes of gradient, we will choose

$$\boldsymbol{\Delta} = -\frac{\nabla f(\boldsymbol{\theta})}{||\nabla f(\boldsymbol{\theta})||}$$

- Now that we know the direction in which we want to head, we need to know how far in that direction we should go. For this we'll consider some $\alpha > 0$, so that

$$\begin{aligned} f(\boldsymbol{\theta} + \boldsymbol{\Delta}) &\simeq f(\boldsymbol{\theta}) - \alpha \frac{[\nabla f(\boldsymbol{\theta})]^{\mathsf{T}} [\nabla f(\boldsymbol{\theta})]}{||\nabla f(\boldsymbol{\theta})||}, \\ &= f(\boldsymbol{\theta}) - \alpha ||\nabla f(\boldsymbol{\theta})||, \end{aligned}$$

  which means that $\boldsymbol{\Delta} = -\nabla f(\boldsymbol{\theta})/||\nabla f(\boldsymbol{\theta})||$ brings a decrease in $f(\boldsymbol{\theta} + \boldsymbol{\Delta})$ that's proportional to $||\nabla f(\boldsymbol{\theta})||$ for $\alpha > 0$, and is the fastest possible rate at which $f(\boldsymbol{\theta} + \boldsymbol{\Delta})$ can decrease

Example: Weibull maximum likelihood: gradient descent I

- Repeat Example 5.5 using gradient descent with $\alpha = 0.5$ and $\alpha = 0.1$, using 30 iterations for each
- Comment on how these compare to each other, and to Newton's method

```
alpha_seq <- c(.5, .1)
iterations <- 30
for (j in 1:length(alpha_seq)) {
  xx2 <- matrix(0, iterations + 1, 2)
  dimnames(xx2) <- list(paste('iter', 0:iterations), c('lambda', 'k'))
  xx2[1, ] <- lk0
  for (i in 2:(iterations + 1)) {
    gi <- weib_d1(xx2[i - 1, ], y0, mult = -1)
    gi <- gi / sqrt(crossprod(gi)[1, 1])
    xx2[i, ] <- xx2[i - 1,] - alpha_seq[j] * gi
  }
}
```

Example: Weibull maximum likelihood: gradient descent II

- Here's a plot of the iterations

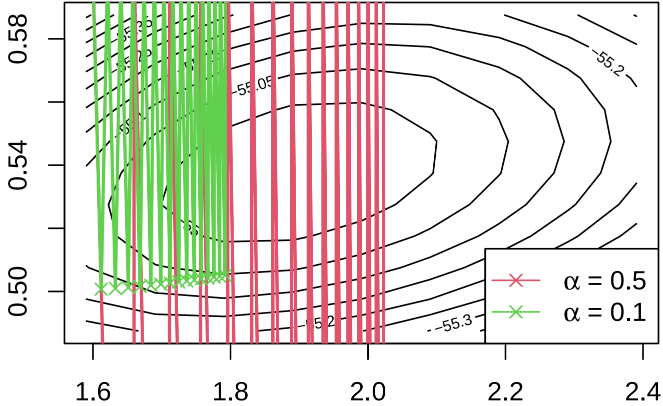


Figure 3: Iterations of the gradient descent algorithm with $\alpha = 0.5$ and $\alpha = 0.1$.

Example: Weibull maximum likelihood: gradient descent III

- In the above example we see that convergence towards $\hat{\lambda}$ and $\hat{k}$ is slow and has not been achieved after 30 iterations of $\alpha = 0.5$ and $\alpha = 0.1$, whereas Newton's method had essentially converged after four or five iterations
- Worse still, if we allowed more iterations, we'd see that both eventually converge, but away from $\hat{\lambda}$ and $\hat{k}$, as opposed to converging

Line search I

- Above we see that, for fixed α, gradient descent has diverged, i.e. not homed in on the minimum of $f()$
- This often happens with gradient descent
- A solution, which also applies to Newton's method, is to use a *line search*
- Consider Newton's method and a search direction of $\mathbf{p}_i = -\left[\nabla^2 f(\boldsymbol{\theta}_i)\right]^{-1} \nabla f(\boldsymbol{\theta}_i)$
- We want $f(\boldsymbol{\theta}_i + \mathbf{p}_i) < f(\boldsymbol{\theta}_i)$ in order for $\boldsymbol{\theta}_i + \mathbf{p}_i$ to be an improvement on $\boldsymbol{\theta}_i$
- If we employ a line search, we instead consider $\boldsymbol{\theta}_i + \alpha\mathbf{p}_i$ for some $\alpha > 0$ and ideally want α to minimise $f(\boldsymbol{\theta}_i + \alpha\mathbf{p}_i)$

