NEWTON AND MAJORIZATION ALGORITHMS FOR THE RECIPROCAL OF A REAL NUMBER

JAN DE LEEUW

ABSTRACT. We consider the function $f_a(x) = ax - \log(x)$ on the positive reals, where $a > 0$ is a constant. In this (short, didactic) paper we study two algorithms for minimizing this function.

1. INTRODUCTION

Minimizing the function $f_a(x) = ax - \log(x)$, where $a > 0$ is a constant, over $x > 0$ is trivial. The first and second derivatives of f_a are

$$f_a'(x) = a - \frac{1}{x}$$

and

$$f_a''(x) = \frac{1}{x^2}$$

We see from $f_a''(x) > 0$ that f_a is strictly convex on the positive reals. It has its unique minimum for $a - \frac{1}{x} = 0$, i.e. for $x = \frac{1}{a}$, and the minimum value is $1 + \log(a)$.

Thus iterative algorithms to minimize the function, which can also be thought of as iterative algorithms to compute the reciprocal of a positive number a, are of little interest in themselves. But it is of some interest to compare two of these algorithms, Newton's method and a majorization method, in terms of robustness, speed of convergence, and so on.

Here are plots of the function f_a for $a = 3/2$ and for $a = 1/3$.

INSERT FIGURE 1 ABOUT HERE

INSERT FIGURE 2 ABOUT HERE

Date: Tuesday 11th February, 2014 — 12h 51min — Typeset in TIMES ROMAN.

2. NEWTON'S METHOD

Newton's method to approximate the minimum is

$$x^+ = x - \frac{f_a'(x)}{f_a''(x)} = x - x^2(a - \frac{1}{x}) = 2x - ax^2. \tag{1}$$

Note that this means we can compute reciprocals of positive numbers without using division at all. In the past the algorithm was actually used by computers to implement division.

The algorithmic map for Newton iterations is $g_a(x) = 2x - ax^2$. Note that this is a logistic map for $a = 2$. We see that $g_a'(x) = 0$ if $x = \frac{1}{a}$, indicating superlinear convergence. In fact

$$x^+ - \frac{1}{a} = -a(x - \frac{1}{a})^2 \tag{2}$$

and thus

$$\frac{x^+ - \frac{1}{a}}{(x - \frac{1}{a})^2} = -a, \tag{3}$$

and we actually have local quadratic convergence (if we have convergence at all). We also see from (2) that $x^+ < \frac{1}{a}$, no matter what x is. Since

$$x^+ - x = x(1 - ax) \tag{4}$$

it follows that $x^+ > x$ if $0 < x < \frac{1}{a}$. Thus the sequence $x^{(k)}$ is increasing and converges quadratically to $\frac{1}{a}$ as long as $0 < x^{(0)} < \frac{1}{a}$.

But observe we have $x^+ < 0$ if $x > \frac{2}{a}$. Thus if $\frac{1}{a} < x^{(0)} < \frac{2}{a}$ we have $x^{(1)}) < x^{(0)}$, but $\frac{1}{a} > x^{(k+1)} > x^{(k)}$ for all $k \geq 1$, and thus there still is convergence. But if $x^{(0)} > \frac{2}{a}$ then the $x^{(k)}$ with $k \geq 1$ are a decreasing sequence of negative numbers, diverging to $-\infty$.

These results can also be derived more directly from the explicit form

$$x^{(k+1)} = \frac{1}{a}\left(1 - \left[a(x^{(0)} - \frac{1}{a})\right]^{2^k}\right). \tag{5}$$

This illustrates that Newton's method works well, if it works at all. But from some starting points it simply does not converge. There is fast local convergence, but no guarantee of global convergence.

2.1. **Examples.** Newton's method is implemented in the two R functions `iterationWriter()`, which writes out the iterations, and `cobwebPlotter()`, which makes a Cobweb plot. The code is in Appendix B.2.

In our three example runs we compute the reciprocal of $a = 1.5$, which is obviously $0.666\cdots$. First we start at 0.1, which is less than $\frac{1}{a}$.

```
iterationWriter(1.5, xold = 0.1, func = upNewton)

##    0.185000000000000
##    0.318662500000000
##    0.485006316640625
##    0.617165942509291
##    0.662991184128510
##    0.666646402908834
##    0.666666666050737
##    0.666666666666667
##    0.666666666666667
```

INSERT FIGURE 3 ABOUT HERE

Next we start at 1.0, which is larger than $\frac{1}{a}$, but smaller than $\frac{2}{a}$.

```
iterationWriter(1.5, xold = 1.0, func = upNewton)

##    0.500000000000000
##    0.625000000000000
##    0.664062500000000
##    0.666656494140625
##    0.666666666511446
##    0.666666666666667
##    0.666666666666667
```

INSERT FIGURE 4 ABOUT HERE

And finally we start at 1.5, which is larger than $\frac{2}{a}$.

```
iterationWriter(1.5, xold = 1.5, itmax = 5, func = upNewton)

##   -0.375000000000000
```

```
##   -0.960937500000000
##   -3.306976318359375
##  -23.018091192003340
## -840.784965569079304
```

INSERT FIGURE 5 ABOUT HERE

3. MAJORIZATION

In majorization [De Leeuw, 1994; Heiser, 1995; Lange et al., 2000] we want to find a function $g(x,y)$ such that $f(x) \leq g(x,y)$ for all positive x and y and $f(x) = g(x,x)$ for all positive x. The algorithm then is

$$x^{(k+1)} = \underset{x}{\textbf{argmin}}\ g(x,x^{(k)}).$$

If $x^{(k+1)} = x^{(k)}$ we stop, otherwise we go to the next iteration. Since

$$f(x^{(k+1)}) < g(x^{(k+1)},x^{(k)}) \leq g(x^{(k)},x^{(k)}) = f(x^{(k)})$$

this generates a decreasing sequence of function values. If the function is bounded below this sequence converges. Under continuity and identification conditions we can also show the accumulation points of $x^{(k)}$ are fixed points of f, or even that the sequence $x^{(k)}$ converges to a fixed point of f.

Finding an appropriate majorization function is somewhat of an art, where the basic tools are convexity, inequalities, and the Taylor series. Typically, majorization is used in complicated multivariate problems, where the majorization function g is hopefully much simpler to minimize than the original function f. We merely use our simple $f(x) = ax - \log(x)$ as in illustrative example to show the global convergence of majorization methods, and the price we have to pay to guarantee global convergence.

