Chapter 1

Function Maximization

Goals:

1. a few basic methods for function maximization

1.1 Function maximization

Many statistical methods involve the maximization (or minimization) of a function of one or several variables. To begin, we consider maximizing a function of a single variable $f(x)$ over an interval, say $a < x < b$ or $a \leq x \leq b$. Maximization is often carried out by solving for x or xs that satisfy

$$g(x) \;=\; f'(x) \;=\; 0$$

assuming $f(x)$ is differentiable. That is, we search for *roots* of the first derivative function $g(x)$.

I will discuss a few simple methods for function maximization, most of which require some smoothness on $f(x)$ and possibly $g(x)$.

1.2 Direct maximization

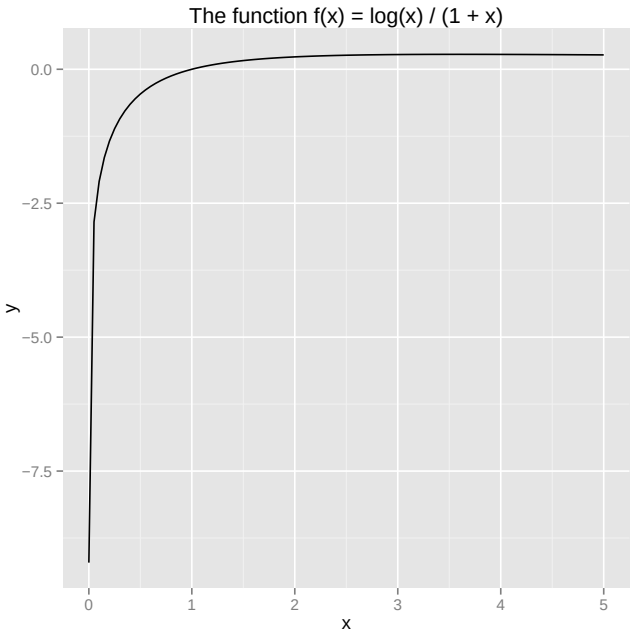
Direct maximization is effective in a vector or matrix programming language. It does not generalize well when $f(\underset{\sim}{x})$ is defined for $\underset{\sim}{x} \in \mathbb{R}^p$ where p is larger than 2, 3, or 4.

As an example, consider maximizing the function

$$f(x) = \frac{\log(x)}{1+x}, \qquad 0 \leq x \leq 5.$$

The basic idea is to finely divide the interval into a set of points on which the function is evaluated. Then we find the element in the vector of function values at which the function is maximized. Note that the maximization is *approximate*. The error in the approximation is a function of the coarseness of the grid.

```
# define function f(x)
f.f <- function(x) {
  log(x) / (1 + x)
}

# plot function
library(ggplot2)
p <- ggplot(data.frame(x = c(0.0001, 5)), aes(x))
p <- p + stat_function(fun = f.f)
p <- p + labs(title = "The function f(x) = log(x) / (1 + x)")
print(p)
```



Looking at the plot, we see that $f(x)$ initially is increasing then slowly decreases pass the point at which the maximum of $f(x)$ occurs.

```
# a grid of x-values
x <- seq(0.0001, 5, by = 0.0001)
# evaluate the function over the grid
f.x <- f.f(x)
# determine the index of the maximum value
ind <- which(f.x == max(f.x))
# print the value of x and f(x) at the maximum
c(x[ind], f.x[ind])

## [1] 3.5911 0.2785
```

The max occurs at 3.5911 and the maximum value is 0.2785.

1.3 Bisection (bracketing)

This is the simplest, but slowest,, method to solve

$$g(x) \;=\; f'(x) \;=\; 0.$$

However, it is "guaranteed to work" provided simple precautions are taken. For simplicity, we assume $g(x)$ is *continuous*. The idea is to find an interval $a \leq x \leq b$ on which $g(x)$ is monotonic (either strictly increasing or decreasing) and such that $g(x)$ changes sign (that is, $g(a)g(b) < 0$). This implies there is a unique root in this interval.

The basic idea of bisection is to sequentially halve the interval by checking whether the root is to the left or right of the interval midpoint and then modifying the interval appropriately. That is, if at

$$x_0 \;=\; (a+b)/2$$

we have

$$\begin{aligned} g(a)g(x_0) > 0 \;&\Rightarrow\; g(x) \text{ has same sign at } a \text{ and } x_0 \\ &\Rightarrow\; \text{root is to the } \textit{right} \text{ of } x_0 \\ &\Rightarrow\; \text{redefine } a = x_0 \end{aligned}$$

else if

$$\begin{aligned} g(a)g(x_0) < 0 \;&\Rightarrow\; g(x) \text{ changes sign between } a \text{ and } x_0 \\ &\Rightarrow\; \text{root is to the } \textit{left} \text{ of } x_0 \\ &\Rightarrow\; \text{redefine } b = x_0. \end{aligned}$$

The process iterates until $b - a \leq \varepsilon$ (a user-specified small value).

Remarks

1. By construction, if $g(a)g(x_0) = 0$, then we know that x_0 is the root. One could build this into the routine but because of machine roundoff it is not likely that the machine representation of $g(a)g(x_0)$ will give you zero exactly, so the extra coding probably does not pay off.

2. Bisection is relatively slow because it ignores information about how quickly $g(x)$ changes over $[a, b]$, that is, it does not use information on derivatives of $g(x)$.

3. If we let $[a_i, b_i]$ be the search interval at the ith step with $[a_0, b_0]$ as the initial interval, then

$$b_i - a_i \;=\; 2^{-i}(b_0 - a_0)$$

Given the user defined ε, we have

$$\begin{aligned} & b_i - a_i \;=\; 2^{-i}(b_0 - a_0) \;<\; \varepsilon \\ \Leftrightarrow\; & -i + \log_2(b_0 - a_0) \;<\; \log_2(\varepsilon) \\ \Leftrightarrow\; & i > \log_2\left(\frac{b_0 - a_0}{\varepsilon}\right). \end{aligned}$$

That is, we need approximately that

$$\log_2\left(\frac{b_0 - a_0}{\varepsilon}\right)$$

steps for convergence. Reducing ε by a factor of 10 (that is, adding an additional decimal place of precision) requires an additional

$$\log_2(10) \;=\; 3.3 \;\doteq\; 4$$

iterations.