Line search II

- In practice line search can be done informally, through the following process

1. Choose an initial value for α, α_0, say, and set $j = 0$
2. Evaluate $f(\boldsymbol{\theta}_i + \alpha_j \mathbf{p}_i)$
3. Set $j = j + 1$
4. Set $\alpha_j = \rho\alpha_{j-1}$, for $0 < \rho < 1$
5. Evaluate $f(\boldsymbol{\theta}_i + \alpha_j \mathbf{p}_i)$
6. If $f(\boldsymbol{\theta}_i + \alpha_j \mathbf{p}_i) < f(\boldsymbol{\theta}_i + \alpha_{j-1} \mathbf{p}_i)$, repeat steps 3 to 6 until $f(\boldsymbol{\theta}_i + \alpha_j \mathbf{p}_i) \geq f(\boldsymbol{\theta}_i + \alpha_{j-1} \mathbf{p}_i)$
7. Choose $\alpha = \alpha_{j-1}$

Line search III

- We can implement this in R

```r
line_search <- function(theta, p, f, alpha0 = 1, rho = .5, ...) {
  best <- f(theta, ...)
  cond <- TRUE
  while (cond & alpha0 > .Machine$double.eps) {
    prop <- f(theta + alpha0 * p, ...)
    cond <- prop >= best
    if (!cond)
      best <- prop
    alpha0 <- alpha0 * rho
  }
  alpha <- alpha0 / rho
  alpha
}
```

- *Remark*: Notice the use of the `...` argument here, which passes any extra arguments given to `line_search()` on to `f()`, and hence avoids the need to include `f()`'s arguments in `line_search()`
- This is useful because it makes `line_search()` applicable to any `f()`

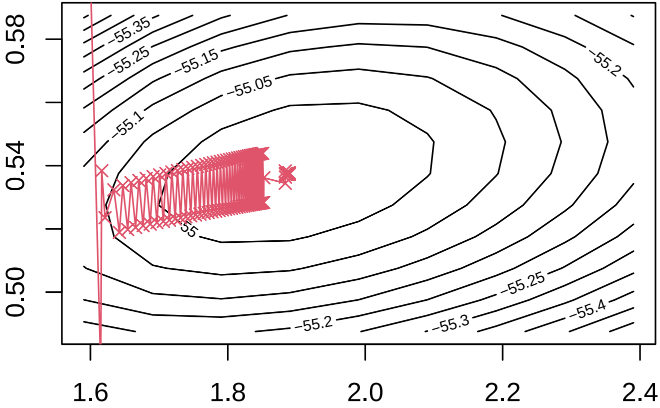
Example: Weibull maximum likelihood: gradient descent with line search I

- Repeat Example 5.5 using gradient descent but with line search and 200 iterations.

```r
iterations <- 200
xx2 <- matrix(0, iterations + 1, 2)
dimnames(xx2) = list(paste('iter', 0:iterations), c('lambda', 'k'))
xx2[1, ] <- c(1.6, .6)
for (i in 2:(iterations + 1)) {
  gi <- weib_d1(xx2[i - 1, ], y0, mult = -1)
  gi <- gi / sqrt(crossprod(gi)[1, 1])
  alpha_i <- line_search(xx2[i - 1, ], -gi, weib_d0, y = y0, mult = -1)
  xx2[i, ] <- xx2[i - 1,] - alpha_i * gi
}
```

Example: Weibull maximum likelihood: gradient descent with line search I

- We see that line search does at least bring us convergence of the parameter estimates, but that it's also very slow



