Optimization

Kasper Rosenkrands

Fall 2019

Contents

1 Line search

A line serach algorithm chooses a direction p_k and searches along this direction from the current iterate x_k for a new iterate with a lower function value.

1.1 Search direction

The first step is to find a search direction, the most obvious one being $-\nabla f_k$. This direction have intuitive appeal as this is the direction that decreases f most rapidly. With a search direction in place we can now focus on the step.

1.2 Step size

For demonstration purposes, before looking more into intelligent ways of choosing step sizes, we will implement a simple algorithm which uses a constant step size.

1.2.1 Basic gradient descent algorithm

First we must import some data to work with. We will use the cars dataset from base R.

```
s <- cars$speed; d <- cars$dist
data_length <- length(s)
```

Then we define our objective function with we choose as the MSE. Furthermore we will provide the gradient.

```
f <- function(x) 1/data_length * sum(((x[1] + x[2] * s) - d)^2)
g <- function(x) {
  gradient.a <- 2/data_length * sum((x[1] + x[2] * s) - d)
  gradient.b <- 2/data_length * sum(s * ((x[1] + x[2] * s) - d))
  return(c(gradient.a, gradient.b))
}
```

Next we start to implement a basic gradient descent algorithm.

```
# Usign the norm as a criteria for convergence
norm2 <- function(x) norm(as.matrix(x), type = "2")

basic_gd <- function(output = FALSE, x_0 = c(-19,3.6), alpha = 1e-4, tolerance = 1e-4,
                     k_max = 100000) {
  x_k <- x_0
  f_iterates <- rep(0, k_max + 1)
  keep_going <- TRUE
  k <- 0
  f_iterates[1] <- f(x_k)
  while(keep_going) {
    k <- k + 1
    g_k <- g(x_k)
    p_k <- -g_k
    x_k <- x_k + alpha * p_k
    keep_going <- ((norm2(g_k) >= tolerance) & (k < k_max))
    f_iterates[k + 1] <- f(x_k)
  }
  if((norm2(g_k) <= tolerance)) {
    cat('Converged in', k, 'steps', '\n')
  } else {
    cat('Iteration limit reached', '\n')
  }
```

```
  cat('f(x_k) = ',f(x_k),'\t','x_k = ',x_k,'\n')
  if(output == TRUE) {
    return(f_iterates)
  }
}
basic_gd()
```

```
## Iteration limit reached
## f(x_k) =  227.0737    x_k =  -17.75634 3.94273
```

We did not converge in 100.000 steps, so we can assume that this algortihm is not very fast.

1.2.2 The wolfe conditions

the best choice of step length we can make is the global minimizer of the function

$$\phi(\alpha) = f(x_k + \alpha p_k), \quad \alpha > 0. \tag{1}$$

However it is not practical to calculate this every time as it would require to many evaluations of the objective function and/or gradient. We can however use some conditions to determine desireable step size at minimal cost.

Sufficient decrease: This condition ensure that the step insures a sufficient decrease in the objective function and can be formulated as

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k, \quad c_1 \in (0, 1). \tag{2}$$

In other words the condtion ensures that the reduction in the objective function is proportional to the step size and the directional derivative $\nabla f_k^T p_k$.

Curvature condition: This conditions ensures that when the objective function is decreasing at a rate beyond a certain point we will continue in that direction until gradient is not as steep anymore. The condition is formulated as follows

$$\nabla f(x_k + \alpha p_k)^T p_k \geq c_2 \nabla f_k^T p_k, \quad c_2 \in (c_1, 1). \tag{3}$$

These two conditions are known collectively as the *Wolfe conditions*, and if we furthermore pose a restriction on how positive the gradient can be we have what are called the *strong Wolfe conditions*

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k, \qquad c_1 \in (0, 1) \tag{4}$$

$$|\nabla f(x_k + \alpha p_k)^T p_k| \geq c_2 |\nabla f_k^T p_k|, \qquad c_2 \in (c_1, 1). \tag{5}$$

1.2.3 A strong Wolfe gradient descent algorithm

```
norm2 <- function(x) norm(as.matrix(x), type = "2")

# Implementation of Algorithm 3.5
# (Line Search Algorithm)
alpha <- function(a_0, x_k, c1, c2, r_k) {
  a_max <- 4*a_0
  f_k <- f(x_k)
  phi_k <- f_k
  a_1 <- a_0
  a0 <- 0
  a_k <- a_1
  a_k_old <- a0
```

```
  k <- 0
  k_max <- 10000
  done <- FALSE
  while(!done) {
    k <- k + 1
    f_k <- f(x_k)
    g_k <- g(x_k)
    p_k <- -g_k
    phi_k_old <- f(x_k + a_k_old * p_k)
      phi_k <- f(x_k + a_k * p_k)
      dphi_k_0 <- t(g(x_k)) %*% p_k
      l_k <- f_k + c1 * a_k * dphi_k_0
      if ((phi_k > l_k) || ((k > 1) && (phi_k >= phi_k_old))) {
        return(zoom(a_k_old, a_k, x_k, c1, c2))
      }
    dphi_k <- t(g(x_k + a_k * p_k)) %*% p_k
      if (abs(dphi_k) <= -c2*dphi_k_0) {
        return(a_k)
      }
    if (dphi_k >= 0) {
      return(zoom(a_k, a_k_old, x_k, c1, c2))
    }
    a_k_old <- a_k
    a_k <- r_k*a_k + (1 - r_k)*a_max
    done <- (k > k_max)
  }
  return(a_k)
}

# Implementation of Algorithm 3.6
# (Zoom Algorithm)
zoom <- function(a_lo, a_hi, x_k, c1, c2) {
  f_k <- f(x_k)
  g_k <- g(x_k)
  p_k <- -g_k
  k <- 0
  k_max <- 10000   # Maximum number of iterations.
  done <- FALSE
  while(!done) {
    k <- k + 1
    phi_lo <- f(x_k + a_lo * p_k)
      a_k <- 0.5*(a_lo + a_hi)
      phi_k <-  f(x_k + a_k * p_k)
      dphi_k_0 <- t(g(x_k)) %*% p_k
      l_k <-  f_k + c1 * a_k * dphi_k_0
      if ((phi_k > l_k) || (phi_k >= phi_lo)) {
        a_hi <- a_k
      } else {
        dphi_k <-  t(g(x_k + a_k * p_k)) %*% p_k
          if (abs(dphi_k) <= -c2*dphi_k_0) {
            return(a_k)
          }
        if (dphi_k*(a_hi - a_lo) >= 0) {
```

```
          a_hi <- a_lo
        }
        a_lo <- a_k
      }
    done <- (k > k_max)
  }
  return(a_k)
}
```

We can then create a function that uses the line search and zoom algortihm to actually minimize the objective function.

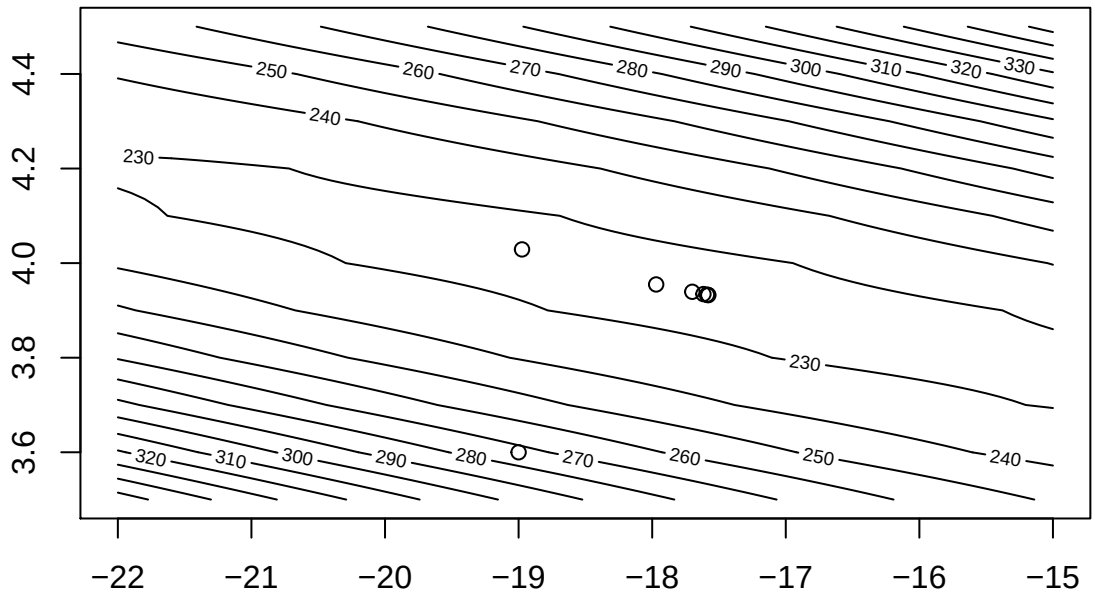
```
strong_wolfe_gd <-
  function(f, g, x_0, a_0 = 1, r_k = 0.5,
           c1 = 1e-4, c2 = 4e-1, tol_grad_f = 1e-4,
           k_max = 100000, output = FALSE, plot = FALSE) {
  f_iterates <- rep(0,k_max + 1)
  keep_going <- TRUE

  if (plot == TRUE) {
    agrid <- seq(-22,-15,.1)
    bgrid <- seq(3.5,4.5,.1)
    zvalues <- matrix(NA_real_,length(agrid),length(bgrid))
    for (i in 1:length(agrid)) {
      for (j in 1:length(bgrid)) {
        zvalues[i,j] <- f(c(agrid[i],bgrid[j]))
      }
    }
    my_levels <- seq(150, 400, 10)
    contour(agrid, bgrid, zvalues, levels = my_levels)
    points(x_0[1], x_0[2])
  }

  x_k <- x_0
  k <- 0
  f_iterates[1] <- f(x_k)
  while (keep_going) {
    k <- k + 1
    a_k <- alpha(a_0, x_k, c1, c2, r_k)
    g_k <- g(x_k)
    p_k <- -g_k
    x_k <- x_k + a_k*p_k
    keep_going <- (norm2(g_k) >= tol_grad_f) & (k < k_max)
    f_iterates[k + 1] <- f(x_k)

    #plot iterates
    if (plot == TRUE) {
      if (k == 1 || k %% 100 == 0) {
        points(x_k[1], x_k[2])
      }
    }
  }
  if (!output == TRUE || !plot == TRUE) {
    if((norm2(g_k) <= tol_grad_f)) {
```