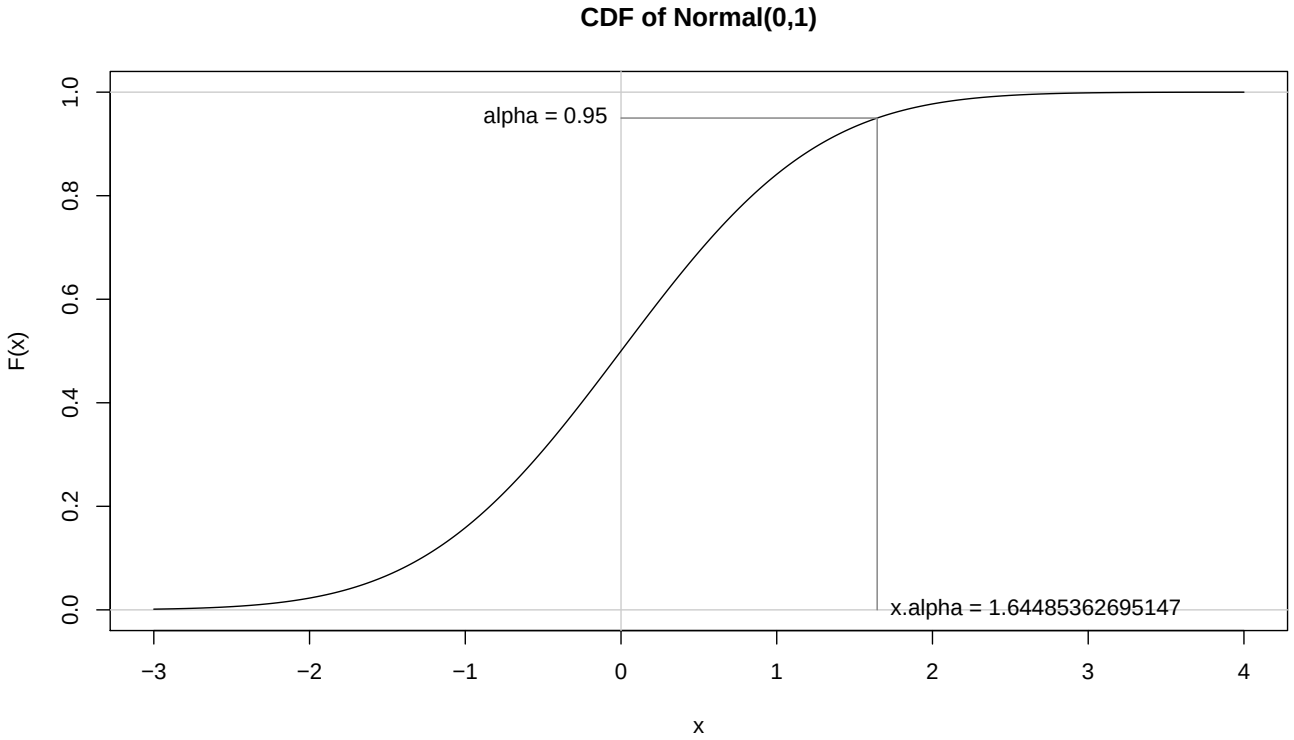
4. The method is *guaranteed* to converge to within ε (or 0.5ε) of the root, provided $g(x)$ is continuous.

Example: Quantiles of a rv X Suppose a univariate rv X has a cumulative distribution function

$$F(t) \;=\; \Pr(X \leq t), \tag{1.1}$$

where, for simplicity, we assume $F(t)$ is continuous and strictly increasing. Given $0 < \alpha < 1$, the αth percentile of X is the unique value x_α so that

$$\begin{aligned} F(x_\alpha) \;&=\; \alpha \\ F(x_\alpha) - \alpha \;&=\; 0. \end{aligned}$$



Given $F(t)$, it is relatively straightforward to use bisection to compute x_α. R has routines to compute quantiles for many standard distributions, typically using more complex approximations based on rational function expansions and the like. The following function illustrates a bisection evaluation of percentiles for a standard normal distribution.

```
## function bisect.qnorm - standard normal quantiles
# input:    alpha = probability for desired quantile
# output:   xa = desired quantile with max error eps=0.001
#           provided alpha is not extreme
bisect.qnorm <- function(alpha, eps = 0.001, a = -5, b = 5, sw.more = 0) {
  # First some error catching
  if(length(alpha) > 1) {
    xa <- NULL
    warning("alpha must be a scalar.")
    return(xa)
  }
  if((alpha < 0) | (alpha > 1)) {
    warning("alpha out of bounds, specify 0 < alpha < 1")
    xa <- NULL
    return(xa)
  }
  if(alpha <= pnorm(a)) {
    warning("alpha specified less than lower bound, pnorm(a)")
    xa <- NULL
    return(xa)
  }
  if(alpha >= pnorm(b)) {
    warning("alpha specified greater than upper bound, pnorm(b)")
    xa <- NULL
    return(xa)
  }
  if(alpha == 0.5) {  # what happens if we don't have this?
    xa <- 0
    return(xa)
  }

  if (sw.more != 1) { # don't provide additional output
    while ((b - a) > eps) {
      x0 <- a + (b - a) / 2
      if ((pnorm(x0) - alpha) < 0) {
        a <- x0
      } else {
        b <- x0
      }
    }
    xa <- a + (b - a) / 2
    return(xa)
  }
  if (sw.more == 1) { # provide additional output for creating plot later
    ii <- 1
```

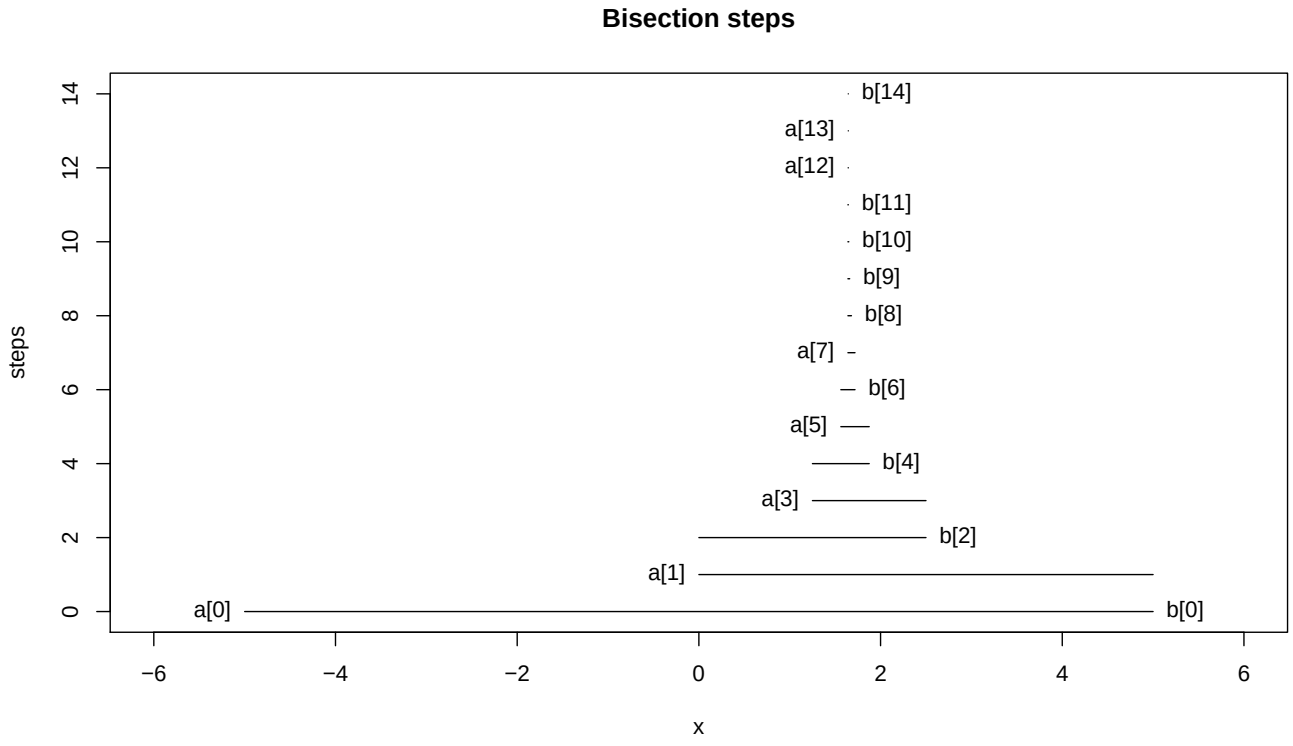
```
    while ((b[ii] - a[ii]) > eps) {
      x0 <- a[ii] + (b[ii] - a[ii]) / 2
      ii <- ii + 1
      if ((pnorm(x0) - alpha) < 0) {
        a[ii] <- x0
        b[ii] <- b[ii-1]
      } else {
        a[ii] <- a[ii-1]
        b[ii] <- x0
      }
    }
    xa <- a[ii] + (b[ii] - a[ii]) / 2

    out <- list()
    out$xa     <- xa
    out$a      <- a
    out$b      <- b
    # since step 0 is ii=1, n.iter is the expected maximum for (ii - 1)
    out$n.iter <- ceiling(log((b[1] - a[1]) / eps, base = 2))
    out$ii     <- ii
    return(out)
  }
}
```