Line search: Wolfe conditions

- *Remark*: So far we've adopted an informal approach to line search
- A more formal approach is to choose α so that it satisfies the **Wolfe conditions**
- A step length α_k is said to satisfy the Wolfe conditions, restricted to the direction $\mathbf{p}_i$, if the following two inequalities hold:

$$\textbf{i)} \quad f(\boldsymbol{\theta}_i + \alpha_i \mathbf{p}_i) \leq f(\boldsymbol{\theta}_i) + c_1 \alpha_i \mathbf{p}_i^{\mathrm{T}} \nabla f(\boldsymbol{\theta}_i),$$

$$\textbf{ii)} \quad -\mathbf{p}_i^{\mathrm{T}} \nabla f(\boldsymbol{\theta}_i + \alpha_i \mathbf{p}_i) \leq -c_2 \mathbf{p}_i^{\mathrm{T}} \nabla f(\boldsymbol{\theta}_i),$$

  with $0 < c_1 < c_2 < 1$. c_1 is usually chosen to be quite small while c_2 is much larger
- Nocedal and Wright (2006, sec. 6.1) give example values of $c_1 = 10^{-4}$ and $c_2 = 0.9$ for Newton or quasi-Newton methods

Week 10 lecture 3

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

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

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

```r
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

```
iterations <- 5
xx <- matrix(0, iterations + 1, 2)
dimnames(xx) <- list(paste('iter', 0:iterations), c('lambda', 'k'))
lk0 <- c(min(ls), max(ks))
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

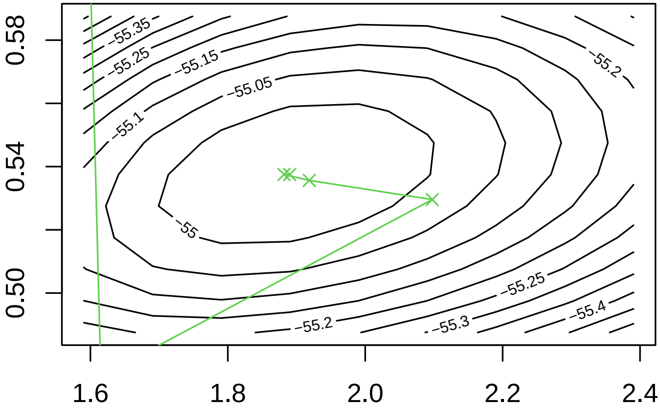
```
xx
```

```
##          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.

```r
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.

```r
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
```

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

Challenges I

- Go to Challenges I of the week 10 lecture 3 challenges at https://byoungman.github.io/MTH3045/challenges

Nelder-Mead polytope method I

- So far we have considered derivative-based optimisation algorithms
- When we cannot analytically calculate derivatives, we can use finite-difference approximations
- However, sometimes we may want to find the minimum point on a surface for a surface that is not particularly smooth
- Then derivative information may not be helpful
- Instead, we might want an algorithm that explores a surfaces differently
- The Nelder-Mead polytope algorithm is one such approach (Nelder and Mead (1965))
- In fact, it is `R`'s default if we use `optim()`, i.e. if we don't supply `method = '...'`

Nelder-Mead polytope method II

- For the Nelder-Mead algorithm, consider $\boldsymbol{\theta} \in \mathbb{R}^p$. The algorithm starts with $p+1$ test points, $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{p+1}$, which we call *vertices*, and then proceeds as follows. . .

1. Compute the order statistics of $f(\boldsymbol{\theta}_1), \ldots, f(\boldsymbol{\theta}_{p+1})$ vertices, i.e. find the order
$$f(\boldsymbol{\theta}^{(1)}) \leq f(\boldsymbol{\theta}^{(2)}) \leq \ldots \leq f(\boldsymbol{\theta}^{(p+1)})$$
and check whether the termination criteria have been met (which are given later). If not, proceed to Step 2.
2. Calculate the centroid, $\boldsymbol{\theta}_o$, of $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_p$, i.e. omitting $\boldsymbol{\theta}_{p+1}$, because $\boldsymbol{\theta}_{p+1}$ is the worst vertex.
3. Reflection. Compute the reflected point $\boldsymbol{\theta}_r = \boldsymbol{\theta}_o + \alpha(\boldsymbol{\theta}_o - \boldsymbol{\theta}_{p+1})$. If $f(\boldsymbol{\theta}_1) \leq f(\boldsymbol{\theta}_r) < f(\boldsymbol{\theta}_p)$, replace $\boldsymbol{\theta}_{p+1}$ with $\boldsymbol{\theta}_r$ and return to Step 1. Otherwise, proceed to Step 4.