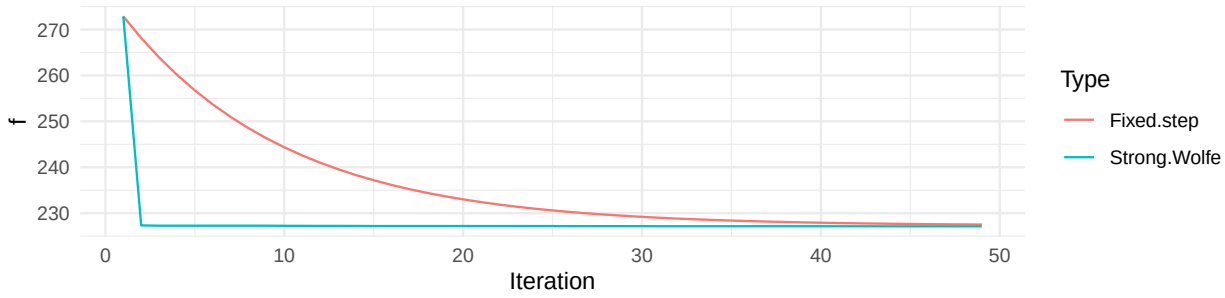
```
      cat('Converged in', k, 'steps', '\n')
    } else {
      cat('Iteration limit reached', '\n')
    }
    cat('f(x_k) = ',f(x_k),'\t', 'x_k = ', x_k, '\n')
  }
  if (output == TRUE) {
    return(f_iterates)
  }
}
strong_wolfe_gd(f,g, c(-19, 3.6), output = FALSE, plot = TRUE)
```




```
## Converged in 662 steps
## f(x_k) =  227.0704    x_k =  -17.57954 3.932433
```

We can conclude that his algorithm converges must faster for the given starting value. This algortihm converged in 662 steps whereas the simpler algorithm did not converge in 100.000 iterations.

To see the difference in the two algorithms for just the first 50 iterations the following plot can be considered. Notice how much difference progress there is the two algorithms in just the first step.



I suspect that the reason for the little reduction in the objective function after the first step is due to poor scaling.

2 Calculating derivatives

Many numerical optimization algorithms use the gradient to minimize the respective objective function. However sometimes it can be time consuming if the objective function is complicated. Therefore it would beneficial to have the algortihm calculate the derivative automatically. There are several different approaches that could be taken. The first approach we will consider is called finite differencing.

2.1 Finite differencing

The idea is to estimate the derivatives by observing the change in function values in response to small perturbations of the unknowns near a given point.

2.1.1 Foward difference

In the case of forward-difference our estimate of the gradient for a function is given by

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + \varepsilon e_i)}{\varepsilon}. \tag{6}$$

In the case of central-difference our estimate is given as

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + \varepsilon e_i) - f(x - \varepsilon e_i)}{2\varepsilon}. \tag{7}$$

These estimate arises from Taylor's formula that states

$$f(x + h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) - O(h^4), \tag{8}$$

rearranging gives that

$$f'(x) = \frac{f(x + h) - f(x)}{h} + \frac{1}{2}hf''(x) + \frac{1}{6}h^2 f'''(x) - O(h^3). \tag{9}$$

By considering only the first term after the equality we give rise to a truncation error. We are only interested in $0 < h < 1$, such that the truncation error becomes $O(h)$. Furthermore the truncation error decreases as h decreases. The could lead us to believe that we can then choose a very small h a go on our merry way. However this is not the case due to floating point arithmetic producing a round-off error. It can be shown that the round-off error bound is $O(\varepsilon_M/h)$, i.e. the round-off error increases as h decreases. Thus we have established that there is a trade-off between truncation and round-off error.

2.1.2 Central difference

Consider now both forward and backward difference, giving by

$$f(x + h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + O(h^3) \tag{10}$$

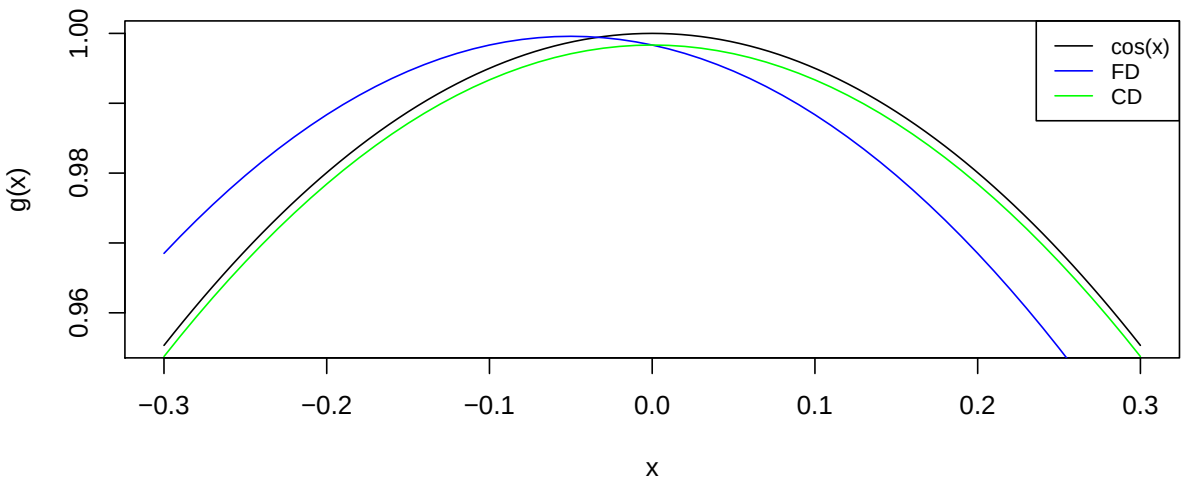
$$f(x - h) = f(x) - hf'(x) + \frac{1}{2}h^2 f''(x) + O(h^3), \tag{11}$$

subtracting these two equations leads us to the following expression

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h} - O(h^2). \tag{12}$$

In other words the truncation error is now smaller, however we need to evaluate the objective function twice. This will have an impact on the performance of the algorithm, that depends on how expensive the objective function is to evaluate.

The difference between the estimates from the two methods can be seen in the following plot.



It is clear to see that the CD is closer to the actual gradient than the FD is.

2.2 Five-Point stencil

Going further than the central difference formula, one could also include the terms $f(x+2h)$ and $f(x-2h)$ to cancel out even more terms. Recalliung the taylor expansion of f gives

$$f(x \pm h) = f(x) \pm hf'(x) + \frac{h^2}{2!}f''(x) \pm \frac{h^3}{3!}f'''(x) + O(h^4)$$

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{3}f'''(x) + O(h^4)$$

$$f(x+2h) - f(x-2h) = 4hf'(x) + \frac{8h^3}{3}f'''(x) + O(h^4)$$

To get rid of the third term we write

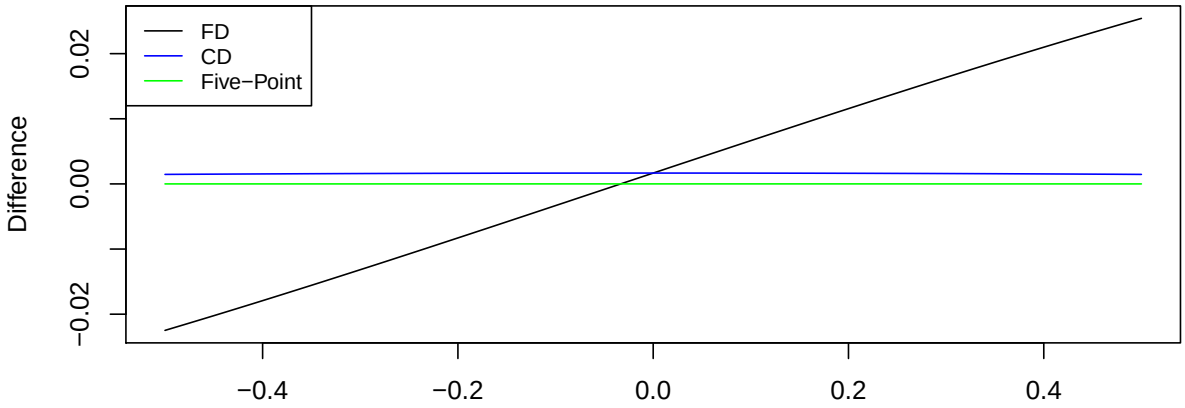
$$8f(x+h) - 8f(x-h) - f(x+2h) + f(x-2h) = 12hf'(x) + O(h^4)$$

Thus we have

$$f'(x) = \frac{8f(x+h) - 8f(x-h) - f(x+2h) + f(x-2h)}{12h}$$

where our truncation error is $O(h^4)$, which all else equal is better than for both forward and central differencing. These are $O(h)$ and $O(h^2)$ respectively.