Demonstration:

```
# running function with more output to create detailed plot of iterations
out <- bisect.qnorm(0.95, sw.more = 1)
out


## $xa
## [1] 1.645
##
## $a
##  [1] -5.000  0.000  0.000  1.250  1.250  1.562  1.562  1.641  1.641
## [10]  1.641  1.641  1.641  1.643  1.644  1.644
##
## $b
##  [1] 5.000 5.000 2.500 2.500 1.875 1.875 1.719 1.719 1.680 1.660 1.650
## [12] 1.646 1.646 1.646 1.645
##
## $n.iter
## [1] 14
##
```

```
## $ii
## [1] 15
```

```
# create plot for iteration steps
plot(0, 0, xlim = c(out$a[1], out$b[1])*1.2, ylim = c(0, out$ii-1), type = "n",
    main = "Bisection steps", xlab = "x", ylab = "steps")
ii <- 1;
  lines(c(out$a[ii], out$b[ii]), rep(ii-1, 2))
  text(out$a[ii], ii-1, labels = paste("a[", ii-1, "]", sep=""), pos = 2)
  text(out$b[ii], ii-1, labels = paste("b[", ii-1, "]", sep=""), pos = 4)
for (ii in 2:out$ii) {
  lines(c(out$a[ii], out$b[ii]), rep(ii-1, 2))
  if (out$a[ii] != out$a[ii-1]) {
    text(out$a[ii], ii-1, labels = paste("a[", ii-1, "]", sep=""), pos = 2)
  }
  if (out$b[ii] != out$b[ii-1]) {
    text(out$b[ii], ii-1, labels = paste("b[", ii-1, "]", sep=""), pos = 4)
  }
}
```



Remarks

1. The function checks some error conditions before performing bisection.

2. The function uses R's `pnorm()` function for evaluating the normal cdf function.

3. Since I know that $g(x)$ is *increasing*, I also know that $g(a) < 0$ and $g(b) < 0$ at each iteration. Thus, I do not need to check $g(a)g(x_0) > 0$ or $g(a)g(x_0) < 0$ at interval midpoint $x = (a + b)/2$. I only have to check whether $g(x_0) > 0$ or $g(x_0) < 0$ for which endpoint to change.

4. Default convergence criterion is $\varepsilon = 0.001$, so we need approximately

$$\log_2((5 - (-5))/0.001) \;=\; \log_2(10000) \;=\; 13.29 \;\doteq\; 14$$

   steps to find the quantile.

5. Can the symmetry of the Normal$(0, 1)$ distribution be used to our advantage here? Think.

Other examples:

```
# R qnorm() function
format(qnorm(0.95), digits=16)

## [1] "1.644853626951472"

# our bisection function
format(bisect.qnorm(0.95), digits=16)

## [1] "1.64459228515625"

format(bisect.qnorm(0.95, a = 2, b = 3), digits=16)
```

```
## Warning:  alpha specified less than lower bound, pnorm(a)

## [1] "NULL"

# more precision
format(bisect.qnorm(0.95, eps = 1e-7), digits=16)

## [1] "1.644853614270687"

format(bisect.qnorm(0.95, eps = 1e-10), digits=16)

## [1] "1.644853626967233"

format(bisect.qnorm(0.95, eps = 1e-15), digits=16)

## [1] "1.644853626951472"


# other examples
bisect.qnorm(0.025)

## [1] -1.96

bisect.qnorm(0.975)

## [1] 1.96

bisect.qnorm(0.5)

## [1] 0
```

1.4 Newton-Raphson (NR)

This is a very popular derivative-based method for solving

$$g(x) \;=\; f'(x) \;=\; 0, \qquad a \le x \le b.$$

This method requires $g(x)$ to be differentiable on $[a, b]$ and $g'(x) = 0$ at a root.

Suppose x^* satisfies $g(x^*) = 0$, that is x^* is a root. Then for x close to x^*, using the linear Taylor series approximation,

$$\begin{aligned} 0 &= g(x^*) \doteq g(x) + g'(x)(x^* - x) \qquad \text{or} \\ x^* - x &\doteq \frac{-g(x)}{g'(x)} \qquad \text{implying} \\ x^* &\doteq x - \frac{g(x)}{g'(x)}. \end{aligned}$$

The purpose of course is to find x^*! However, this relationship suggests an iterative scheme for finding x^*, starting from an initial guess x_0, which is hopefully close to x^*:

$$\begin{aligned} x_1 &= x_0 - \frac{g(x_0)}{g'(x_0)} \\ &\vdots \\ x_{i+1} &= x_i - \frac{g(x_i)}{g'(x_i)} \qquad i = 0, 1, 2, \ldots. \end{aligned}$$

Then we iterate until

$$|x_{i+1} - x_i| < \varepsilon.$$

It is easy to see that this method works for finding a root if we can get the difference $|x_{i+1} - x_i|$ between successive approximations arbitrarily small. That is,

$$|x_{i+1} - x_i| < \varepsilon \qquad \text{implies} \qquad \frac{g(x_i)}{g'(x_i)} < \varepsilon.$$

If $g'(x_i)$ is bounded away from zero then we must have $g(x_i) \doteq 0$, that is our approximation is close to a root x^*. Formally, one can show that if $x_i \to x^*$ as $i \to \infty$ then $g(x_i^*) = 0$, that is, x^* is a root.