Nelder-Mead polytope method III

4. Expansion. If $f(\boldsymbol{\theta}_r) < f(\boldsymbol{\theta}_1)$, i.e. is the best point so far, compute the expanded point $\boldsymbol{\theta}_e = \boldsymbol{\theta}_o + \gamma(\boldsymbol{\theta}_r - \boldsymbol{\theta}_o)$ for $\gamma > 1$. If $f(\boldsymbol{\theta}_e) < f(\boldsymbol{\theta}_r)$, replace $\boldsymbol{\theta}_{p+1}$ with $\boldsymbol{\theta}_r$ and return to Step 1. Otherwise, replace $\boldsymbol{\theta}_{p+1}$ with $\boldsymbol{\theta}_r$ and return to Step 1.

5. Contraction. Now $f(\boldsymbol{\theta}_r) \geq f(\boldsymbol{\theta}_p)$. Compute the contracted point $\boldsymbol{\theta}_c = \boldsymbol{\theta}_o + \rho(\boldsymbol{\theta}_{p+1} - \boldsymbol{\theta}_o)$ for $0 < \rho \leq 0.5$. If $f(\boldsymbol{\theta}_c) < f(\boldsymbol{\theta}_{p+1})$ then replace $\boldsymbol{\theta}_{p+1}$ with $\boldsymbol{\theta}_c$ and return to Step 1. Otherwise proceed to Step 6.

6. Shrink. For $i = 2, \ldots, p+1$ set $\boldsymbol{\theta}_i = \boldsymbol{\theta}_1 + \sigma(\boldsymbol{\theta}_i - \boldsymbol{\theta}_1)$ and return to Step 1.

Nelder-Mead polytope method IV

- Often the values $\alpha = 1$, $\gamma = 2$, $\rho = 0.5$ and $\sigma = 0.5$ are used
- In Step 1, termination is defined in terms of tolerances
- The main criterion is for $f(\boldsymbol{\theta}_{p+1}) - f(\boldsymbol{\theta}_1)$ to be sufficiently small, so that $f(\boldsymbol{\theta}_i)$ for $i = 1, \ldots, p+1$ are close together for all θ_i
    - hence we are hoping that *all* the $\boldsymbol{\theta}_i$ values are in the region of the true minimum
- We won't code the Nelder-Mead algorithm in `R`; instead we'll just use `optim()` and look what it does, by requesting a trace with `control = list(trace = TRUE)`

Example: Weibull maximum likelihood: Nelder-Mead I

- Use the Nelder-Mead method and `R`'s `optim()` function to find the maximum likelihood estimates of the Weibull distribution for the data of Example 5.5
- We've got everything we need for this example, i.e. the data, which we stored earlier as `y0`, and a function to evaluate the Weibull distribution's log-likelihood, `weib_d0()`
- So we just pass these to `optim()`, ensuring that `mult = -1`, so that we find the negative log-likelihood's minimum

Example: Weibull maximum likelihood: Nelder-Mead II

```r
fit_nelder <- optim(c(1.6, .6), weib_d0, y = y0, mult = -1,
                    control = list(trace = TRUE))
```

```
##   Nelder-Mead direct search function minimizer
## function value for initial parameters = 55.509247
##   Scaled convergence tolerance is 8.27152e-07
## Stepsize computed as 0.159049
## BUILD              3 60.369866 55.293827
## LO-REDUCTION       5 55.509247 55.045750
## REFLECTION         7 55.293827 55.038084
## HI-REDUCTION       9 55.045750 54.993703
## REFLECTION        11 55.038084 54.962562
## HI-REDUCTION      13 54.993703 54.961598
## ** quite a few lines suppressed **
## HI-REDUCTION      41 54.953162 54.953159
## HI-REDUCTION      43 54.953160 54.953159
## Exiting from Nelder Mead minimizer
##     45 function evaluations used
```

- The above output is telling us what `optim()` is doing as it's going along
- Specifically, `LO-REDUCTION` corresponds to a contraction, `HI-REDUCTION` to an expansion and `REFLECTION` to a reflection
- On this occasion, there was no need to shrink the simplex, which is identified as `SHRINK`