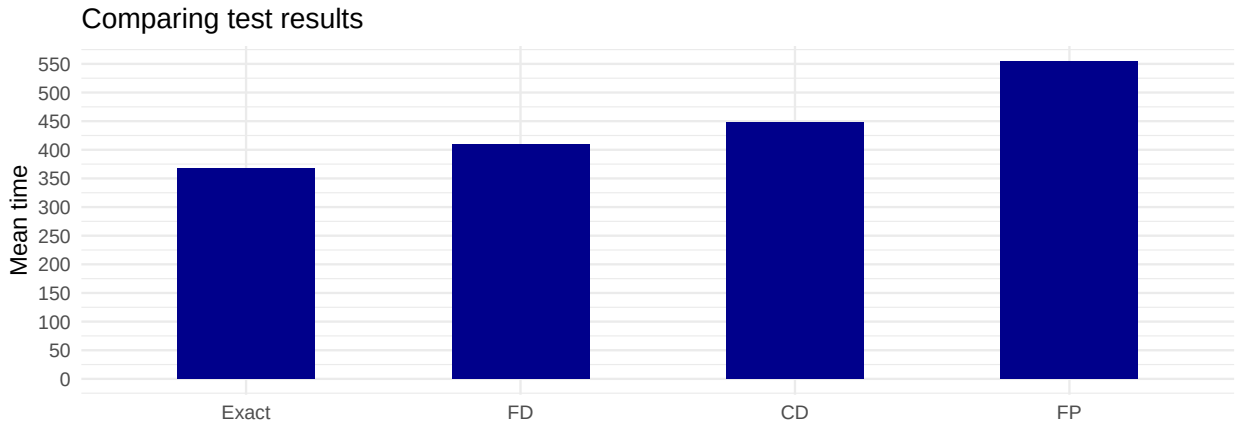
The performance for the three different methods can be seen in the plot below.



2.3 Finite difference gradient descent algortihm

We have implemented four different algortihms that utilizes the exact gradient, forward differencing, central difference and the five point stencil respectively.

The following graph show how the different algorithms performed, on average, across 10 runs.



We can conclude that in this particular instance, we have the exact gradient giving the best performance. From there, the more complicated we make the finite differencing, the more performance we lose. However this is only in this particular instance. Suppose we have a very complicated objective function, in that case we could imagine that if we can make our gradient estimate more precise it would be worth the added computation invloved.

2.4 Algorithmic differentiation (AD)

2.4.1 Forward mode

Algorithmic differentiation builds on the principle that, no matter how complicated a function might be, it builds on a sequence of simple elementary operations. To illustrate we will consider an example. Take the

function

$$f(x) = \cos(x_1 + x_2) \cdot x_3. \tag{13}$$

We can express the evaluation of f as

$$x_4 = x_1 + x_2 \tag{14}$$
$$x_5 = \cos(x_4) \tag{15}$$
$$x_6 = x_5 \cdot x_3, \tag{16}$$

with the final node x_6 being the function value $f(x)$.

Suppose we now want to find the derivative of this function w.r.t. x_1. If we denote the intermidiate variables by v, and to each variable associate a new variable, $\dot{v}_i = \partial v_i / \partial x_1$. We can then apply the chain rule mechanically to each line in the evaluation trace, this will then give the numerical value to each $\dot{v}_i$. First we will go thorugh the first iterations as an example, and then present the full evaluation trace

$$\begin{aligned}
v_{-2} &= x_1 && = 1 \\
v_{-1} &= x_2 && = 2 \\
v_0 &= x_3 && = 3 \\
v_1 &= v_{-2} + v_{-1} && = 3 \\
v_2 &= \cos(v_1) && = -0.99 \\
v_3 &= v_2 \cdot v_0 && = -2.97 \\
y &= v_3 && = -2.97
\end{aligned}$$

Using the chain rule, given by

$$\frac{\partial y}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial y}{\partial u_j} \frac{\partial u_j}{\partial x_i} \tag{17}$$

we find that

$$\dot{v}_{-2} = \frac{\partial v_{-2}}{\partial x_1} = \frac{\partial}{\partial x_1} x_1 = 1, \quad \dot{v}_{-1} = \frac{\partial v_{-1}}{\partial x_1} = \frac{\partial}{\partial x_1} x_2 = 0 \quad \text{and} \quad \dot{v}_0 = \frac{\partial v_0}{\partial x_1} = \frac{\partial}{\partial x_1} x_3 = 0$$
$$\dot{v}_1 = \frac{\partial v_1}{\partial x_1} = \frac{\partial v_1}{\partial v_{-2}} \frac{\partial v_{-2}}{\partial x_1} + \frac{\partial v_1}{\partial v_{-1}} \frac{\partial v_{-1}}{\partial x_1} + \frac{\partial v_1}{\partial v_0} \frac{\partial v_0}{\partial x_1} = \frac{\partial v_1}{\partial v_{-2}} 1 + \frac{\partial v_1}{\partial v_{-1}} 0 + \frac{\partial v_1}{\partial v_0} 0 = 1.$$

The rest is calculated in the same manner, yielding the full evaluation trace as

$$\begin{aligned}
v_{-2} &= x_1 && = 1 \\
\dot{v}_{-2} &= \dot{x}_1 && = 1 \\
v_{-1} &= x_2 && = 2 \\
\dot{v}_{-1} &= \dot{x}_2 && = 0 \\
v_0 &= x_3 && = 3 \\
\dot{v}_0 &= \dot{x}_3 && = 0 \\
v_1 &= v_{-2} + v_{-1} && = 3 \\
\dot{v}_1 &= 1 && = 1 \\
v_2 &= \cos(v_1) && = -0.99 \\
\dot{v}_2 &= -\sin(v_1) && = -0.83 \\
v_3 &= v_2 \cdot v_0 && = -2.97 \\
\dot{v}_3 &= -v_0 \cdot \sin(v_1) && = -2.51 \\
y &= \cos(x_1 + x_2) x_3 && = -2.97 \\
\dot{y} &= -\sin(x_1 + x_2) x_3 && = -0.41.
\end{aligned}$$

2.4.1.1 Implementation

Below is an implementation of algorithmic differentiation (forward mode.

```
# Make an algorithmic differentiation number
create_ADnum <- function(val, deriv = 1) {
  # a list that includes the number and its derivative
  x <- list(val = val, deriv = deriv)
  # make a class in R called 'ADnum'
  class(x) <- 'ADnum'
  return(x)
}

# Make the number 4 in a smart way
x <- create_ADnum(4)
x
```

```
## $val
## [1] 4
##
## $deriv
## [1] 1
##
## attr(,"class")
## [1] "ADnum"
```

```
# construct a custom print function for the 'ADnum' class
print.ADnum <- function(x, ...) {
  cat('value = ', x$val,
      ' and deriv = ', x$deriv, '\n', sep = '')
  return(invisible(x))
}
x
```

```
## value = 4 and deriv = 1
```

R knows how to calculate with regular numbers, but cannot yet calculate using number from the "ADnum" class. For R to be able to do that we need to do what is known as operator overloading.

```
Ops.ADnum <- function(e1, e2) {
  # Convert the first number to ADnum
  if (.Method[1] == '') {
    e1 <- create_ADnum(e1, 0)
  }
  # Convert the second number to ADnum
  if (.Method[2] == '') {
    e2 <- create_ADnum(e2, 0)
  }

  if (.Generic == '*') {
    return(create_ADnum(e1$val * e2$val, e1$deriv*e2$val + e2$deriv*e1$val))
  }

  if (.Generic == '+') {
    return(create_ADnum(e1$val + e2$val, e1$deriv + e2$deriv))
  }
```

```
  if (.Generic == '-') {
    return(create_ADnum(e1$val - e2$val, e1$deriv - e2$deriv))
  }

  if (.Generic == '/') {
    return(create_ADnum(e1$val / e2$val, (e1$deriv*e2$val - e1$val*e2$deriv)/e2$val^2))
  }

  stop("Function '", Generic, "' not yet implemented for ADnum")
}

Math.ADnum <- function(x, ...) {
  if (.Generic == "cos") {
    return(create_ADnum(cos(x$val), x$deriv * -sin(x$val)))
  } else if (.Generic == "sin") {
    return(create_ADnum(sin(x$val), x$deriv * cos(x$val)))
  } else if (.Generic == "exp") {
    return(create_ADnum(exp(x$val), x$deriv * exp(x$val)))
  }
  stop("Function '", .Generic, "' not yet implemented for Ad num")
}
```

Using the example from earlier we will now check that using our implementation we will in fact gain the same result.

```
x <- create_ADnum(1); cos(x + 2)*3
```

```
## value = -2.969977 and deriv = -0.42336
```

We do in fact get the same result using our implementation of algorithmic differentiation.