Remarks

1. Iterating until $|x_{i+1} - x_i| < \varepsilon$ is an example of an *absolute* convergence criterion, which is dependent on the units for x.

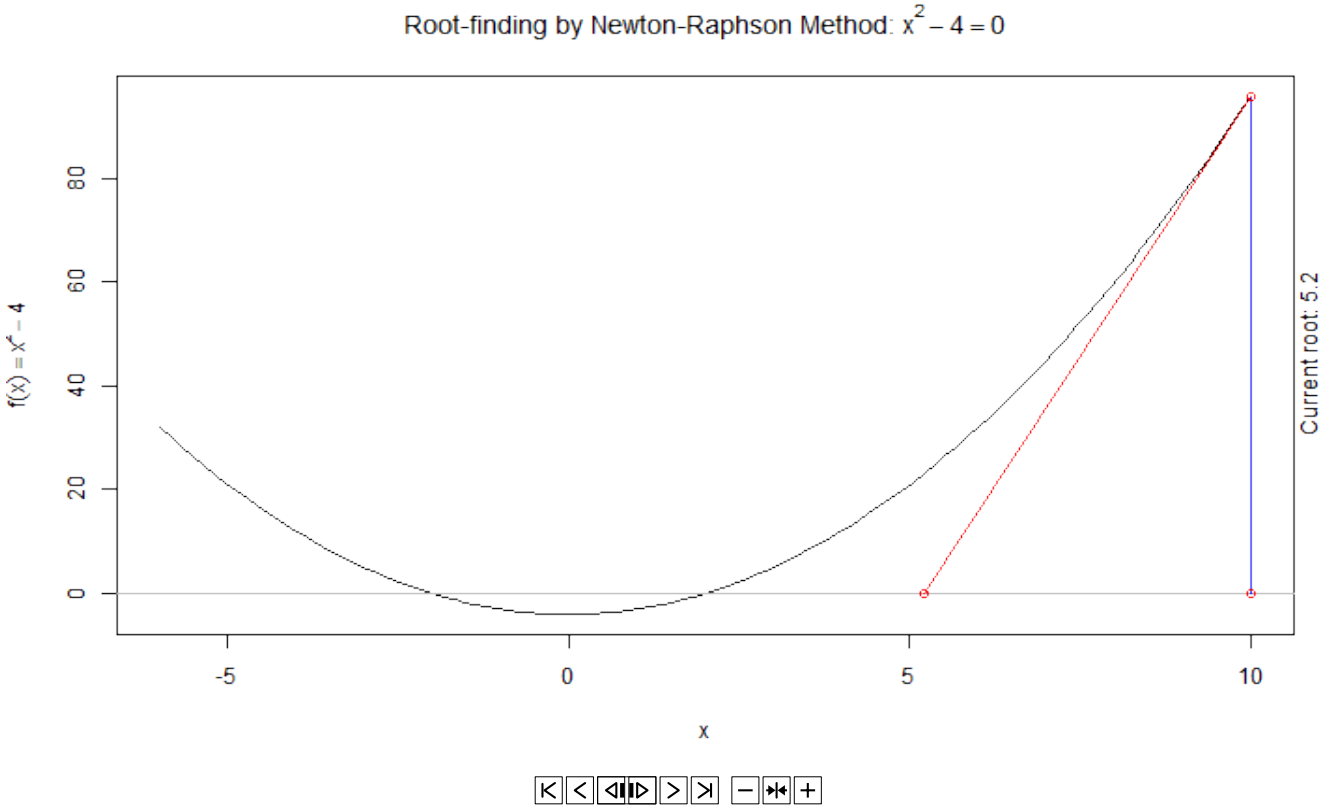
2. A *relative* convergence criterion would require iterating until The relative change in successive approximations,

$$\frac{|x_{i+1} - x_i|}{|x_i|} < \varepsilon,$$

   which makes sense provided the root is not near 0.

3. What is NR doing? The following animations illustrates the idea. At the $(i+1)$th step, we are approximating $g(x)$ by its tangent line at x_i, whose root serves as the next approximation to the root of $g(x)$.

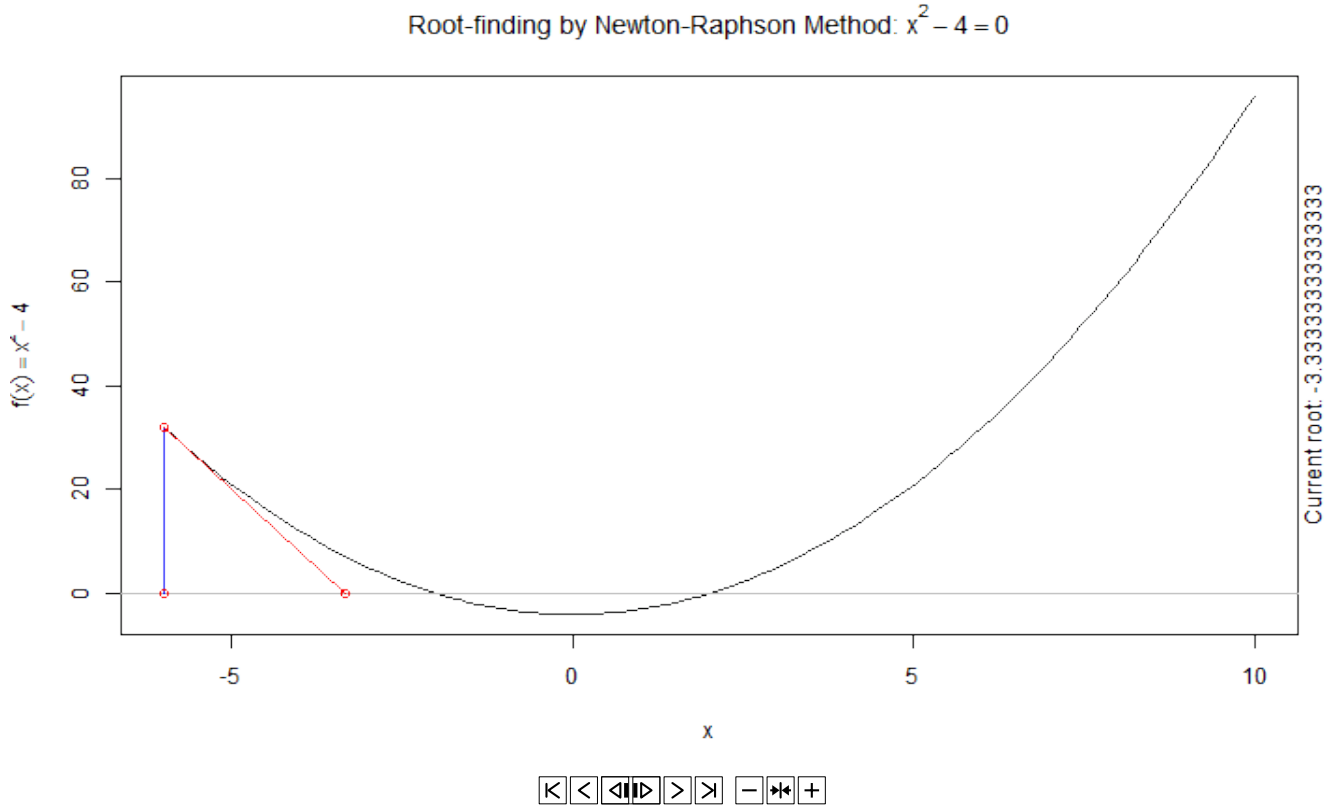
```
library(animation)
# FUN is the function to find the root of (derivative of function to max/minimize)
newton.method(function(x) x^2 - 4, init = 10, rg = c(-6, 10), tol = 0.001)
```