Example: Weibull maximum likelihood: Nelder-Mead III

```
fit_nelder
```

```
## $par
## [1] 1.8900970 0.5374706
##
## $value
## [1] 54.95316
##
## $counts
## function gradient
##       43       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

- What `optim()` returns is essentially the same as we saw before for the BFGS method, i.e. roughly the same parameter estimates and function minimum, except that now there are no gradient evaluations

Example: Weibull maximum likelihood: Nelder-Mead IV

- *Remark*: The Nelder-Mead method is usually great if you're quickly looking to find a function's minimum with its gradient
  - provided it's a relatively low-dimensional function, and the function is fairly quick to evaluate
  - this is probably why it's `R`'s default
- It can be very slow for high-dimensional optimisation problems, and is also typically less reliable than gradient-based methods, except for strangely-behaved surfaces

Challenge I

1. Consider the alternative parameterisation of the Weibull pdf in terms of parameters $(\log \lambda, \log k)$ given by

$$f(y \mid \tilde{\lambda}, \tilde{k}) = \frac{\exp(\tilde{k})}{\exp(\tilde{\lambda})} \left(\frac{y}{\exp(\tilde{\lambda})} \right)^{\exp(k)-1} e^{-(y/\exp(\lambda))^{\exp(k)}} \quad y > 0,$$

   where $-\infty < \tilde{\lambda}, \tilde{k} < \infty$. Write a function in `R`, `weib2_d0(pars, y, mult = 1)`, that evaluates the scaled log-likelihood of the model, i.e. $m \log f(\mathbf{y} \mid \tilde{\lambda}, \tilde{k})$, where `pars` is a 2-vector such that `pars[1]` $= \tilde{\lambda}$ and `pars[2]` $= \tilde{k}$, `y` is the data $\mathbf{y}$ and `mult` $= m$ is a multiplier as for `weib_d0()`.

2. Find the maximum likelihood estimates of $(\tilde{\lambda}, \tilde{k})$ using the Nelder-Mead method, starting at $(\tilde{\lambda}_0, \tilde{k}_0) = \big(\log(1.6), \log(0.6)\big)$.

3. Confirm that these are related to those of the original parameterisation through the logarithm.

Challenge I solution I

```
## $par
## [1]  0.6363058 -0.6206920
##
## $value
## [1] 54.95316
##
## $counts
## function gradient
##       41       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Challenge I solution II

```
## $par
## [1] 1.8900970 0.5374706
##
## $value
## [1] 54.95316
##
## $counts
## function gradient
##       43       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
## [1] 1.8900970 0.5374706
```

```
## [1] 1.8894878 0.5375723
```

Exam tips

1. Remember your differentiation rules. If you're asked to write a function to evaluate derivatives, don't lose marks because you've forgotten a differentiation rule.
2. Following on from above, 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.
3. 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.
4. Know your strengths. Section A of the exam may assess *any* material covered in MTH3045; however, in Section B, you can choose two out of three questions.

Global optimisation

- So far we have considered optimisation algorithms that usually home in on local minima, given starting points
- For multimodal surfaces, this can be undesirable
- Here we consider approaches to **global optimisation**, which are designed to find the *overall* minimum

Stochastic optimisation

- The optimisation algorithms that have been introduced so far have been deterministic
  - given a set of starting values, they will always return the same final value
- Stochastic optimisation algorithms go from one point to another probabilistically, and so will go through different sets of parameters
- We should expect them to converge to the same final value, though

Simulated annealing I

- Simulated annealing gains its name from the physical annealing process
  - a heat treatment that alters the physical and sometimes chemical properties of a material to increase its ductility and reduce its hardness, making it more workable
  - you can of course read more about it on Wikipedia[1]
- Consider a current parameter value $\boldsymbol{\theta}$ and some function we seek to minimise $f()$
- For example, $f()$ might be a negated log-likelihood
- The key to simulated annealing is that a *random* point is proposed, $\boldsymbol{\theta}^*$
  - $\boldsymbol{\theta}^*$ is drawn from some proposal density $q(\boldsymbol{\theta}^* \mid \boldsymbol{\theta})$, depends on the current value $\boldsymbol{\theta}$
- The proposal density $q()$ is chosen to be symmetric, but otherwise its choice is arbitrary

[1] https://en.wikipedia.org/wiki/Annealing_(materials_science)

Simulated annealing II

- The main aspect of simulated annealing is to work with the function

$$\pi_T(\boldsymbol{\theta}) = \exp\{-f(\boldsymbol{\theta})/T\}$$

  for some *temperature* T

- We note that as $T \searrow 0$, $\pi_T(\boldsymbol{\theta}) \to \exp\{-f(\boldsymbol{\theta})\}$
- Put algorithmically, simulated annealing works as follows