3 Quasi Newton

Recall that the newton method uses the hessian to minimize the objective function. On the contrary quasi-Newton methods uses an approximation of the true Hessian. Therefore these method are a good alternative when the Hessian is unavailable or just to expensive to compute. We could state it more formally in the following way; line search iterations are given by $x_{k+1} = x_k + \alpha_k p_k$ where the search direction is given by $p_k = -B_k^{-1}\nabla f_k$. In the case of steepest descent $B_k = I$, for Newton's method $B_k = \nabla^2 f_k$ and for quasi-Newton methods B_k is an approximation to the Heassian that is updated at every iteration by means of a low-rank formula.

3.1 BFGS

The BFGS algorithm is the most popular of the quasi-Newton methods. First we will state the quadratic model of the objective function we will use at any given iteration

$$m_k(p) = f_k + \nabla f_k^\top p + \frac{1}{2} p^\top B_k p, \tag{18}$$

here B_k is a symmetric and positive definite matrix that will be updated at every iteration. The minimizer p_k of this convex quadratic model, which we can write explicitly as

$$p_k = -B_k^{-1}\nabla f_k \tag{19}$$

is used as the search direction and the new iterate is

$$x_{k+1} = x_k + \alpha_k p_k, \tag{20}$$

where α_k is the step length.

Computing B_k at each iteration is expensive, therefore we update it iteratively using the curvature from the most recent step. At a new iterate x_{k+1} construct a new quadratic model of the form

$$m_{k+1}(p) = f_{k+1} + \nabla f_{k+1}^\top + \frac{1}{2} p^\top B_k p. \tag{21}$$

A reasonable requirement to B_{k+1}, based on the knowledge gained during the latest step, is that $\nabla m_{k+1}(p) = \nabla f_{k+1} + B_{k+1}p$ at both x_k and x_{k+1}. As $\nabla m_{k+1}(0) = \nabla f_{k+1}$, the second condition is always satisfied. The first condition can be written as

$$\nabla m_{k+1}(-\alpha_k p_k) = \nabla f_{k+1} - \alpha_k B_{k+1} p_k = \nabla f_k. \tag{22}$$

By rearranging, we obtain

$$B_{k+1}\alpha_k p_k = \nabla f_{k+1} - \nabla f_k, \tag{23}$$

to simplify the expression we introduce the notation

$$s_k = x_{k+1} - x_k = \alpha p_k, \quad y_k = \nabla f_{k+1} - \nabla f_k. \tag{24}$$

This lets us rewrite (23) as

$$B_{k+1} s_k = y_k. \tag{25}$$

This formula is also known as the secant equation. In other words we can say that the symmetric positive definite matrix B_{k+1} must map s_k into y_k. This is only possible if s_k and y_k satisfy the curvature condtion $s_k^\top y_k > 0$ as we have $s_k^\top B_{k+1} s_k = s_k^\top y_k$ by multiplying (25) with $s_k^\top$. The equation is satisfied when f is strongly convex, for any two x_k and x_{k+1}. This will not always hold for nonconvex functions but can be enforced explicitly by choosing a step length α that satisfy the Wolfe or Strong Wolfe conditions. When the curvature condition is met, the secant equation will have an infinite number of solution. To choose

B_{k+1} uniquely we add the additional constraint: Among all of the symmetric matrices satisfying the secant equation, choose the B_{k+1} closest to the current matrix B_k.

The additional constraint corresponds to solving the problem

$$\min_B \|B - B_k\|, \tag{26}$$

subject to

$$B = B^\top, \quad B_{s_k} = y_k, \tag{27}$$

where s_k satisfies the secant equation and B_k is symmetric and positive definite.

The BFGS updating formula can be derived with just a slight modification in the above argumentation, instead of imposing conditions on the Hessian approximations B_k, we impose similar conditions on their inverses giving by H_k. The updated approximation must be symmetric and positive definite, and must satisfy the secant equation, now written as

$$H_{k+1} y_k = s_k. \tag{28}$$

The condition of closeness to H_k is now specified by the follwoing analogue of (26)

$$\min_H \|H - H_k\|, \tag{29}$$

$$\text{subject to} \quad H = H^\top, \quad H_{s_k} = y_k. \tag{30}$$

A norm that allows easy solution of the minimization problem is the weighted Frobenius norm

$$\|A\|_W = \|W^{1/2} A W^{1/2}\|_F \quad \text{with the defintion of } \|\cdot\|_F \text{ as } \|C\|_F^2 = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}^2. \tag{31}$$

Note that W can be chosen as any matrix satisfying the equation $W s_k = y_k$.

The unique solution H_{k+1} to the minimization problem is given by

$$\text{(BFGS)} \qquad H_{k+1} = (I - \rho_k s_k y_k^\top) H_k (I - \rho_k y_k s_k^\top) + \rho_k s_k s_k^\top, \tag{32}$$

with ρ_k defined as

$$\rho_k = \frac{1}{y_k^\top s_k}. \tag{33}$$

3.1.1 Choosing the initial approximation

There is no magic formula to choose the initial approximation H_0. One could choose to make a finite difference approximation of the Hessian and then invert this approximation. Another approach is just to set it equal to the identity matrix, or a multiple of the identity that reflects the scaling of the problem.

3.2 Implementation of BFGS

Below is an implementation of the BFGS method, the initial approximation H_0 is set to the identity matrix and the step length is set to be 1. Although it could be beneficial to choose a step length that satisfy the Wolfe or Strong Wolfe conditions as mentioned above, we will see that BFGS will work even if the conditions are not satisfied. Firstly we will define an objective function to test the BFGS on, we will use a convex elliptical objective function.

```r
# to determine convergence
norm2 <- function(x) norm(as.matrix(x), type = "2")

# objective function
f <- function(x) 0.5*(100*x[1]^2 + x[2]^2)
f_xy <- function(x,y) 0.5*(100*x^2 + y^2)
x_min_true <- c(0, 0)

# analytic derivatives
g <- function(x) c(0.5*200*x[1], 0.5*2*x[2])
H <- function(x) 0.5*rbind(c(200, 0), c(0, 2))
```

Now we will implement the BFGS method

```r
BFGS <- function(x_0) {
  x_k <- x_0
  tolerance <- 1e-10
  I <- rbind(c(1,0),c(0,1))
  h_k <- I
  k <- 0
  k_max <- 10000
  while (norm2(g(x_k)) >= tolerance & k < k_max) {
    x_k_old <- x_k
    g_k <- g(x_k)
    g_k_old <- g(x_k_old)
    p_k <- -h_k %*% g_k
    a_k <- 1
    x_k <- x_k + a_k * p_k
    g_k <- g(x_k)
    s_k <- x_k - x_k_old
    y_k <- g_k - g_k_old
    rho_k <- as.numeric(1/(t(y_k)%*%s_k))
    h_k <- {(I - rho_k * s_k %*% t(y_k)) %*% h_k %*%
      (I - rho_k * y_k %*% t(s_k)) + rho_k * s_k %*% t(s_k)}
    k <- k + 1
  }
  cat('number of iterations:', '\t',  k, '\n')
  return(x_k)
}
```

We can now give an initial value and try the BFGS function

```r
x_0 <- c(1,1.5)
BFGS(x_0)
```

```
## number of iterations:     4
```

```
##                [,1]
## [1,] -5.539186e-21
## [2,] -7.940934e-23
```

The BFGS method reaches convergence after 4 iterations. Comparing this to the earlier implementation of gradient descent gives

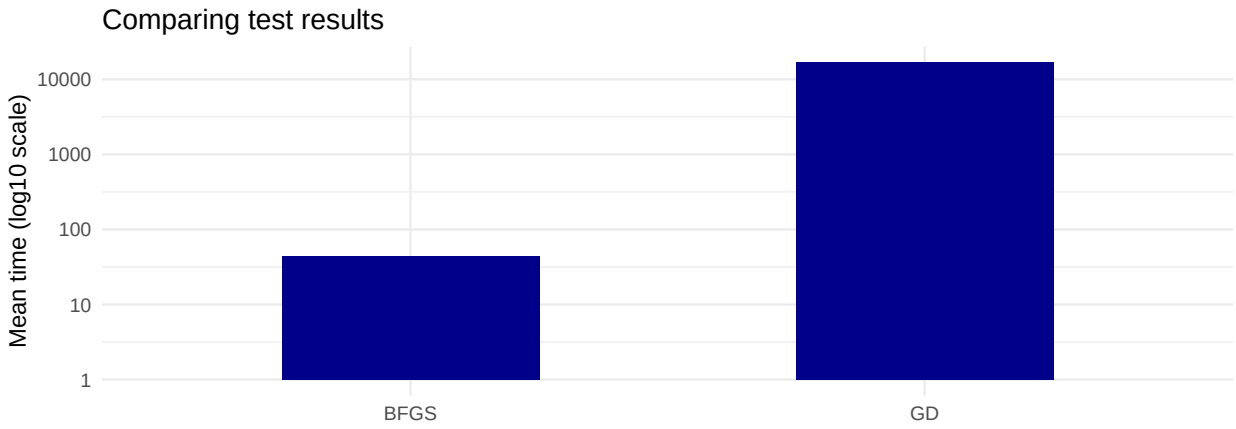
```r
strong_wolfe_gd(f,g,x_0,tol_grad_f = 1e-10)
```

```
## Converged in 1043 steps
```

```
## f(x_k) =  3.532431e-21    x_k =  1.004416e-12 8.345045e-11
```

Thus we can conclude that on this problem the BFGS uses considerably less iterations, even with a step length not satisfiying the the Wolfe or strong Wolfe conditions.