4. Convergence of NR depends on the form of $g(x)$ and the choice of the *starting value*, that is, how close x_0 is to x^*.

   With multiple roots, different initial values will find different roots.

```
library(animation)
newton.method(function(x) x^2 - 4, init = -6, rg = c(-6, 10), tol = 0.001)
```



Here are two results about starting values:

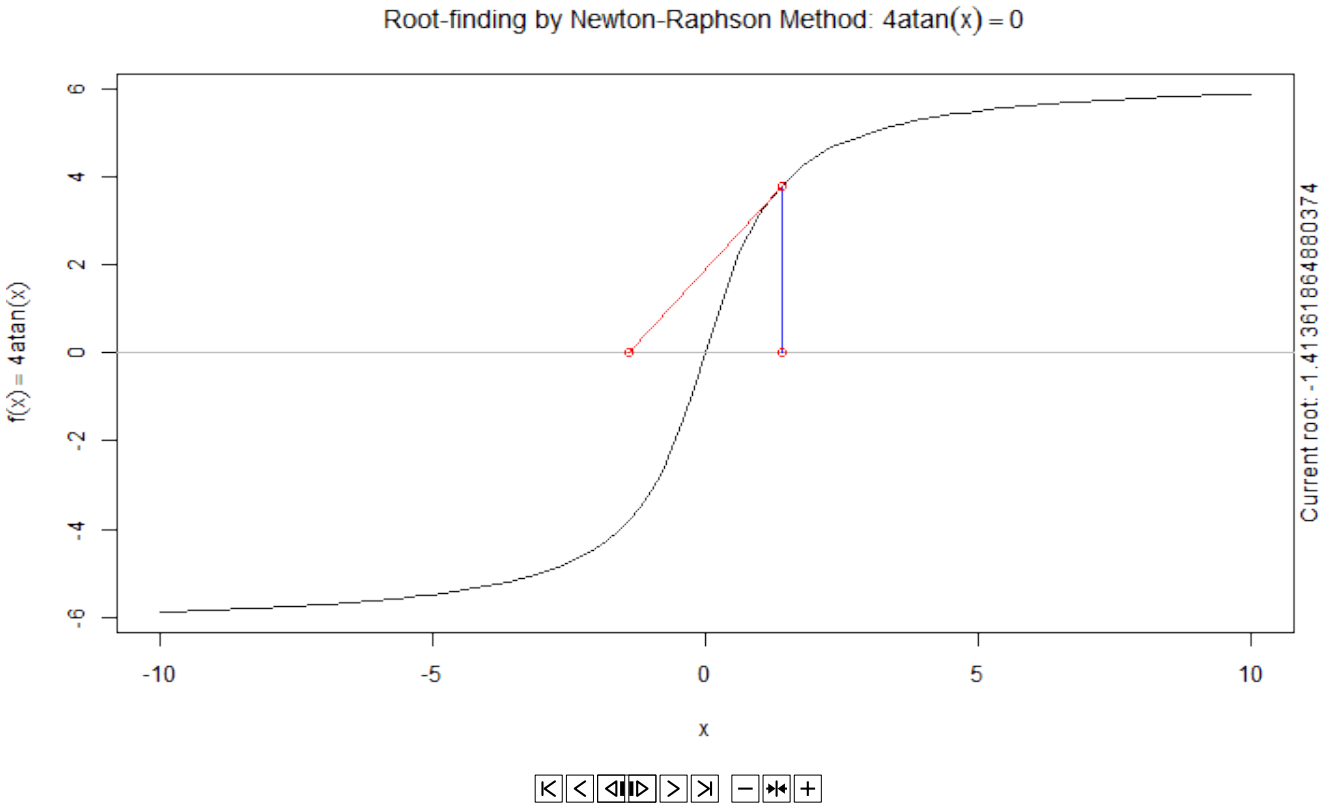
- If $g(x)$ has two continuous derivatives and x^* is a simple root of $g(x)$, (that is, $g'(x) \neq 0$) then there exists a neighborhood of x^* for which NR converges to x^* for any x_0 in that neighborhood.
- If, in addition, $g(x)$ is convex (that is, $g''(x) > 0$ for all x) then NR converges from any starting point.

Most functions will not be convex, so the first result is most practical. However, it does not tell you how to find the neighborhood from which NR converges regardless of the starting value. The first result suggests convergence will occur if you start close to x^*.

The following example shows where NR does not converge from the designated starting value. In this example, the distance between approximations $|x_{i+1} - x_i|$ is increasing, which is a clear indication

of a problem! A slightly different starting value (for example, 1.3) converges nicely.

```
library(animation)
newton.method(function(x) 4 * atan(x), init = 1.4, rg = c(-10, 10), tol = 0.001)
```



5. Convergence order (or rate): If we let $\varepsilon_i = |x_{i+1} - x_i|$ be the error of our approximation of the root x^* at the ith step, then we can show with NR that if $\varepsilon_i \to 0$ (that is, if NR converges) then

$$\varepsilon_{i+1} \;=\; \text{constant} \times \varepsilon_i^2.$$

That is, the size of the $(i+1)$th step error is proportional to the square of the ith step error. This is known as *quadratic convergence*, in contrast to bisection which has a *linear convergence*:

$$\varepsilon_{i+1} \;=\; \text{constant} \times \varepsilon_i \;=\; 0.5\varepsilon_i \qquad \text{constant} = 0.5 \text{ for bisection.}$$

Quadratic convergence is typically *faster* than linear convergence. That is, you should expect NR to converge in fewer iterations. The caveat is robustness: NR is not guaranteed to converge in general, but if it does, it converges faster typically than bisection.