1. Propose $\boldsymbol{\theta}$ from $q(\boldsymbol{\theta}^* \mid \boldsymbol{\theta})$.
2. Generate $U \sim \text{Uniform}[0, 1]$.
3. Calculate

$$\begin{aligned}\alpha(\boldsymbol{\theta}^* \mid \boldsymbol{\theta}) &= \min\left\{\frac{\exp\left[-f(\boldsymbol{\theta}^*)/T\right]}{\exp\left[-f(\boldsymbol{\theta})/T\right]}, 1\right\} \\ &= \min\left(\exp\left\{-\left[f(\boldsymbol{\theta}^*) - f(\boldsymbol{\theta})\right]/T\right\}, 1\right).\end{aligned}$$

4. Accept $\boldsymbol{\theta}^*$ if $\alpha(\boldsymbol{\theta}^* \mid \boldsymbol{\theta}) > U$; otherwise keep $\boldsymbol{\theta}$.
5. Decrease T.

Simulated annealing III

- It is worth noting that steps 1 to 4 implement a special case of the Metropolis-Hastings algorithm for symmetric $q()$
- This algorithm is a heavily used in statistics, especially Bayesian statistics, to sample from posterior densities that have do not have or have unwieldy closed forms, typically as part of Markov chain Monte Carlo (MCMC) sampling.
- *Remark*: `R`'s default is to use a Gaussian distribution for $q()$ and the temperature at iteration i, T_i, is chosen according to $T_i = T_1 / \log\{t_{\max}\lfloor (i-1)/t_{\max} \rfloor + \exp(1)\}$ with $T_1 = t_{\max} = 10$ the default values

Example: Weibull maximum likelihood: Simulated annealing I

- Write a function to update the simulated annealing temperature according to R's rule and another function to generate Gaussian proposals with standard deviation 0.1
- Then use simulated annealing to repeat Example 5.5 with $N = 1000$ iterations and plot λ_i and k_i at each iteration using initial temperatures of $T_1 = 10$, 1 and 0.1
- The following function, `update_T()`, updates the temperature according to R's rule

```r
update_T <- function(i, t0 = 10, t1 = 10) {
  # Function to update simulated annealing temperature
  # i is an integer giving the current iteration
  # t0 is a scalar giving the initial temperature
  # t1 is a integer giving how many iterations of each temperature to use
  # returns a scalar
  t0 / log(((i - 1) %/% t1) * t1 + exp(1))
}
```

Example: Weibull maximum likelihood: Simulated annealing II

- Then the following function, `q_fn()`, generates Gaussian proposals with standard deviation 0.1

```
q_fn <- function(x) {
  # Function to generate Gaussian proposals with standard deviation 0.1
  # x is the Gaussian mean as either a scalar or vector
  # returns a scalar or vector, as x
  rnorm(length(x), x, .1)
}
```

- The following function can perform simulated annealing. You won't be asked to write such a function for MTH3045, but it's illuminating to see how such a function can be written.

```
sa <- function(p0, h, N, q, T1, ...) {
# Function to perform simulated annealing
# p0 p-vector of initial parameters
# h() function to be minimised
# N number of iterations
# q proposal function
# T1 initial temperature
# ... arguments to pass to h()
# returns p x N matrix of parameter estimates at each iteration
# commands suppressed
}
```

Example: Weibull maximum likelihood: Simulated annealing III

```r
sa <- function(p0, h, N, q, T1, ...) {
# comments suppressed
out <- matrix(0, N, length(p0)) # matrix to store estimates at each iteration
out[1, ] <- p0 # fill first row with initial parameter estimates
for (i in 2:N) { # N iterations
  T <- update_T(i, T1) # update temperature
  U <- runif(1) # generate U
  out[i, ] <- out[i - 1,] # carry over last parameter estimate, by default
  proposal <- q(out[i - 1,]) # generate proposal
  if (min(proposal) >= 0) { # ensure proposal valid
    h0 <- h(out[i - 1, ], ...) # evaluate h for current theta
    h1 <- h(proposal, ...) # evaluate h for proposed theta
    alpha <- min(exp(- (h1 - h0) / T), 1) # calculate M-H ratio
    if (alpha >= U) # accept if ratio sufficiently high
      out[i, ] <- proposal # swap last with proposal
  }
}
out # return all parameter estimates
}
```