Using `microbenchmark` we can see how much time it took to perform the iterations. As every iteration in the BFGS algorithm is more expensive than in the gradient descent, the latter might still be faster. However as we can see in the following plot, this is not the case.



3.2.1 Inverse Hessian approximations

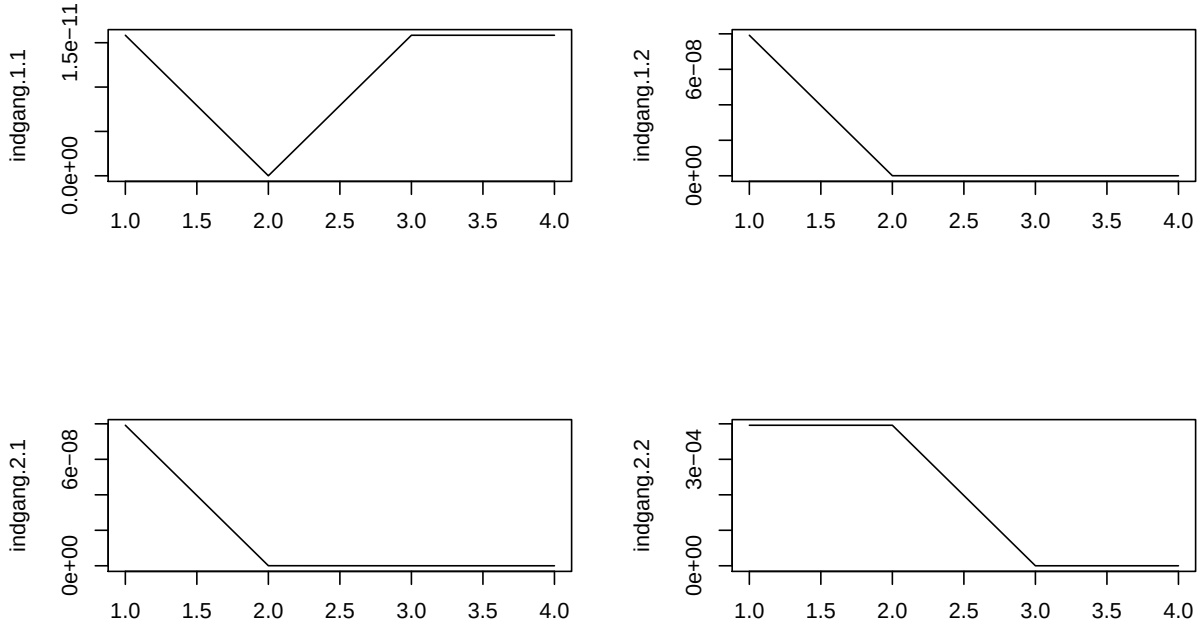
In this section we will investigate how well the Hessian is approximated in the BFGS algorithm. The following `chunk` is a modified version of the BFGS implementation from earlier. It has the added functionality that it saves the analytical Hessian and the approximated Hessian for each iteration.

```
BFGS2 <- function(x_0, print = TRUE) {
  x_k <- x_0
  tolerance <- 1e-10
  I <- rbind(c(1,0),c(0,1))
  h_k <- I
  k <- 0
  k_max <- 10000
  #Create empty lists for matricies
  Hessian <- c()
  Truehessian <- c()
  while (norm2(g(x_k)) >= tolerance & k < k_max) {
    x_k_old <- x_k
    g_k <- g(x_k)
    g_k_old <- g(x_k_old)
    p_k <- -h_k %*% g_k
    a_k <- 1
    x_k <- x_k + a_k * p_k
    g_k <- g(x_k)
    s_k <- x_k - x_k_old
    y_k <- g_k - g_k_old
    rho_k <- as.numeric(1/(t(y_k)%*%s_k))
    h_k <- {(I - rho_k * s_k %*% t(y_k)) %*% h_k %*%
        (I - rho_k * y_k %*% t(s_k)) + rho_k * s_k %*% t(s_k)}
    k <- k + 1
    Truehessian[[k]] <- solve(H(x_k))
```

```
    Hessian[[k]] <- h_k
    if (print == TRUE) {
      cat('Approximated inverse hessian:' , '\t' , h_k , '\n')
      cat('Analytical inverse hessian:' , '\t' , solve(H(x_k)) , '\n')
    }
  }
  Hes_list <- list(Hessian, Truehessian)
  if (print == TRUE) {
    cat('number of iterations:', '\t',  k, '\n')
  }
  return(Hes_list)
}
Hes_list2 <- BFGS2(c(1,2))
```

```
## Approximated inverse hessian:     0.01 -7.919937e-08 -7.919937e-08 1.000396
## Analytical inverse hessian:   0.01 0 0 1
## Approximated inverse hessian:     0.01 3.13627e-11 3.13627e-11 1.000396
## Analytical inverse hessian:   0.01 0 0 1
## Approximated inverse hessian:     0.01 3.166695e-13 3.166695e-13 1
## Analytical inverse hessian:   0.01 0 0 1
## Approximated inverse hessian:     0.01 -1.254503e-16 -1.254503e-16 1
## Analytical inverse hessian:   0.01 0 0 1
## number of iterations:     4
```

The following plot shows the difference for each entry between the approximated and analytical Hessian.

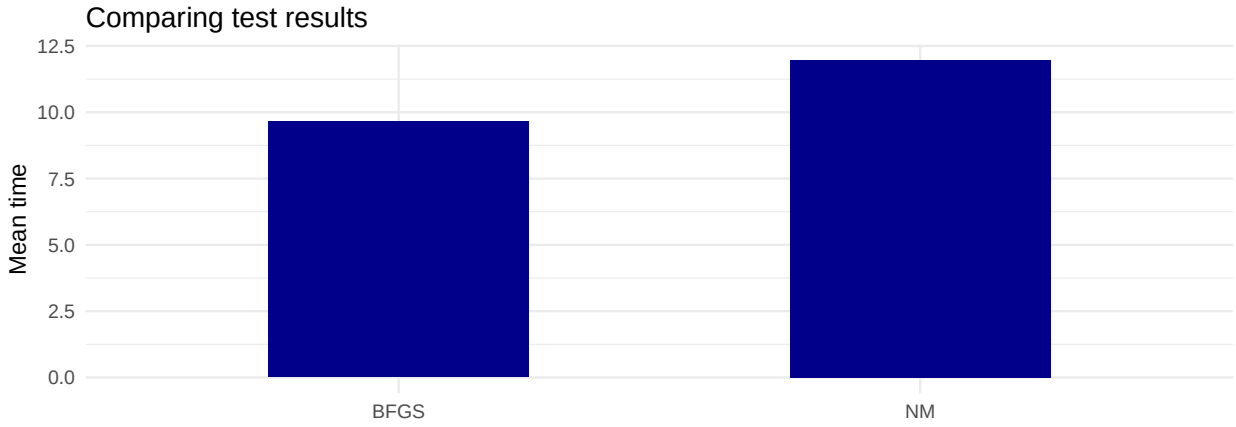


They are very similar, even from the first iteration. The approximation is therefore practically identical to the analytical Hessian. For 3 of the entries the difference even shrinks after the 4 iterations.

3.3 Nelder-Mead

A popular derivative free optimization method is the Nelder-Mead method. At any given point it keeps track of $n+1$ points of interest in $\mathbb{R}^n$. In a single iteration of the Nelder-Mead algorithm we seek to remove the matrix with the worst function value and replace it with another point with a better value. The new point is found be either reflecting, expanding or contracting the simplex along the line joining the worst vertex with the centroid of the remaining vertices. If we cannot find a better point in this manner, we retain only the vertex with the best function value, and we shrink the simplex by moving all other vertices toward this value.

We can now compare that to the `optim` functions implementation of 'Nelder-Mead', which is a derivative free optimisation algorithm with their implementation of BFGS.



We conclude that the BFGS algorithm is faster than Nelder-Mead, however Nelder-Mead has the advantage that we don't have to provide the gradient.

4 Least Squares

In least-squares problems, the objective have the following special form:

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x). \tag{34}$$

We call r_j a residual. Furthermore we now that it is a smooth function from $\mathbb{R}^n$ to $\mathbb{R}$. The special form of f makes this particular type of problem easier to solve than general unconstrained minimization problems.

4.1 Linear least-squares problems

If the model $\phi(x; t_j)$ is a linear function of x what we have is a linear least-squares problem. The residual is given by $r(x) = Jx - y$, where we call J the design matrix. The objective function becomes

$$f(x) = \frac{1}{2}\|Jx - y\|^2. \tag{35}$$

The derivative and second derivative is given by

$$\nabla f(x) = J^\top (Jx - y) \tag{36}$$

$$\nabla^2 f(x) = J^\top J. \tag{37}$$

The objective function (35) is convex and therefore theorem 2.5 states that: any $x^* \ : \ \nabla f(x^*) = 0$ also the global minimizer of f.

Knowing this we can deduce that

$$\nabla f(x) = J^\top (Jx - y) \text{ and } \nabla f(x^*) = 0 \tag{38}$$

$$0 = f(x^*) = J^\top (Jx^* - y) = J^\top Jx^* - J^\top y. \tag{39}$$

This means that x^* satisfies the linear system of equations

$$J^\top Jx^* - J^\top y, \tag{40}$$

which are also known as the normal equations. So to solve this unconstrained linear least squares problem, is equivalent to solving the set of normal equations given by (40).