4. PIECEWISE MAJORIZATION

Let us first derive a new majorization function for the logarithm. In other contexts the logarithm, which is concave, has been majorized by a linear

function. Since our f uses the negative logarithm, which is convex, that will not work in our case. By Taylor

$$\log(x) = \log(y) + \frac{1}{y}(x-y) - \frac{1}{2}\frac{1}{z^2}(x-y)^2,$$

where z is between x and y. Thus if we define

$$h(x,y) = \log(y) + \frac{1}{y}(x-y) - \frac{1}{2}\begin{cases} \frac{(x-y)^2}{x^2} & \text{if } x \le y, \\ \frac{(x-y)^2}{y^2} & \text{otherwise.} \end{cases}$$

then, for all $x > 0$ and $y > 0$,

$$\log(x) \ge h(x,y),$$

and thus, with $g(x,y) = ax - h(x,y)$, we have $f(x) \le g(x,y)$. with equality if and only if $x = y$.

4.1. **Analysis.** Now g, as a function of x, is differentiable on the positive reals for all y. In fact

$$\mathscr{D}_1 g(x,y) = a - \frac{1}{y} + \begin{cases} \frac{y}{x^3}(x-y) & \text{if } x \le y, \\ \frac{1}{y^2}(x-y) & \text{otherwise.} \end{cases}$$

Let us find the solutions of $\mathscr{D}_1 g(x,y) = 0$. First check if

$$a - \frac{1}{y} + \frac{1}{y^2}(x-y) = 0$$

has a root $x \ge y$. The unique root is $x = 2y - ay^2$. Thus if $2y - ay^2 \ge y$, i.e. if $y \le \frac{1}{a}$, this gives a solution to $\mathscr{D}_1 g(x,y) = 0$. Note that the update in this case is the same update as Newton's update for the reciprocal.

Matters are a bit more complicated for finding a solution of

$$a - \frac{1}{y} + \frac{y}{x^3}(x-y) = 0.$$

with $x \le y$. The equation can be written as the cubic equation in x

$$(a - \frac{1}{y})x^3 + yx - y^2 = 0.$$

If $y > \frac{1}{a}$ then the cubic has only one real root. Because if there are two, then by Rolle the derivative should vanish somewhere on the interval between

them, but the derivative is always positive. Since the cubic is negative for $x=0$ and positive for $x=y$, the unique root is between zero and y, and thus satisfies $\mathscr{D}_1 g(x,y)=0$.

4.2. **Generalization.** If we include additional terms in the Taylor expansion of the logarithm we get

$$\log(x)=\log(y)+\frac{1}{y}(x-y)+\cdots+(-1)^{p-1}\frac{1}{p!}\frac{1}{z^p}(x-y)^p.$$

Because z^p is monotone on the non-negative reals we have

$$\min(x,y)^p\leq z^p\leq\max(x,y)^p.$$

If p is even

$$-\frac{1}{p!}\frac{1}{z^p}(x-y)^p\leq\begin{cases}-\frac{1}{p!}\frac{1}{x^p}(x-y)^p & \text{if } x\leq y,\\ -\frac{1}{p!}\frac{1}{y^p}(x-y)^p & \text{otherwise.}\end{cases}$$

Minimizing the majorization function in this case leads to a convergent algorithm with rate p. In each iteration we have to find the minimum of a polynomial of degree p over $x\geq y$ and a polynomial of degree $p+1$ over $x\leq y$.

For odd p we could use the more complicated form

$$\frac{1}{p!}\frac{1}{z^p}(x-y)^p\leq\frac{1}{p!}\frac{1}{z^p}|x-y|^p\leq\begin{cases}\frac{1}{p!}\frac{1}{y^p}|x-y|^p & \text{if } x\leq y,\\ \frac{1}{p!}\frac{1}{x^p}|x-y|^p & \text{otherwise.}\end{cases}$$

This is obviously less convenient to work with.

4.3. **Examples.**

```
iterationWriter (1.5, 0.1, func = upMajor)
```

```
##     0.185000000000000
##     0.318662500000000
##     0.485006316640625
##     0.617165942509291
##     0.662991184128510
##     0.666646402908834
```

```
##     0.666666666050737
##     0.666666666666667
##     0.666666666666666

iterationWriter (1.5, 1.0, func = upMajor)

##     0.770916997059247
##     0.685981295382923
##     0.667657545315318
##     0.666669592626095
##     0.666666666692350
##     0.666666666666667

iterationWriter (1.5, 1.5, func = upMajor)

##     0.978890109310998
##     0.762346298831599
##     0.683494549754625
##     0.667430025994483
##     0.666668405860389
##     0.666666666675741
##     0.666666666666667
```

INSERT FIGURE 7 ABOUT HERE

INSERT FIGURE 8 ABOUT HERE

INSERT FIGURE 9 ABOUT HERE

The majorization iterations in this section are either Newton iterations or iterations defined by the solution of the cubic equation. They have the convenient property that if $0 < x^{(0)} < \frac{1}{a}$ then they will be Newton all the way, and if $x^{(0)} > \frac{1}{a}$ they will be "cubic" all the way.

We know Newton iterations converge quadratically. To determine the local convergence rate of the cubic iterations we use the implicit function theorem. The update function h is defined implicitly as

$$(a-\frac{1}{x})h(x)^3+xh(x)-x^2=0.$$

Thus the derivative of the update function is

$$\mathscr{D}h(x)=-\frac{\frac{x^3}{y^2}+x-2y}{3(a-\frac{1}{y})x^2+y}$$

which vanishes for $x=y=\frac{1}{a}$. Thus the cubic iterates also have superlinear convergence.

5. ANOTHER MAJORIZATION

We discussed earlier that we cannot use the standard majorization of the logarithm by a linear function, because our objective function has the negative logarithm. But, of course, $-\log(x)=\log(\frac{1}{x})$, and we can use the concavity of the logarithm to show that

$$\log(\frac{1}{x})\leq\log(\frac{1}{y})+y\left(\frac{1}{x}-\frac{1}{y}\right),$$

or

$$-\log(x)\leq-\log(y)+\frac{y}{x}-1.$$

Thus

$$g(x,y)=ax-\log(y)+\frac{y}{x}-1$$

majorizes f, and minimizing the majorizer gives the very simple algorithm

$$x^{(k+1)}=\sqrt{\frac{x^{(k)}}{a}}.$$

The derivative of the update function $h(x)=\sqrt{\frac{x}{a}}$ at $\frac{1}{a}$ is $\frac{1}{2}$. Thus our majorization iterations have linear convergence, with ratio $\frac{1}{2}$.