1.5 Secant method

The secant method (Regula Falsi method) modifies NR iteration

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

by using a numerical approximation to $g'(x_i)$ based on x_i and x_{i-1}:

$$g'(x_i) \doteq \frac{g(x_i) - g(x_{i-1})}{x_i - x_{i-1}}$$

which gives

$$x_{i+1} = x_i - \frac{g(x_i)}{g(x_i) - g(x_{i-1})}(x_i - x_{i-1}).$$

This approach is especially popular when $g'(x_i)$ it is difficult to compute

Remarks

1. The secant method needs two starting values.
2. This is called the secant method because x_{i+1} is the abscissa of the point of intersection between the secant line through $(x_i, g(x_i))$ and $(x_{i-1}, g(x_{i-1}))$ and the x-axis.
3. As with NR, the secant method is sensitive to starting values.

4. Establishing convergence of the secant method is a bit more delicate than either NR or bisection. I will note that if the secant method converges, then

$$\varepsilon_{i+1} \;=\; \text{constant} \times \varepsilon_i^p,$$

where $p = 0.5(1 + \sqrt{5}) \doteq 1.618$ (the golden ratio). This is called *super-linear* convergence: faster than bisection, but slower than NR.

1.6 Illustration of NR and Secant methods

1.6.1 NR method

We've shown how to directly maximize

$$f(x) \;=\; \frac{\log(x)}{1+x}, \qquad 0 \le x \le 5.$$

Let's see how well the NR and secant methods work here. For both methods, we are searching for a root of the function

$$\begin{aligned} g(x) \;=\; f'(x) \;&=\; \frac{1}{x(1+x)} - \frac{\log(x)}{(1+x)^2} \\ &=\; \frac{1}{1+x}\left(\frac{1}{x} - f(x)\right). \end{aligned}$$

Also note that

$$\begin{aligned} g'(x) \;=\; f''(x) \;&=\; -\frac{1}{(1+x)^2}\left(\frac{1}{x} - f(x)\right) + \frac{1}{1+x}\left(-\frac{1}{x^2} - f'(x)\right) \\ &=\; -\frac{1}{1+x}\frac{1}{1+x}\left(\frac{1}{x} - f(x)\right) + \frac{1}{1+x}\left(-\frac{1}{x^2} - f'(x)\right) \\ &=\; -\frac{1}{1+x}f'(x) - \frac{1}{1+x}\left(\frac{1}{x^2} + f'(x)\right) \\ &=\; -\frac{1}{1+x}\left(\frac{1}{x^2} + 2f'(x)\right). \end{aligned}$$

For NR, the iterative scheme is

$$\begin{aligned} x_{i+1} \;&=\; x_i - \frac{g(x_i)}{g'(x_i)} \\ &=\; x_i + b(x_i), \end{aligned}$$

where

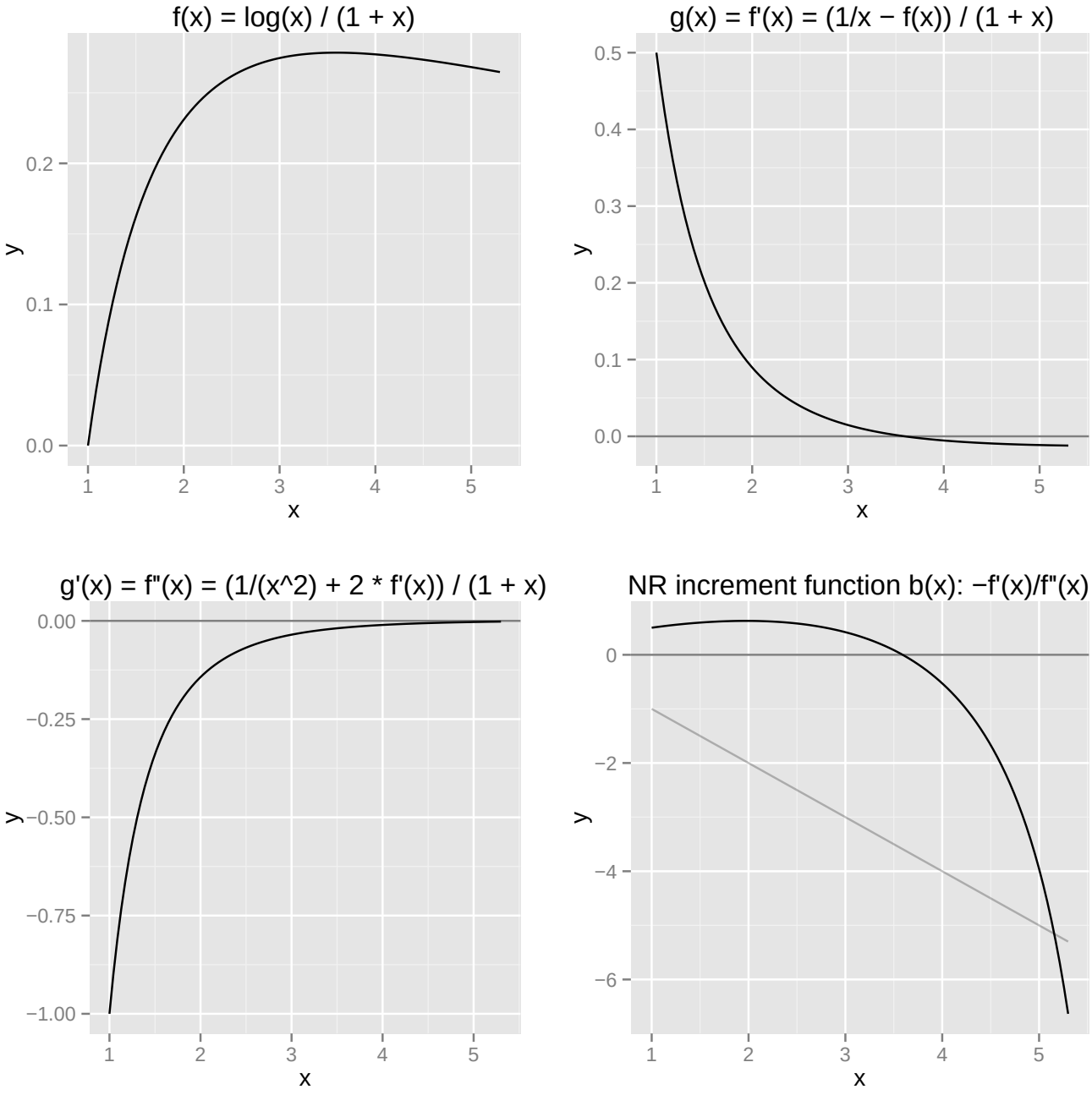
$$b(x_i) \;=\; \text{increment function at } x_i \;=\; -\frac{g(x_i)}{g'(x_i)},$$

that is, the increment function tells you how much the estimate changes.