4.2 Algorithms for solving

There are three major algortihms for solving linear least squares problem, which we will present in this chapter. Furthermore for every method we will present an implementation in `R` and compare the performance of the different methods.

For testing purposes we will use the `cars` dataset which we will now import.

```
spd <- cars$speed
dst <- cars$dist
```

We will then define the design matrix using the `cars` data.

```
j.1 <- rep(1, length(spd))
j.2 <- spd
J <- cbind(j.1, j.2)
y <- dst
```

4.2.1 Cholesky / Normal equations

We start out by noting that

$$z^\top J^\top J z = (Jz)^\top Jz = \|Jz\|^2 > 0 \tag{41}$$

as $Jz \neq 0$ when we assume that J has full column rank. This implies that $J^\top J$ is positive definite. Therefore we can use Cholesky to factorise a matrix A to lower triangular L and L's conjugate transpose, $\overline{L}^\top$. Then solving the equations with a triangular would be easy by back substitution.

We can rephrase the problem by replacing

$$J^\top J x^* - J^\top y \text{ with } Ax = b \tag{42}$$

$$\text{where } A = J^\top J \text{ and } b = J^\top y. \tag{43}$$

We can then use the following three step procedure to solve the system

- compute the coefficient matrix $A = J^\top J$;
- compute the cholesky factorization of the symmetric matrix $J^\top J$;
- perform two triangular substitutions with the Cholesky factors to recover the solution x^*.

The two triangular substitutions are as follows

$$\text{solve for } z\text{: } Lz = b \tag{44}$$

$$\text{solve for } x\text{: } L^\top x = z. \tag{45}$$

Below is a implementation of the procedure.

```
# Compute A matrix by multiplying the transpose of J with J
A <- t(J)%*%J
# Using the chol function to find the upper triangular cholesky factorization
U <- chol(A)
L <- t(chol(A))
# Computing the left side of the equation, by multiplying x transpose with y
b <- t(J)%*%y
# Using first the forwardsolve to find z
z <- forwardsolve(L, b)
# Hereafter using backsolve to find the coefficients
chol_coef <- backsolve(U, z)
# We can make a chol function for testing purposes
chol.est <- function(J) {
  A <- t(J)%*%J
  U <- chol(A)
  L <- t(U)
  b <- t(J)%*%y
  z <- forwardsolve(L, b)
  chol_coef <- backsolve(U, z)
  return(chol_coef)
}
chol.est(J)
```

```
##            [,1]
## [1,] -17.579095
## [2,]   3.932409
```

4.2.2 QR

Another method is using QR factorization. We can decompose $J = QR$ for orthogonal Q and upper triangular R. Assuming J is square we have

$$\begin{aligned} x^* &= \left(J^\top J\right)^{-1} J^\top y = \left((QR)^\top (QR)\right)^{-1} (QR)^\top y \\ &= \left(R^\top Q^\top QR\right)^{-1} R^\top Q^\top y \\ &= \left(R^\top R\right)^{-1} R^\top Q^\top y \\ &= R^{-1}(R^\top)^{-1} R^\top Q^\top y \\ &= R^{-1} Q^\top y. \end{aligned}$$

Therefore we can solve least squares by solving

$$Rx^* = Q^\top y \tag{46}$$

using back substitution as R is triangular.

Below is an implementation of the procedure.

```
# First we make the qr decomposition
QR <- qr(J)
# Then we utilize functions qr.Q and qr.R to extract the corresponding matrices
Q <- qr.Q(QR)
R <- qr.R(QR)
# As the equation we want to solve is Rx*=Q^T*y and R
# is upper triangular we can use the backsolve function
QR_coef <- backsolve(R, t(Q)%*%y)
# We can now make a QR function for testing purposes
qr.est <- function (J) {
  QR <- qr(J)
  Q <- qr.Q(QR)
  R <- qr.R(QR)
  QR_coef <- backsolve(R, t(Q)%*%y)
  return(QR_coef)
}
qr.est(J)
```

```
##             [,1]
## [1,] -17.579095
## [2,]   3.932409
```

4.2.3 SVD

This third approach is based on the singular-value decomposition of J, it argues that $J = USV^\top$ for orthogonal U, V and diagonal S. It describes the effect of a matrix on a vector $Jx = USV^\top x$ as

- a rotation/reflection in the input space ($V^\top$)
- a scaling that takes a vector in the input space to the output space S
- another rotation/reflection in the output space (U)

Assuming J is square we have

$$\begin{aligned} x^* &= \left(J^\top J\right)^{-1} J^\top y \\ &= \left((USV^\top)^\top (USV^\top)\right)^{-1} (USV^\top)^\top y \\ &= \left((VSU^\top)(USV^\top)\right)^{-1} (VSU^\top) y \\ &= \left(VS^2V^\top\right)^{-1} (VSU^\top) y \\ &= (V^\top)^{-1} S^{-2} V^{-1} (VSU^\top) y = (V^\top)^{-1} S^{-1} U^\top y \\ &= VS^{-1}U^\top y. \end{aligned}$$

When $m > n$, similar to QR, then

$$x^* = VS^{-1}U^\top y \tag{47}$$

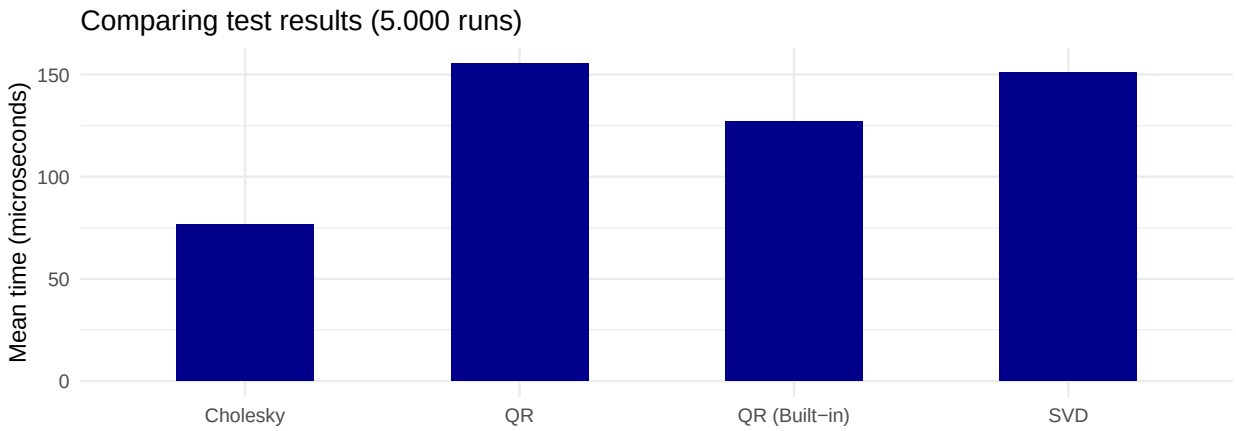
Below is an implementation of the procedure.

```
# Perform SVD decomposition
svd. <- svd(J)
# Extract the three matrices from the decomposition
S <- diag(svd.$d)
U <- svd.$u
V <- svd.$v
# Solve the equation
svd_coef <- V%*%solve(S)%*%t(U)%*%y
# We can now make an SVD function for testing purposes
svd.est <- function (J) {
  svd. <- svd(J)
  S <- diag(svd.$d)
  U <- svd.$u
  V <- svd.$v
  svd_coef <- V%*%solve(S)%*%t(U)%*%y
  return(svd_coef)
}
```

4.3 Comparison of the algorithms

The Cholesky based algorithm is particularly useful when $n < m$ and it is practical to store $J^\top J$ but not J itself. However the method can degrade if J is ill-conditioned. The QR based algortihm avoids squaring of the condition number and hence may be more robust that the Cholesky based method. The SVD approach is the most robust of the three and even reveals information about the sesitivity of the solution to pertubation in the data.

We can now compare the performance of the three implementations, note that in addition to using the `qr.est` function we created above we also compare with the built-in function `qr.solve` as our function has added complexity for illustrative purposes.



We can conclude that in our case there is a positive relationship between robustness and performance. The Cholesky approach which is theoretically the least stable have the best performance, while the SVD approach which is theoretically the most stable have the worst performance if we disregard our implementaion of the QR method. comparing the two QR implementations the built-in have an advantage when compared to our implementation. We suspect that the discrepancy can be explained by our implementation doing some work twice.