If $x^{(0)}<\frac{1}{a}$ the algorithm generates an increasing sequence converging to $\frac{1}{a}$. If $x^{(0)}>\frac{1}{a}$ we have a decreasing sequence converging to $\frac{1}{a}$.

Because $ax^{(k+1)} = \sqrt{ax^{(k)}}$ we have the explicit expression

$$x^{(k)} = a^{\left(\frac{1}{2^k}-1\right)} \left(x^{(0)}\right)^{\left(\frac{1}{2^k}\right)}.$$

```
iterationWriter (1.5, 0.1, func = upOther)

##    0.258198889747161
##    0.414888651525652
##    0.525920558972964
##    0.592126427363258
##    0.628292090970040
##    0.647195020051937
##    0.656858696144986
##    0.661744510691242
##    0.664201029152691
##    0.665432705915327
##    0.666049400527882
##    0.666357962123403
##    0.666512296522429
##    0.666589477125879
##    0.666628070779041
##    0.666647368443538
##    0.666657017485272
##    0.666661842058512
##    0.666664254358225
##    0.666665460511355
##    0.666666063588738
##    0.666666365127634
##    0.666666515897133
##    0.666666591281896
##    0.666666628974280

iterationWriter (1.5, 1.0, func = upOther)

##    0.816496580927726
##    0.737787946466881
```

```
##     0.701326337005763
##     0.683776930977622
##     0.675167599428775
##     0.670903668906734
##     0.668781812405578
##     0.667723402018919
##     0.667194825129271
##     0.666930693615297
##     0.666798667072901
##     0.666732663603087
##     0.666699664318243
##     0.666683165288302
##     0.666674915926446
##     0.666670791283797
##     0.666668728972042
##     0.666667697818557
##     0.666667182242412
##     0.666666924454490
##     0.666666795560566
##     0.666666731113613
##     0.666666698890139
##     0.666666682778403
##     0.666666674722535

iterationWriter (1.5, 1.5, func = upOther)

##     1.000000000000000
##     0.816496580927726
##     0.737787946466881
##     0.701326337005763
##     0.683776930977622
##     0.675167599428775
##     0.670903668906734
##     0.668781812405578
##     0.667723402018919
##     0.667194825129271
```

```
##    0.666930693615297
##    0.666798667072901
##    0.666732663603087
##    0.666699664318243
##    0.666683165288302
##    0.666674915926446
##    0.666670791283797
##    0.666668728972042
##    0.666667697818557
##    0.666667182242412
##    0.666666924454490
##    0.666666795560566
##    0.666666731113613
##    0.666666698890139
##    0.666666682778403
```

6. CONCLUSION

This paper illustrates a number of important points, which we summarize here. Newton's method, in regular cases such as these, has fast quadratic convergence if started close enough to a stationary point. There are, however, starting points for which is does not converge at all. Majorization of the simple form (discussed in the section "Other Majorization") is globally convergent from any starting point, but generally linearly convergent with a rate that may be uncomfortably close to one. It is possible, however, to find majorization methods that converge at a superlinear rate and are still globally convergent (discussed in the section "Piecewise Majorization"). They tend to require more computation in each iteration, in our example finding the root of a cubic polynomial. For other examples of superlinearly convergent majorization methods, see De Leeuw [2006].