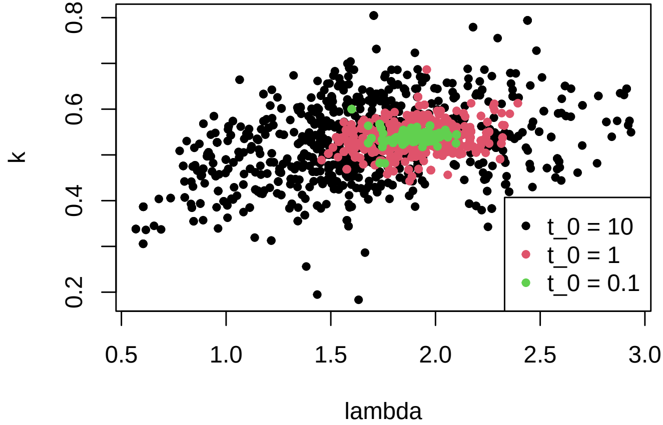
Example: Weibull maximum likelihood: Simulated annealing IV

- Then we'll specify out initial temperatures as `T_vals`, and loop over these with `sa()`...

```r
# values to use for initial temperature
T_vals <- c(10, 1, .1)
# loop over values, and plot
for (j in 1:length(T_vals)) {
  T1 <- T_vals[j]
  sa_result <- sa(c(1.6, .6), weib_d0, 1e3, q_fn, T1, y = y0, mult = -1)
  if (j == 1) {
    plot(sa_result, col = j, pch = 20, xlab = 'lambda', ylab = 'k')
  } else {
    points(sa_result, col = j, pch = 20)
  }
}
legend('bottomright', pch = 20, col = 1:length(T_vals),
       legend = paste("t_0 =", T_vals), bg = 'white')
```

Example: Weibull maximum likelihood: Simulated annealing V

- . . . and then plot the resulting parameter estimates for each temperature and each iteration



- We see that lower initial temperatures bring smaller clouds of parameter estimates.

Simulated annealing in `R` I

- Simulated annealing is built in to `R`'s `optim()` function and requires `method = 'SANN'`
- **Example**: Repeat Example 5.10 using `R`'s `optim()` function to perform simulated annealing
- Report the best value of the objective function every 100 iterations

Simulated annealing in R II

- We'll use the following call

```
optim(c(1.6, .6), weib_d0, y = y0, mult = -1, method = 'SANN',
      control = list(trace = 1, REPORT = 10, maxit = 1e3))
```

```
## sann objective function values
## initial       value 55.677933
## iter      100 value 54.956424
## iter      200 value 54.956424
## iter      300 value 54.956424
## iter      400 value 54.956424
## iter      500 value 54.956424
## iter      600 value 54.953563
## iter      700 value 54.953563
## iter      800 value 54.953563
## iter      900 value 54.953563
## iter      999 value 54.953563
## final         value 54.953563
## sann stopped after 999 iterations
```

```
## $par
## [1] 1.8979313 0.5396443
##
## $value
## [1] 54.95356
##
## $counts
```

Simulated annealing in R III

```
optim(c(1.6, .6), weib_d0, y = y0, mult = -1, method = 'SANN',
      control = list(maxit = 1e3))[1]
```

```
## $par
## [1] 1.8763494 0.5394845
```

- The parameter estimates are some way off those from earlier
- With more iterations, simulated annealing would gradually get closer to the true minimum
- Note that the `control$REPORT` argument specifies the frequency in terms of how often the temperature changes; so R reports the status of the optimiser each (`control$tmax * control$REPORT`)th iteration, noting that `control$tmax` defaults to 10.
- *Remark*: It's sometimes a good tactic to use simulated annealing to get close to the minimum, and then to employ one of the previously discussed deterministic optimisation methods to get a more accurate estimate
- This is especially useful if we're unsure whether we're starting off with sensible initial parameter estimates

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)

References

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

Nelder, J. A., and R. Mead. 1965. "A Simplex Method for Function Minimization." *The Computer Journal* 7 (4): 308–13. https://doi.org/10.1093/comjnl/7.4.308.

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.