4.4 Non-linear least squares

In the case of a non-linear objective function one might utilize the Gauss-Newton method to find a solution. However in the case of NLS the objective function is not convex, and it can therefore be difficult to find a solution. The Gauss-Newton method can be viewed as a modified Newton's method with line search. It solves the following system to find the search direction p_k^{GN}:

$$J_k^\top J_k p_k^{GN} = -J_k^\top r_k.$$

Below is an implementation of the method.

```
# Simulate data
x <- seq(-2*pi, 2*pi, 0.1)
e <- rnorm(length(x), sd = 0.2)
true.b <- c(1,1)
y <- true.b[1]*sin(x*true.b[2]) + e

# Define functions and derivatives
f <- function(x, b) {b[1]%*%sin(b[2]%*%x)}
df.1 <- function(x, b) {sin(b[2]%*%x)}
df.2 <- function(x, b) {x%*%cos(b[2]%*%x)}

# gauss-newton implementation
gauss_newton <- function(b, x, y, f, df.1, df.2, anim, max_it = 10) {
  m <- length(x)
  n <- length(b)
  J <- matrix(nrow = m, ncol = n)
  for (i in 1:m) {
    J[i,1] <- df.1(x[i], b)
    J[i,2] <- df.2(x[i], b)
  }
  for (i in 1:max_it) {
```

```
    b <- b - solve(t(J)%*%J) %*% t(J) %*% t(f(x, b) - y)
    for (i in 1:m) {
      J[i,1] <- df.1(x[i], b)
      J[i,2] <- df.2(x[i], b)
    }
  }
  return(b)
}
gauss_newton(c(.5,.5), x, y, f, df.1, df.2, anim = TRUE, max_it = 1000)
```

```
##           [,1]
## [1,] 0.9918043
## [2,] 1.0020803
```

5 Constrained Optimization

5.1 Task 1

A boatyard produces three types of boats: cabin cruicers; racing sailboats and cruising sailboats. It takes 2 weeks to produce a cabin cruicer; 1 week to produce a racing sailboats and 3 weeks to produce a cruising sailboat. The boatyard is closed during the last week of December. The profit on each kind of boat is 5000 USD, 4000 USD and 6000 USD. Because of space considerations, the boatyard can finish at most 25 boats in one year (51 weeks).

To find the number of boats of each kind that will maximize the annual profit. We wish to maximize the profitfunction.

$$Z(x) = 5000x_1 + 4000x_2 + 6000x_3 \tag{48}$$

subject to the conditions that

$$2x_1 + 1x_2 + 3x_3 \leq 51 \tag{49}$$

$$x_1 + x_2 + x_3 \leq 25. \tag{50}$$

using the simplex method we find the optimal number of boats in the following way

-5000	-4000	-6000	0	0	
2	1	3	1	0	51
1	1	1	0	1	25

The last two columns represent the contraints. The place where we can maximize profit the most is in the 3rd column. We then have to choose whether we want pivot in the first or second row. To determine this we divide the last column with the third to see where we have a bottle neck. We conclude that the bottle neck is in the 1st row as $51/3 = 17 > 25/1 = 25$.

-4000	-2000	0	2000	0	
2/3	1/3	1	1/3	0	17
1/3	2/3	0	-1/3	1	8

We repeat the procedure again

0	0	0	1000	3000	
1/2	0	1	-1/2	0	13
1/2	1	0	-1/2	3/2	12

and we get the result

$$x_1 = 0, x_2 = 12, x_3 = 13. \tag{51}$$

Inserting in the profitfunction yields

$$Z(x) = 12 \cdot 4000\$ + 13 \cdot 6000\$ = 126.0000\$. \tag{52}$$

5.1.1 Altering the constraints

The owner is looking to increase the profit and is considering various options.

- Keeping the boatyard open all year round
- Increasing the capacity to 25 boasts
- Keeping the boatyard open all year round and icreasing the capacity to 26 boats

5.1.1.1 52 weeks and 25 boats

Below we will use R to solve the optimization problem using the same technique as above. First we will import the MASS library to display fractions for easier interpretation. Next we will create a function that can pivot the matrix for a given matrix and pivot position.

```
library(MASS)
```

```
##
## Attaching package: 'MASS'
```

```
## The following object is masked from 'package:dplyr':
##
##     select
```

```
pivot_matrix <- function(M, pivot){

  rp <- pivot[1]
  cp <- pivot[2]
  z <- (1:nrow(M))[-rp]

  M[rp,] <- M[rp,] / M[rp, cp]

  for (i in z){
    k <- M[i, cp]
    M[i, ] <- M[i,] - k* M[rp,]
  }
  M
}
```

We can now construct the matrix that corresponds to the optimization problem.

```
M <- matrix(c(-5,1,2, -4,1,1, -6,1,3, 0,1,0, 0,0,1, 0,25,52), nr = 3); M
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   -5   -4   -6    0    0    0
## [2,]    1    1    1    1    0   25
## [3,]    2    1    3    0    1   52
```

```
cp <- 3
M[-1, 6] / M[-1, cp]
```

```
## [1] 25.00000 17.33333
```

```
rp <- 3
M2 <- pivot_matrix(M, c(rp,cp));fractions(M2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   -1   -2    0    0    2  104
## [2,]  1/3  2/3    0    1 -1/3 23/3
## [3,]  2/3  1/3    1    0  1/3 52/3
```

```
cp <- 2
M2[-1, 6] / M2[-1, cp]
```

```
## [1] 11.5 52.0
```

```
rp <- 2
M3 <- pivot_matrix(M2, c(2,2));M3
```

```
##      [,1] [,2] [,3] [,4] [,5]  [,6]
```

```
## [1,]  0.0    0    0  3.0  1.0 127.0
## [2,]  0.5    1    0  1.5 -0.5  11.5
## [3,]  0.5    0    1 -0.5  0.5  13.5
```

As we don't suspect anyone to buy half a boat we must choose which number we would round up. If we construct 11 of the 2nd type and 14 of the third type we are able to store them but will use 53 weeks which we cannot. Therefore we choose to construct 12 of the 2nd type and 13 of the 3rd type. The profit in this case would amount to 126000. Note that will only use 51 weeks.

5.1.1.2 51 weeks and 26 boats

```
M <- matrix(c(-5,1,2, -4,1,1, -6,1,3, 0,1,0, 0,0,1, 0,26,51), nr = 3); M
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   -5   -4   -6    0    0    0
## [2,]    1    1    1    1    0   26
## [3,]    2    1    3    0    1   51
```

```
cp <- 3
M[-1, 6] / M[-1, cp]
```

```
## [1] 26 17
```

```
rp <- 3
M2 <- pivot_matrix(M, c(rp,cp));fractions(M2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   -1   -2    0    0    2  102
## [2,]  1/3  2/3    0    1 -1/3    9
## [3,]  2/3  1/3    1    0  1/3   17
```

```
cp <- 2
M2[-1, 6] / M2[-1, cp]
```

```
## [1] 13.5 51.0
```

```
rp <- 2
M3 <- pivot_matrix(M2, c(2,2));M3
```

```
##      [,1] [,2] [,3] [,4] [,5]  [,6]
## [1,]  0.0    0    0  3.0  1.0 129.0
## [2,]  0.5    1    0  1.5 -0.5  13.5
## [3,]  0.5    0    1 -0.5  0.5  12.5
```

If we construct 13 of the 2nd type and 13 of the third type we are able to store them but will use 52 weeks which we cannot. Therefore we choose to construct 14 of the 2nd type and 12 of the 3rd type. The profit in this case would amount to 128000. Note that we will only use 50

5.1.1.3 52 weeks and 26 boats

```
M <- matrix(c(-5,1,2, -4,1,1, -6,1,3, 0,1,0, 0,0,1, 0,26,52), nr = 3); M
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   -5   -4   -6    0    0    0
## [2,]    1    1    1    1    0   26
## [3,]    2    1    3    0    1   52
```

```
cp <- 3
M[-1, 6] / M[-1, cp]
```

```
## [1] 26.00000 17.33333
```

```
rp <- 3
M2 <- pivot_matrix(M, c(rp,cp));fractions(M2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   -1   -2    0    0    2  104
## [2,]  1/3  2/3    0    1 -1/3 26/3
## [3,]  2/3  1/3    1    0  1/3 52/3
```

```
cp <- 2
M2[-1, 6] / M2[-1, cp]
```

```
## [1] 13 52
```

```
rp <- 2
M3 <- pivot_matrix(M2, c(2,2));M3
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]  0.0    0    0  3.0  1.0  130
## [2,]  0.5    1    0  1.5 -0.5   13
## [3,]  0.5    0    1 -0.5  0.5   13
```

In this case we can construct 13 of 2nd and 3rd type while using all weeks and storage capacity to achieve a profit of 130000. We conclude that economically this is the best option.

5.2 Task 2

What should the ratio between diameter and height of a cylinder, with a volume of 1 liter, be to use the smallest amount of material. Solve the exercise using Lagrange multiplier technique.

The exercise can be rephrased to a optimization problem of the form: Minimize the objective function

$$A(r,h) = 2\pi r(h+r) \tag{53}$$

subject to

$$V(r,h) = \pi r^2 h = 1. \tag{54}$$

The lagrangian becomes

$$\mathcal{L}(r,h,\lambda) = 2\pi r(h+r) - \lambda(\pi r^2 h - 1). \tag{55}$$

We then find the partial derivatives

$$\mathcal{L}'_\lambda = -(\pi r^2 h - 1) \tag{56}$$
$$\mathcal{L}'_r = 2\pi h + 4\pi r - 2\lambda\pi r h \tag{57}$$
$$\mathcal{L}'_h = 2\pi r - \lambda\pi r^2 \tag{58}$$

Setting the gradient equal to 0 gives the a system of 3 equations with 3 unknowns

$$\nabla\mathcal{L} = \begin{bmatrix} -(\pi r^2 h - 1) \\ 2\pi h + 4\pi r - 2\lambda\pi r h \\ 2\pi r - \lambda\pi r^2 \end{bmatrix} = \mathbf{0} \tag{59}$$

The solution is

$$h \approx 1.0839 \tag{60}$$
$$r \approx 0.5419 \tag{61}$$
$$\lambda \approx -3.6904. \tag{62}$$

5.3 Task 3

Suppose the following situation: We have n independent stochastic variables $y_1, \ldots, y_n$ where $y_i \sim N(\mu, \sigma^2 v_i^2)$, and all v_i's are known and σ^2 is unknown.