APPENDIX A. FIGURES

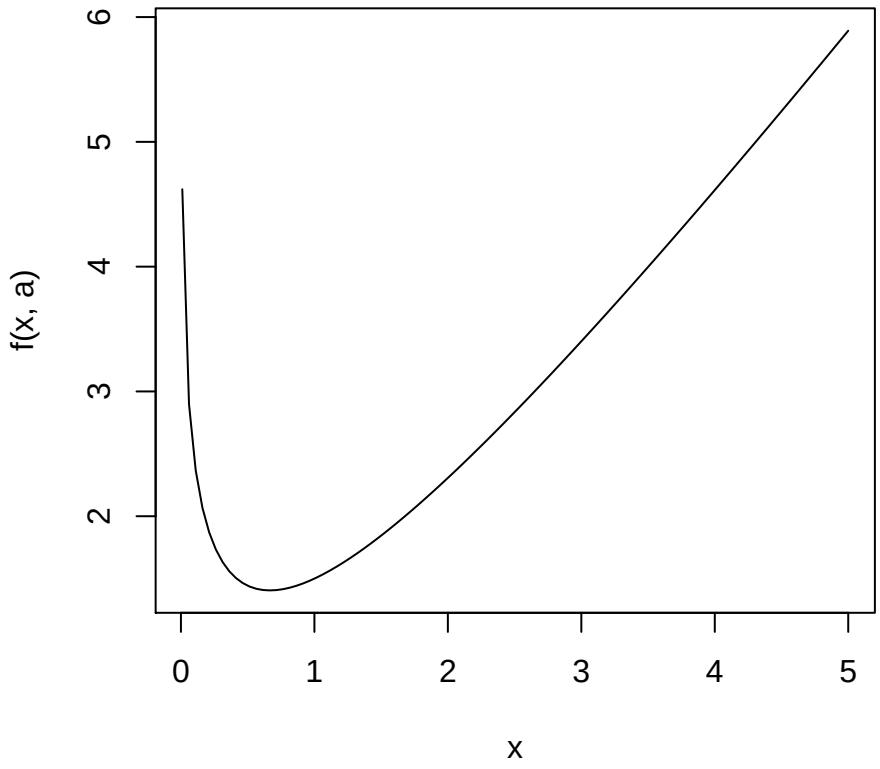

FIGURE 1. Function f for a=3/2

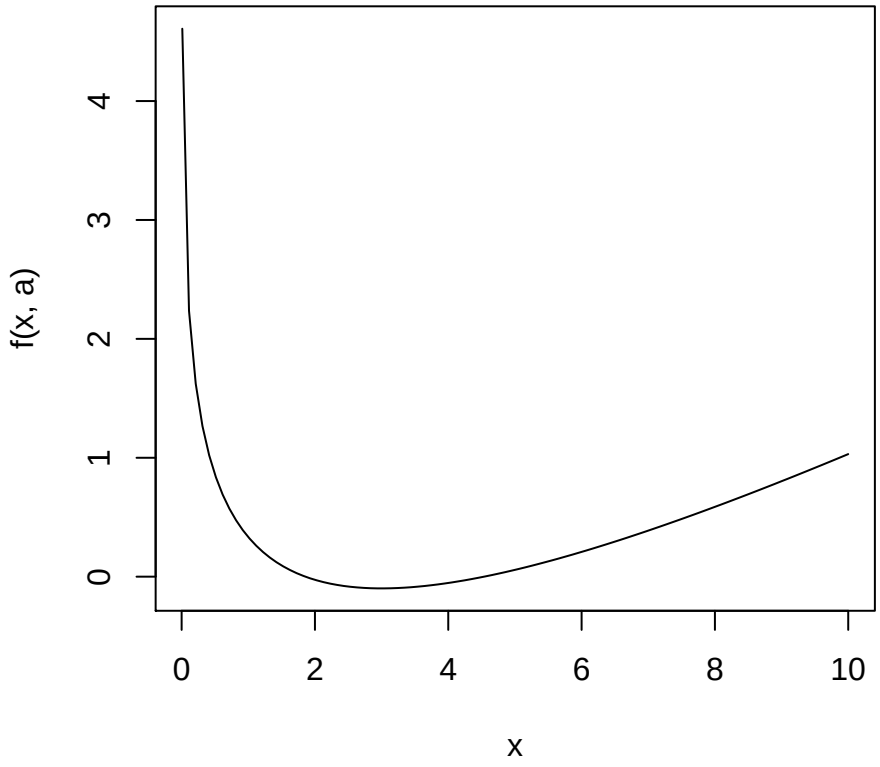

FIGURE 2. Function f for a=1/3

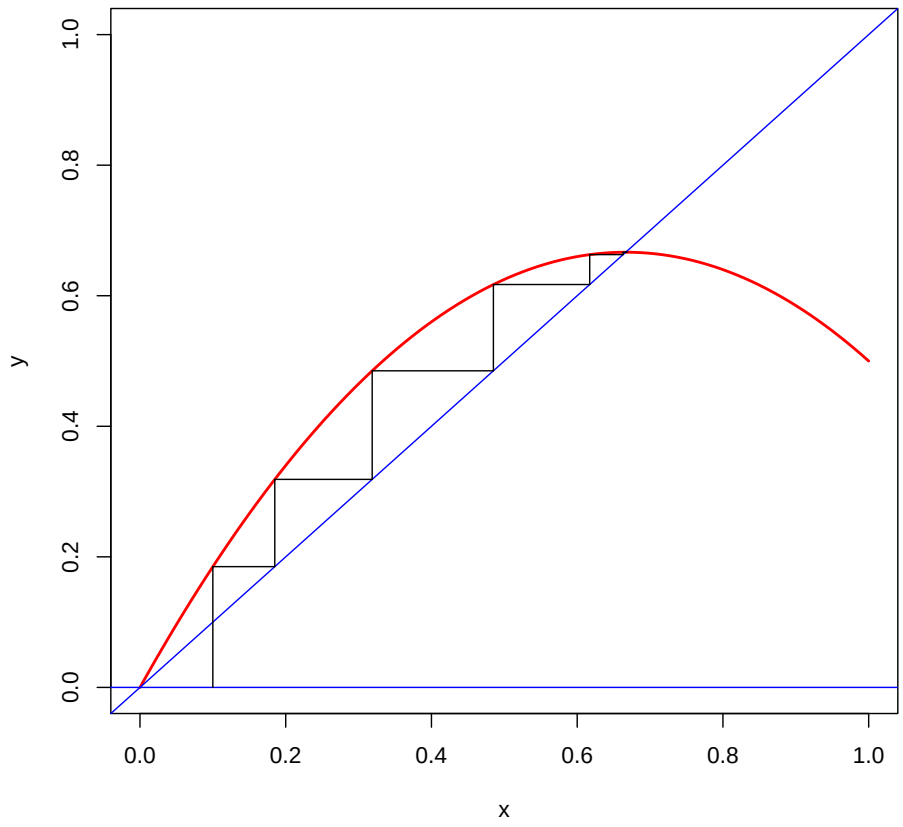

FIGURE 3. Newton Iterations for a=1.5 starting at 0.1

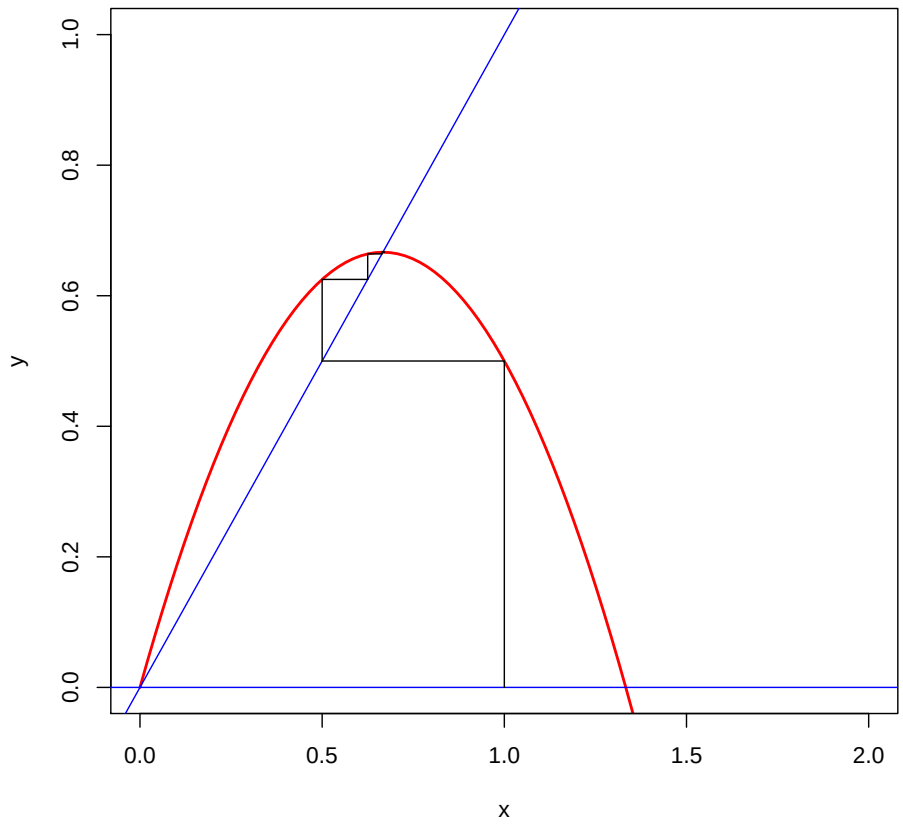

FIGURE 4. Newton Iterations for a=1.5 starting at 1.0

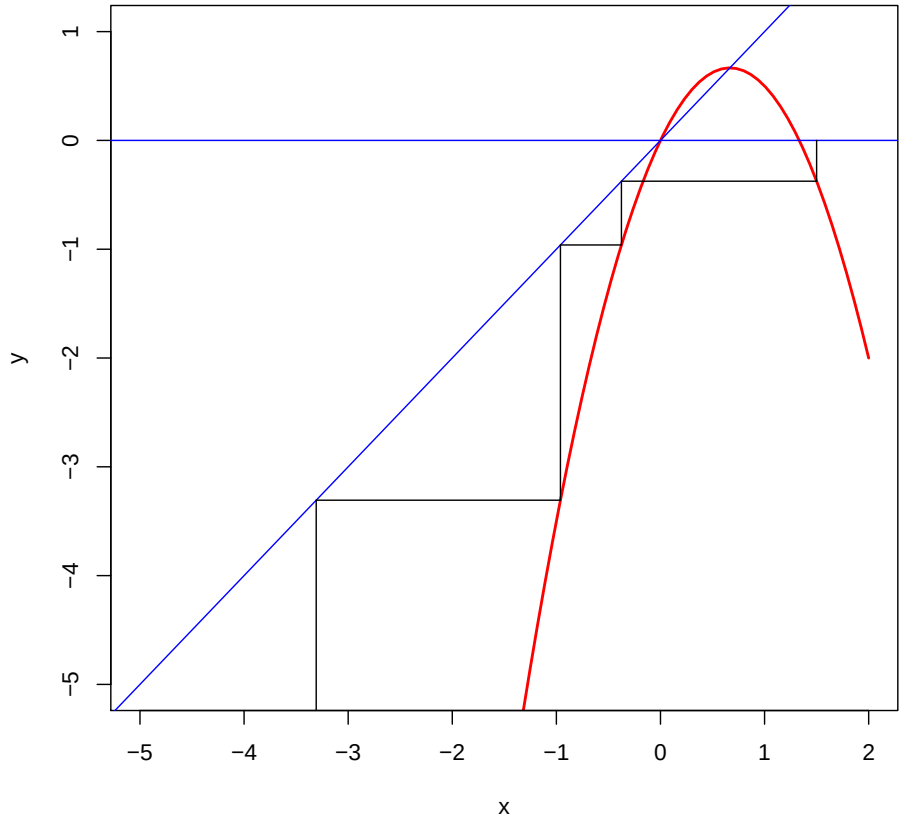

FIGURE 5. Newton Iterations for a=1.5 starting at 1.5

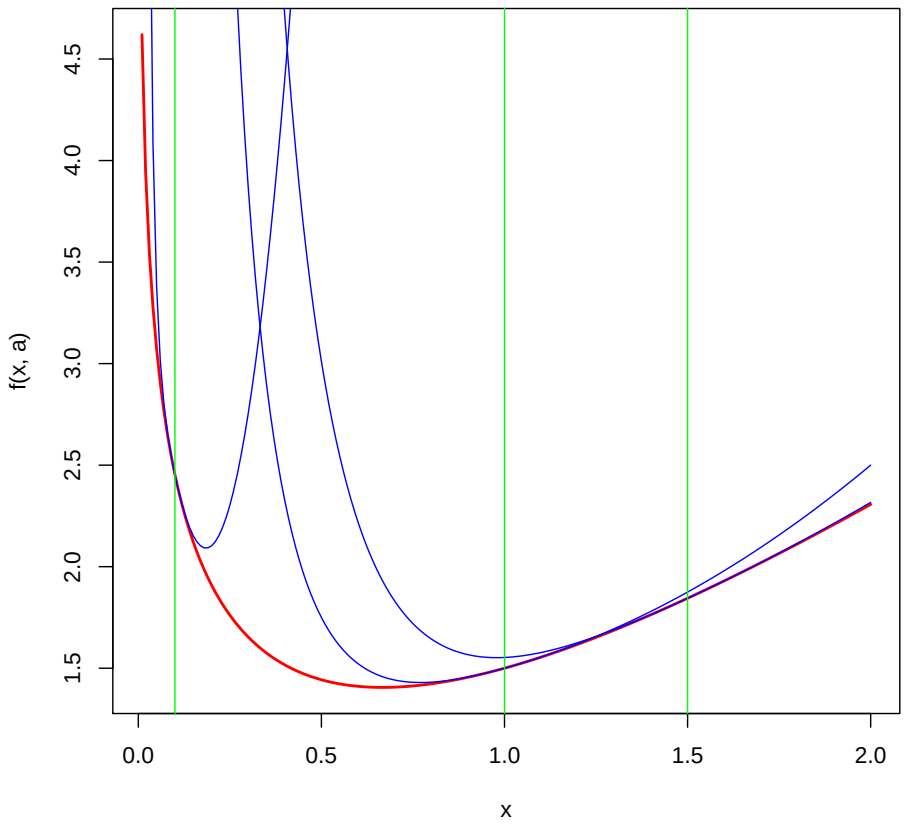

FIGURE 6. Piecewise Majorization for a=1.5 and y is 0.1, 1.0, and 1.5

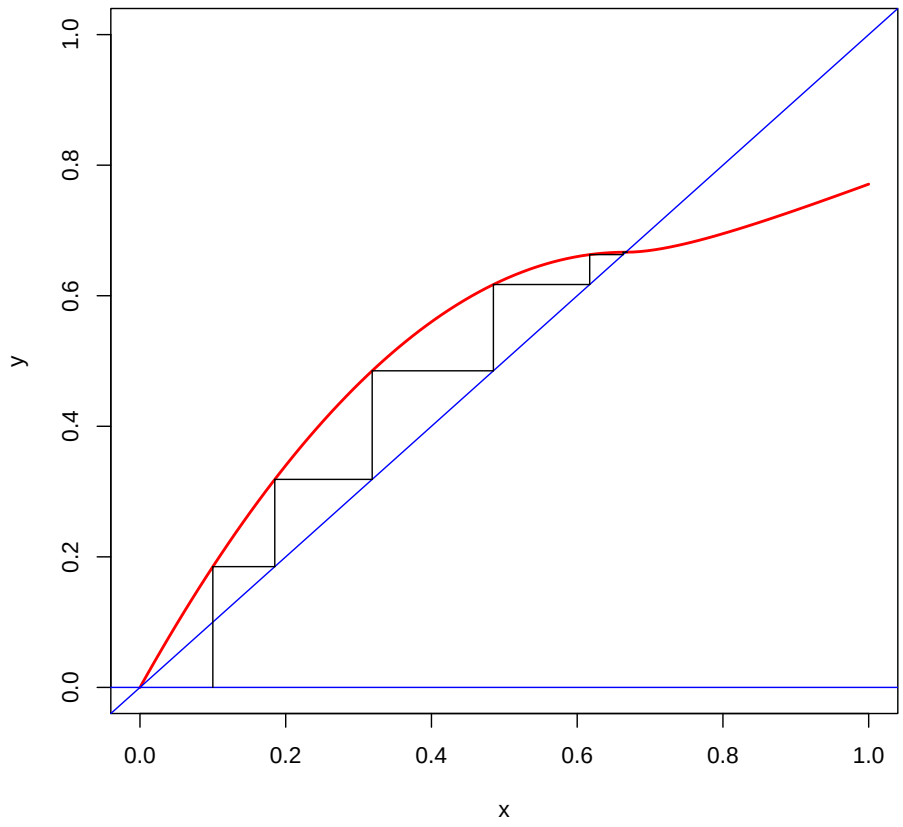

FIGURE 7. Majorization Iterations for a=1.5 starting at 0.1

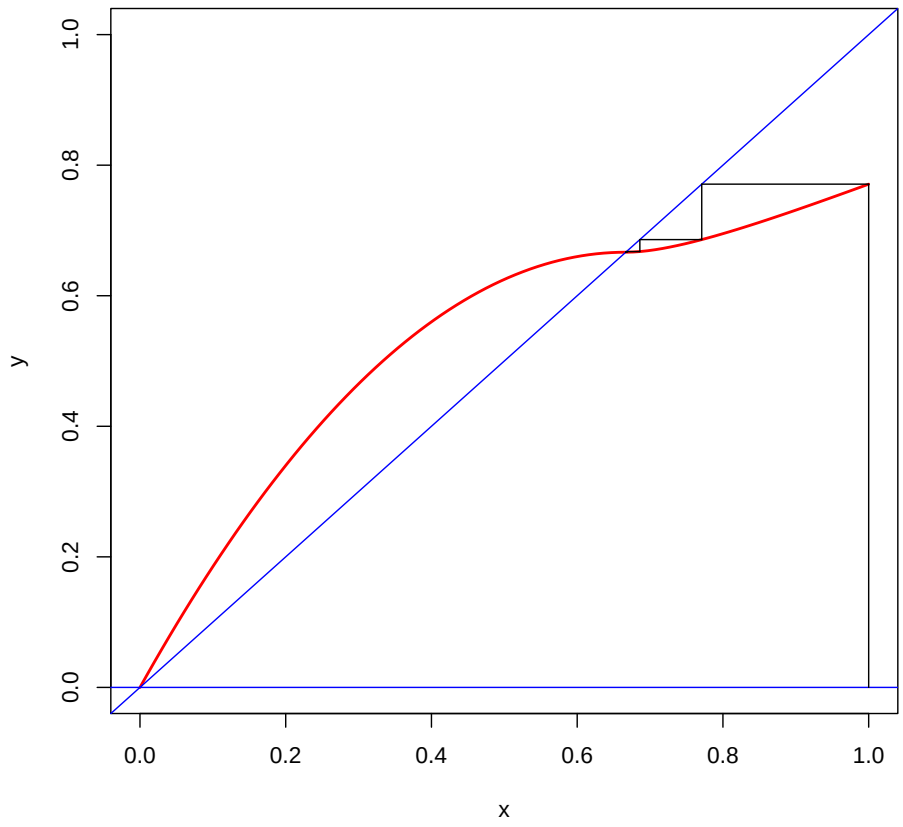

FIGURE 8. Majorization Iterations for a=1.5 starting at 1.0