Before doing NR, let us look at some plots of the function and its derivatives.

```
# f(x), function
f.f <- function(x) {
  log(x) / (1 + x)
}

# f'(x), 1st derivative
f.fp <- function(x) {
  (1/x - f.f(x)) / (1 + x)
}

# f''(x), 2nd derivative
```

```
f.fpp <- function(x) {
  - (1/(x^2) + 2 * f.fp(x)) / (1 + x)
}

# plot function
library(ggplot2)
p1 <- ggplot(data.frame(x = c(1, 5.3)), aes(x))
p1 <- p1 + stat_function(fun = f.f)
p1 <- p1 + labs(title = "f(x) = log(x) / (1 + x)")
#print(p1)

p2 <- ggplot(data.frame(x = c(1, 5.3)), aes(x))
p2 <- p2 + geom_hline(yintercept = 0, alpha = 0.5)
p2 <- p2 + stat_function(fun = f.fp)
p2 <- p2 + labs(title = "g(x) = f'(x) = (1/x - f(x)) / (1 + x)")
#print(p2)

p3 <- ggplot(data.frame(x = c(1, 5.3)), aes(x))
p3 <- p3 + geom_hline(yintercept = 0, alpha = 0.5)
p3 <- p3 + stat_function(fun = f.fpp)
p3 <- p3 + labs(title = "g'(x) = f''(x) = (1/(x^2) + 2 * f'(x)) / (1 + x)")
#print(p3)

p4 <- ggplot(data.frame(x = c(1, 5.3)), aes(x))
p4 <- p4 + geom_hline(yintercept = 0, alpha = 0.5)
p4 <- p4 + stat_function(fun = function(x) {-f.fp(x) / f.fpp(x)})
p4 <- p4 + stat_function(fun = function(x) {-x}, alpha = 0.25) # for later discussion
p4 <- p4 + labs(title = "NR increment function b(x): -f'(x)/f''(x)")
#print(p4)

library(gridExtra)
grid.arrange(p1, p2, p3, p4, ncol=2)
```



Looking at the plots of $g(x)$, we see that $x^* \doteq 3.5$ or so. Because of continuity and $g'(x^*) \neq 0$ we know that NR will converge, provided our starting value is close to the root.

The plot of the increment function shows you that if you start to the left of the root you will move to the right ($b(x_i) > 0$) while if you start to the right of the route you will move left ($b(x_i) < 0$). In both cases, you're

moving in the right direction! (There is more to this story!)

NR is easy to program if you don't build in any safeguards. Here is a simple algorithm:

initialize $x_{\text{old}} =$ old guess, $x_{\text{new}} =$ new guess

iterate while $|x_{\text{new}} - x_{\text{old}}| > \varepsilon$ (absolute convergence

- update old guess: $x_{\text{old}} = x_{\text{new}}$
- update new guess: $x_{\text{new}} = x_{\text{old}} - g(x_{\text{old}})/g'(x_{\text{old}})$

A problem here is that you may never satisfy the convergence criterion. A simple way to avoid this problem is to keep track of how many iterations you have performed, and do not allow this to exceed a prespecified limit.

```
# NR routine for finding root of g(x) = 0.
# Requires predefined g(x) and gp(x) = deriv of g(x)
# The iteration is controlled by:
#   eps   = absolute convergence criterion
#   maxit = maximum allowable number of iterations
# Input:  xnew = user prompted starting value
# Output: number of root, steps, and note
f.NR <- function(g, gp, xnew = 1, eps = 0.001, maxit = 35) {
  xold <- -Inf   # needed so argument in while() loop is defined

  i <- 1;   # initial iteration index

  NR.hist <- data.frame(i, xnew, diff = abs(xnew - xold)) # iteration history
  while ((i <= maxit) & (abs(xnew - xold) > eps)) {
    i <- i + 1                          # increment iteration
    xold <- xnew                        # old guess is current guess
    xnew <- xold - g(xold) / gp(xold)  # new guess

    NR.hist <- rbind(NR.hist, c(i, xnew, abs(xnew - xold))) # iteration history
  }

  out <- list()
  out$root <- xnew
  out$iter <- i
```

```
  out$hist <- NR.hist
  if (abs(xnew - xold) <= eps) {
    out$note <- paste("Absolute convergence of", eps, "satisfied")
  }
  if (i > maxit) {
    out$note <- paste("Exceeded max iterations of ", maxit)
  }
  return(out)
}
```

A few illustrations of our NR function follow:

```
out <- f.NR(f.fp, f.fpp)
out
```

```
## $root
## [1] 3.591
##
## $iter
## [1] 9
##
## $hist
##   i  xnew      diff
## 1 1 1.000       Inf
## 2 2 1.500 5.000e-01
## 3 3 2.095 5.949e-01
## 4 4 2.719 6.242e-01
## 5 5 3.244 5.245e-01
## 6 6 3.526 2.828e-01
## 7 7 3.589 6.224e-02
## 8 8 3.591 2.471e-03
## 9 9 3.591 3.702e-06
##
## $note
## [1] "Absolute convergence of 0.001 satisfied"
```

```
# function value at root
f.f(out$root)
```

```
## [1] 0.2785
```

```
# try for a few more starting values
out <- f.NR(f.fp, f.fpp, xnew = 0.001)
out
```

```
## $root
## [1] 3.591
##
## $iter
## [1] 19
##
## $hist
##     i     xnew     diff
## 1   1 0.001000      Inf
## 2   2 0.002005 0.001005
## 3   3 0.004026 0.002022
## 4   4 0.008109 0.004082
## 5   5 0.016393 0.008284
## 6   6 0.033291 0.016898
## 7   7 0.067767 0.034476
## 8   8 0.136844 0.069077
## 9   9 0.267472 0.130629
## 10 10 0.489561 0.222088
## 11 11 0.823230 0.333669
## 12 12 1.274981 0.451751
## 13 13 1.834557 0.559576
## 14 14 2.458315 0.623758
## 15 15 3.044364 0.586049
## 16 16 3.440809 0.396445
## 17 17 3.578174 0.137365
## 18 18 3.591021 0.012846
## 19 19 3.591121 0.000101
##
## $note
## [1] "Absolute convergence of 0.001 satisfied"


# increased precision
out <- f.NR(f.fp, f.fpp, xnew = 3.5, eps = 1e-12)
out


## $root
## [1] 3.591
##
## $iter
## [1] 6
##
## $hist
##   i  xnew      diff
## 1 1 3.500       Inf
```

```
## 2 2 3.586 8.626e-02
## 3 3 3.591 4.845e-03
## 4 4 3.591 1.427e-05
## 5 5 3.591 1.232e-10
## 6 6 3.591 0.000e+00
##
## $note
## [1] "Absolute convergence of 1e-12 satisfied"

out <- f.NR(f.fp, f.fpp, xnew = 5.1)
out


## $root
## [1] 3.591
##
## $iter
## [1] 11
##
## $hist
##     i   xnew      diff
## 1   1 5.1000       Inf
## 2   2 0.4174 4.6825896
## 3   3 0.7189 0.3015235
## 4   4 1.1381 0.4191518
## 5   5 1.6703 0.5321824
## 6   6 2.2835 0.6132815
## 7   7 2.8942 0.6106185
## 8   8 3.3577 0.4635227
## 9   9 3.5608 0.2030951
## 10 10 3.5906 0.0297850
## 11 11 3.5911 0.0005505
##
## $note
## [1] "Absolute convergence of 0.001 satisfied"

# can not be evaluated (complex numbers)
out <- f.NR(f.fp, f.fpp, xnew = 5.2)

## Warning:  NaNs produced
## Warning:  NaNs produced
## Error:  missing value where TRUE/FALSE needed
```