5.3.1 Part 1

We wish to estimate μ. Let $\overline{y} = \frac{1}{n}\sum_i y_i$. What is $E[\overline{y}]$ and $V[\overline{y}]$? Can we find a better estimate for μ than a simple average.

Let $\bar{y} = \frac{1}{n}\sum_i y_i$. We want to estimate the mean and variance. We leave out σ^2 as this is an unknown constant.

$$E(\bar{y}) = E(\frac{1}{n}\sum_i y_i) = \frac{1}{n}E(\sum_i y_i) = \frac{1}{n}n\mu = \mu$$
$$V(\bar{y}) = \frac{1}{n^2}V(\sum_i y_i) = \frac{1}{n^2}\sum_i V(y_i) = \frac{1}{n^2}\sum_i v_i^2.$$

One would be able to come up with a better estimate by weighting the data points with lowest variance higher, but still consider all y_i's, as they still hold information about the true mean value.

5.3.2 Part 2

Let $\tilde{y} = \sum_i p_i y_i$ where $p = (p_1, ..., p_n)$ be a deterministic vector. what value of $\mathbf{p}$ gives $\tilde{y}$ the lowest possible variance? What restriction should we put on $p_1, \ldots p_n$ if we want $E(\tilde{y}) = \mu$?

First we calculate the variance

$$V(\tilde{y}) = V(\sum_i p_i y_i) = \sum_i p_i^2 V(y_i) = \sum_i p_i^2 v_i^2$$

The value for which we get the lowest variance is 0, however this is a trivial case that gets us no closer to the true mean. Next we find the mean

$$E(\tilde{y}) = E(\sum_i p_i y_i) = \sum_i p_i E(y_i) = \mu \sum_i p_i \Rightarrow \sum_i p_i = 1$$

This tells us that the weights must sum to one.

5.3.3 Part 3

In statistics we often wish to find a parameter estimate, that is unbiased and have the lowest possible variance. We wish to choose p so $\tilde{y}$ have mean value μ and lowest possible variance. Solve the problem using the Lagrange multiplier method.

The problem can be formulated as the following Lagrange function

$$L = \sum_i p_i^2 v_i^2 - \lambda(\sum_i p_i - 1)$$

Setting the partial derivatives equal to zero yields

$$\frac{\partial L}{\partial p_i} = 2p_i v_i^2 - \lambda = 0$$
$$\frac{\partial L}{\partial \lambda} = -\sum_i p_i + 1 = 0.$$

Thus we have a system of $k+1$ equations with $k+1$ unknowns. For $k=3$ we get the following system

$$\begin{bmatrix} 2v_1^2 & 0 & 0 & -1 & 0 \\ 0 & 2v_2^2 & 0 & -1 & 0 \\ 0 & 0 & 2v_3^2 & -1 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & 0 & \frac{-1}{2v_1^2} & 0 \\ 0 & 1 & 0 & \frac{-1}{2v_2^2} & 0 \\ 0 & 0 & 1 & \frac{-1}{2v_3^2} & 0 \\ 0 & 0 & 0 & \frac{1}{2v_1^2}+\frac{1}{2v_2^2}+\frac{1}{2v_3^2} & 1 \end{bmatrix}$$

$$\sim \begin{bmatrix} 1 & 0 & 0 & 0 & \frac{\frac{1}{2v_1^2}}{\frac{1}{2v_1^2}+\frac{1}{2v_2^2}+\frac{-1}{2v_3^2}} \\ 0 & 1 & 0 & 0 & \frac{\frac{1}{2v_2^2}}{\frac{1}{2v_1^2}+\frac{1}{2v_2^2}+\frac{1}{2v_3^2}} \\ 0 & 0 & 1 & 0 & \frac{\frac{1}{2v_3^2}}{\frac{1}{2v_1^2}+\frac{1}{2v_2^2}+\frac{1}{2v_3^2}} \\ 0 & 0 & 0 & 1 & \frac{1}{\frac{1}{2v_1^2}+\frac{1}{2v_2^2}+\frac{1}{2v_3^2}} \end{bmatrix}.$$

From this we get the equations

$$\lambda = \frac{1}{\frac{1}{2v_1^2}+\frac{1}{2v_2^2}+\frac{1}{2v_3^2}}$$

$$p_i = \frac{1}{2v_i^2}\lambda = \frac{\frac{1}{v_i^2}}{\sum_i \frac{1}{v_i^2}}$$

The same solution applies for arbitrary values of $k \in \mathbb{N}$.

5.4 Task 4

Consider a linear regression model of the form

$$y_i = \beta_0 + \beta_1 x_i + e_i,$$

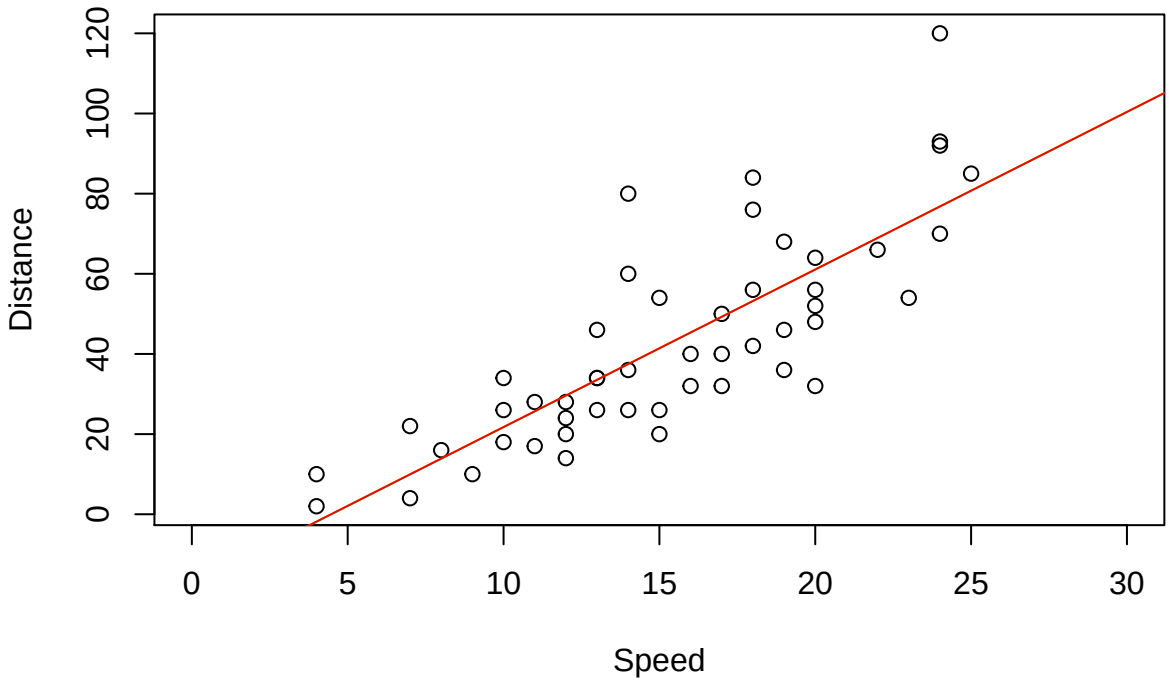
with the constraint $\beta_1 > 0$. In the chunk below we formulate and solve the problem using the `cars` dataset and the `CVXR` package.

```
suppressWarnings(library(CVXR, warn.conflicts = FALSE))
p <- 2
X <- cbind(rep(1,length(cars$speed)),cars$speed)
Y <- cars$dist
betaHat <- Variable(p)
objective <- Minimize(sum((Y - X %*% betaHat)^2))
problem.1 <- Problem(objective)
problem.2 <- Problem(objective, constraints = list(betaHat[2] >= 0))
result.1 <- solve(problem.1)
result.2 <- solve(problem.2)
```

Below we extract the coefficients from the two formulations

```
coef.1 <- result.1$getValue(betaHat)
coef.2 <- result.2$getValue(betaHat)
```

Below is a plot using the coeficients from the two problems.



Next we can explore the problem using the `nls` function with the formulated in such a way that we don't need to impose the constraint.

```
y <- cars$dist
x <- cars$speed
nl.fit.con <- nls(y ~ a + exp(b) * x, start = list(a = 1, b = 1))
nl.fit <- nls(y ~ a + b * x, start = list(a = 1, b = 1))
```

Below we can see the summaries form the two regressions.

```
summary(nl.fit.con)
```

```
##
## Formula: y ~ a + exp(b) * x
##
## Parameters:
##   Estimate Std. Error t value Pr(>|t|)
## a -17.5791     6.7584  -2.601   0.0123 *
## b   1.3693     0.1057  12.959   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
##
## Number of iterations to convergence: 4
## Achieved convergence tolerance: 7.849e-10
```

```r
summary(nl.fit)
```

```
##
## Formula: y ~ a + b * x
##
## Parameters:
##   Estimate Std. Error t value Pr(>|t|)
## a -17.5791     6.7584  -2.601   0.0123 *
## b   3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
##
## Number of iterations to convergence: 1
## Achieved convergence tolerance: 3.137e-09
```