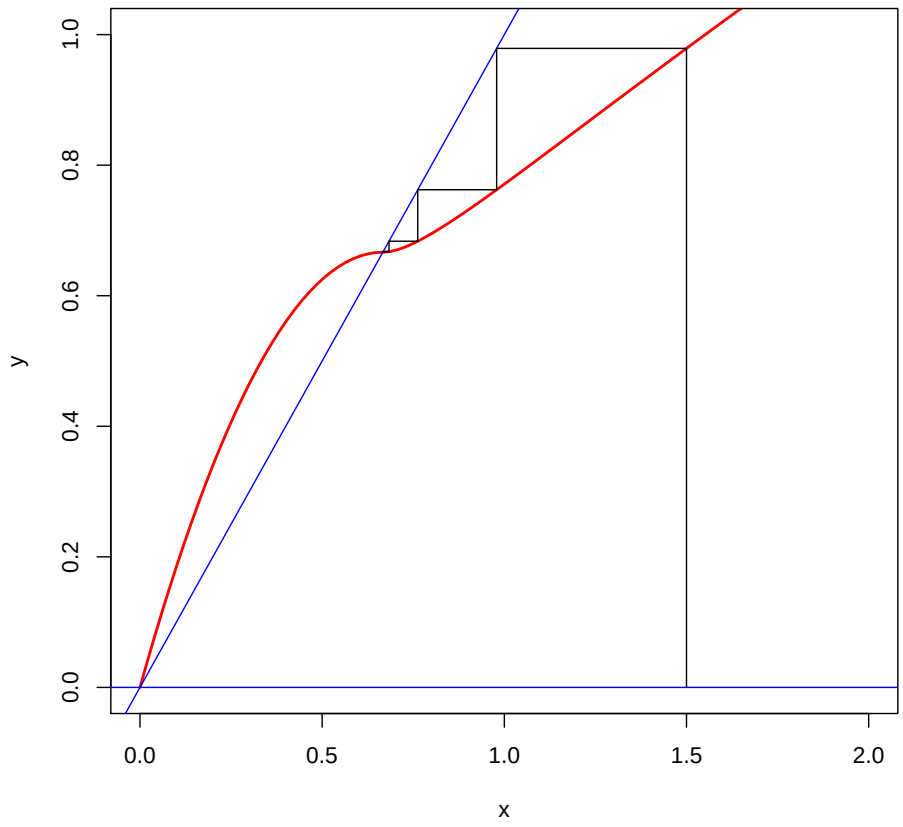

FIGURE 9. Majorization Iterations for a=1.5 starting at 1.5

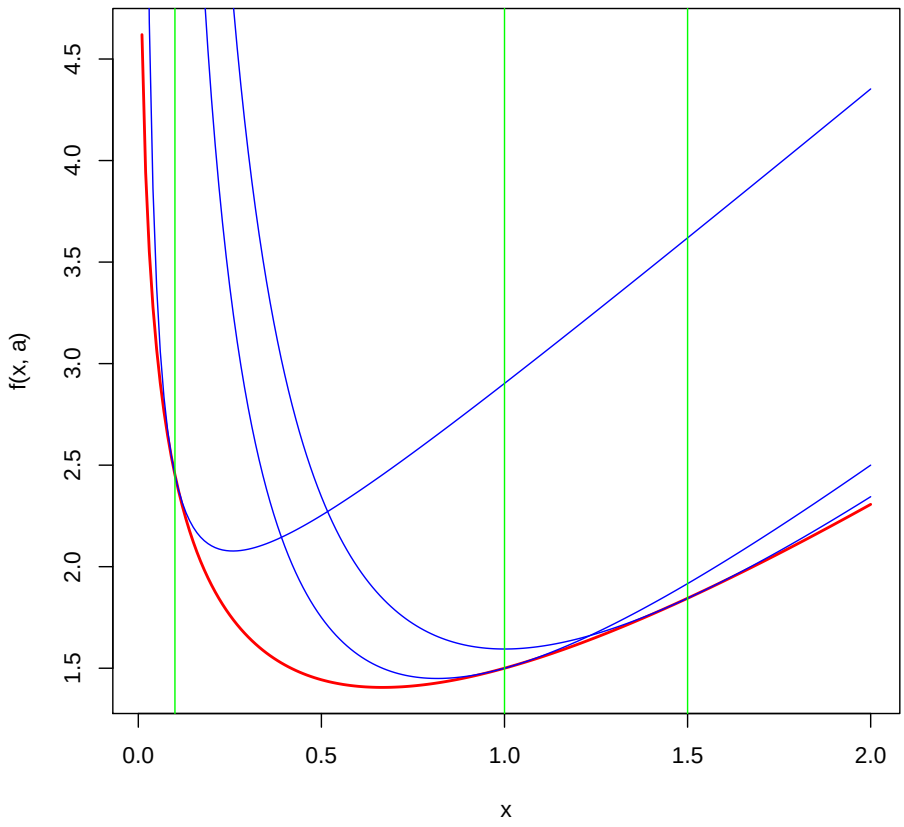

FIGURE 10. Other Majorization for a=1.5 and y is 0.1, 1.0, and 1.5

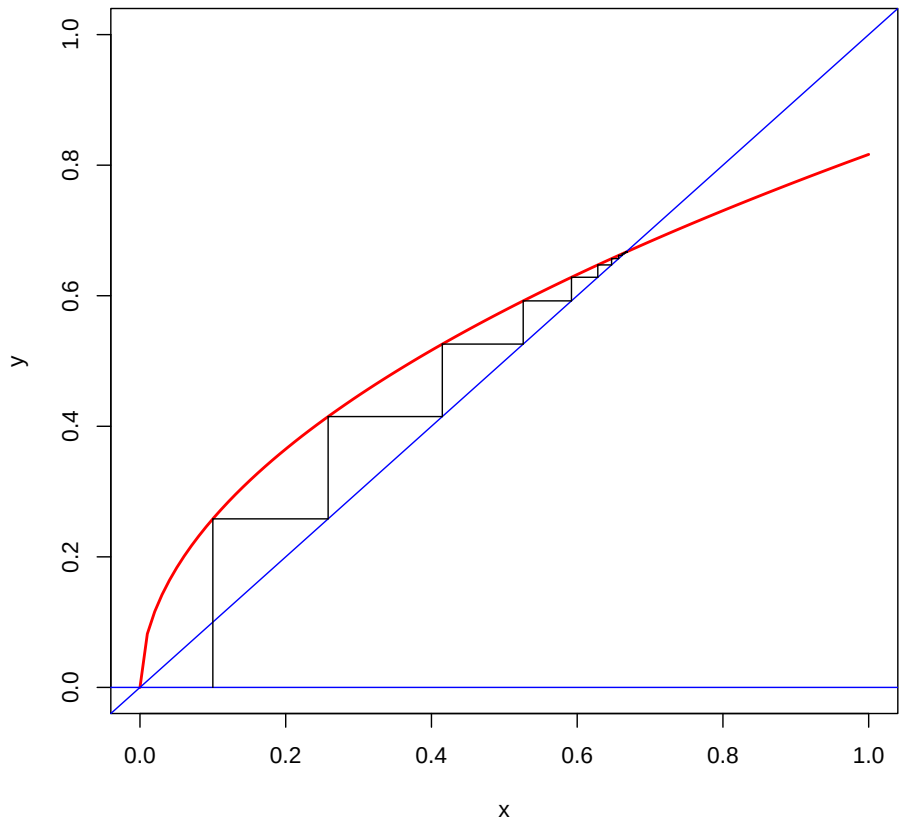

FIGURE 11. Other Majorization Iterations for a=1.5 starting at 0.1

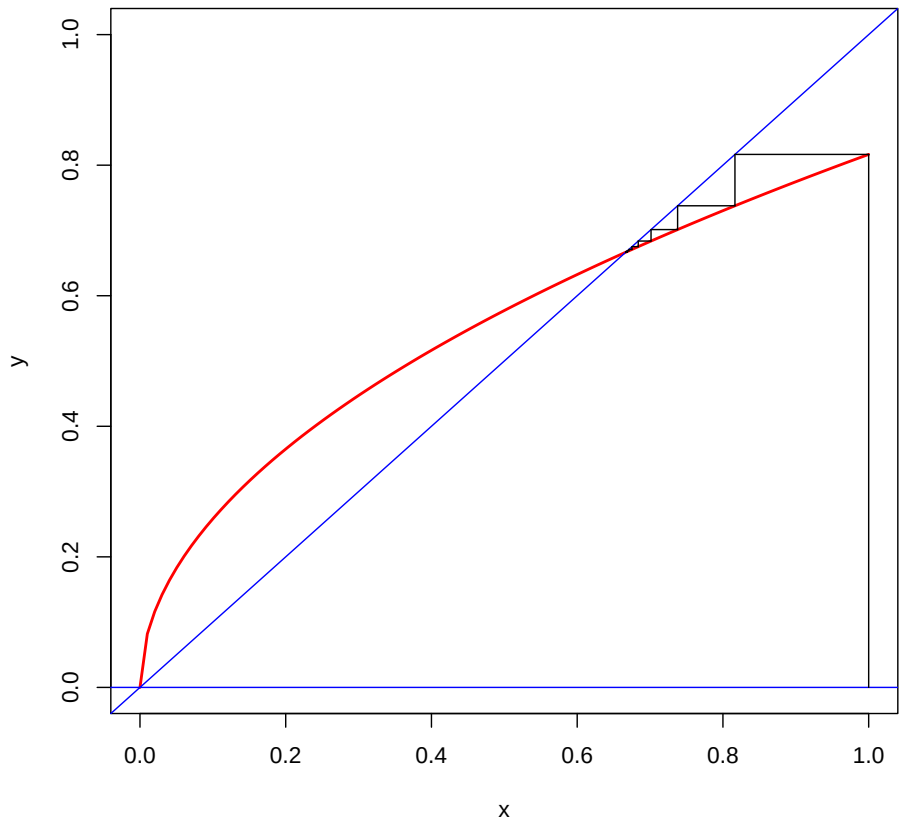

FIGURE 12. Other Majorization Iterations for a=1.5 starting at 1.0

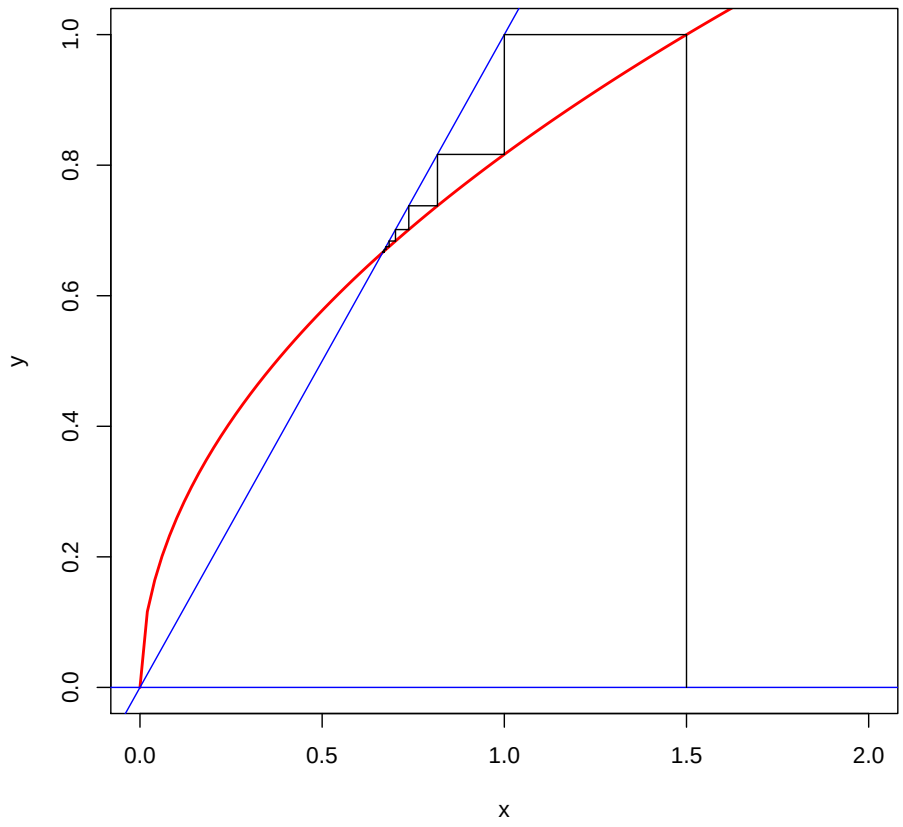

FIGURE 13. Other Majorization Iterations for a=1.5 starting at 1.5

APPENDIX B. CODE

B.1. **Code for Section 1.**

```
f <- function (x, a) {
  return (a * x - log (x))
}
```

```
x <- seq (.01, 5, length = 100)
a <- 3/2
plot(x, f(x, a), type = "l")
```

```
x <- seq (.01, 10, length = 100)
a <- 1/3
plot(x, f(x, a), type = "l")
```