General results

1. There is rapid convergence for $0.001 \leq x_0 \leq 5$.

2. The number of steps for convergence decreases as $|x_0 - x^*|$ decreases.

3. The routine "blows up", or fails to converge, for $x_0 > 5.2$ because the increment function $b(x_i) = -g(x_i)/g'(x_i) < -x$. That is

$$x_{i+1} \;=\; x_i - \frac{g(x_i)}{g'(x_i)} \;<\; 0$$

   for $x_0 > 5.2$ or so. The function $g(x)$ is undefined for $x \leq 0$ (unless we want our numbers to be complex, which we don't), so the routine "crashes" for starting values $x_0 > 5.2$.

   A simple fix here would be to redefine any negative guesses for x to be slightly positive ($x = 0.01$) to force $g(x)$ and $g'(x)$ to be evaluated only for $x > 0$.

1.6.2 Secant method

The secant method is also easy to program. The algorithm for the iteration

$$x_{i+1} \;=\; x_i - \frac{g(x_i)}{g(x_i) - g(x_{i-1})}(x_i - x_{i-1})$$

is very similar to NR, except that two starting values are required.

The script below provides this function and the performance is similar to NR.

```
# Secant routine for finding root of g(x) = 0.
# Requires predefined g(x)
# The iteration is controlled by:
#   eps   = absolute convergence criterion
```

```
#   maxit = maximum allowable number of iterations
# Input:  xnew = user prompted starting value
# Input:  xtwo = user prompted second starting value
# Output: number of root, steps, and note
f.secant <- function(g, xnew = 1, xtwo = 2, eps = 0.001, maxit = 35) {
  i <- 1;   # initial iteration index

  NR.hist <- data.frame(i, xnew, xtwo, diff = abs(xnew - xtwo)) # iteration history
  while ((i <= maxit) & (abs(xnew - xtwo) > eps)) {
    i <- i + 1                          # increment iteration
    xold <- xtwo                        # 2nd previous guess
    xtwo <- xnew                        # previous guess
    xnew <- xtwo - g(xtwo) / (g(xtwo) - g(xold)) * (xtwo - xold)  # new guess

    NR.hist <- rbind(NR.hist, c(i, xnew, xtwo, abs(xnew - xold))) # iteration history
  }

  out <- list()
  out$root <- xnew
  out$iter <- i
  out$hist <- NR.hist
  if (abs(xnew - xold) <= eps) {
    out$note <- paste("Absolute convergence of", eps, "satisfied")
  }
  if (i > maxit) {
    out$note <- paste("Exceeded max iterations of ", maxit)
  }
  return(out)
}
```

A few illustrations of our secant function follow:

```
out <- f.secant(f.fp)
out


## $root
## [1] 3.591
##
## $iter
## [1] 10
##
## $hist
##     i  xnew  xtwo     diff
## 1   1 1.000 2.000 1.000000
```

```
## 2   2 2.218 1.000 0.218473
## 3   3 2.395 2.218 1.394550
## 4   4 2.918 2.395 0.699813
## 5   5 3.232 2.918 0.837341
## 6   6 3.469 3.232 0.550437
## 7   7 3.567 3.469 0.334848
## 8   8 3.589 3.567 0.120641
## 9   9 3.591 3.589 0.024357
## 10 10 3.591 3.591 0.001757

# function value at root
f.f(out$root)

## [1] 0.2785

# try for a few more starting values
out <- f.secant(f.fp, xnew = 1, xtwo = 3)
out

## $root
## [1] 3.591
##
## $iter
## [1] 8
##
## $hist
##   i  xnew  xtwo    diff
## 1 1 1.000 3.000 2.00000
## 2 2 3.060 1.000 0.06045
## 3 3 3.114 3.060 2.11384
## 4 4 3.462 3.114 0.40162
## 5 5 3.558 3.462 0.44390
## 6 6 3.589 3.558 0.12652
## 7 7 3.591 3.589 0.03334
## 8 8 3.591 3.591 0.00253

# increased precision
out <- f.secant(f.fp, xnew = 1, xtwo = 4, eps = 1e-12)
out

## $root
## [1] 3.591
##
```

```
## $iter
## [1] 10
##
## $hist
##     i  xnew  xtwo      diff
## 1   1 1.000 4.000 3.000e+00
## 2   2 3.968 1.000 3.236e-02
## 3   3 3.938 3.968 2.938e+00
## 4   4 3.500 3.938 4.678e-01
## 5   5 3.611 3.500 3.265e-01
## 6   6 3.592 3.611 9.238e-02
## 7   7 3.591 3.592 2.001e-02
## 8   8 3.591 3.591 1.090e-03
## 9   9 3.591 3.591 1.324e-05
## 10 10 3.591 3.591 8.737e-09
```

```
out <- f.secant(f.fp, xnew = 1, xtwo = 5.2)
out
```

```
## $root
## [1] 3.591
##
## $iter
## [1] 17
##
## $hist
##     i   xnew   xtwo     diff
## 1   1 1.0000 5.2000 4.200000
## 2   2 5.1026 1.0000 0.097410
## 3   3 5.0092 5.1026 4.009196
## 4   4 0.7144 5.0092 4.388177
## 5   5 4.9572 0.7144 0.051958
## 6   6 4.9066 4.9572 4.192177
## 7   7 1.4048 4.9066 3.552449
## 8   8 4.7503 1.4048 0.156333
## 9   9 4.6083 4.7503 3.203483
## 10 10 2.3875 4.6083 2.362800
## 11 11 4.2278 2.3875 0.380430
## 12 12 3.9781 4.2278 1.590638
## 13 13 3.4070 3.9781 0.820803
## 14 14 3.6355 3.4070 0.342548
## 15 15 3.5959 3.6355 0.188899
## 16 16 3.5910 3.5959 0.044557
## 17 17 3.5911 3.5910 0.004817
```

```
# can not be evaluated (complex numbers)
out <- f.secant(f.fp, xnew = 1, xtwo = 5.5)

## Warning:  NaNs produced
## Warning:  NaNs produced
## Error:  missing value where TRUE/FALSE needed
```