B.2. Code for Section 2.

```
upNewton <- function (x, a) {
  return (2 * x - a * x ^ 2)
}
```

```
cobwebPlotter <- function (a, xold, func, low = 0, hgh = 1,
                     eps = 1e-10, itmax = 25) {
    x <- seq (low, hgh, length = 100)
    y <- sapply (x, function (x) func (x, a))
    plot (x, y, xlim = c(low ,hgh), ylim = c(low,1),
          type = "l", col = "RED", lwd = 2)
    abline (0, 1, col = "BLUE")
    abline (0, 0, col = "BLUE")
    base <- 0
    itel <- 1
    repeat {
        xnew <- func (xold, a)
        lines (matrix(c(xold,xold,base,xnew),2,2))
        lines (matrix(c(xold,xnew,xnew,xnew),2,2))
        if ((abs (xnew - xold) < eps) || (itel == itmax)) {
            break ()
        }
        base <- xnew
        xold <- xnew
        itel <- itel + 1
    }
}
```

```
iterationWriter <- function (a, xold, func, eps = 1e-10, itmax = 25) {
    itel <- 1
    repeat {
        xnew <- func (xold, a)
        cat (formatC (xnew, digits = 15,
                width = 20, format = "f"), "\n")
        if ((abs (xnew - xold) < eps) || (itel == itmax)) {
          break ()
        }
        xold <- xnew
        itel <- itel + 1
     }
}
```

B.3. Code for Section 3.

```
upMajor <- function (x, a) {
    d <- x - (1 / a)
    if (d < 0) {
      xnew <- 2 * x - a * x ^2
    } else {
      xnew <- psolver (x, a)
    }
    return (xnew)
}

psolver <- function (y, a) {
  e <- a - (1 / y)
  h <- matrix (c(0,1,0,0,0,1,y^2/e,-y/e,0), 3, 3)
  v <- eigen(h)$values
  return (Re(v[which (Im (v) == 0)]))
}
```

B.4. **Code for Section 5.**

```
upOther <- function (x, a) {
  return (sqrt (x / a))
}
```

REFERENCES

J. De Leeuw. Block Relaxation Algorithms in Statistics. In H.H. Bock, W. Lenski, and M.M. Richter, editors, *Information Systems and Data Analysis*, pages 308–324, Berlin, 1994. Springer Verlag.

J. De Leeuw. Quadratic and cubic majorization. *Medium for Econometric Applications*, 14:44–49, 2006.

W.J. Heiser. Convergent Computing by Iterative Majorization: Theory and Applications in Multidimensional Data Analysis. In W.J. Krzanowski, editor, *Recent Advantages in Descriptive Multivariate Analysis*, pages 157–189. Clarendon Press, Oxford, 1995.

K. Lange, D.R. Hunter, and I. Yang. Optimization Transfer Using Surrogate Objective Functions. *Journal of Computational and Graphical Statistics*, 9:1–20, 2000.

DEPARTMENT OF STATISTICS, UNIVERSITY OF CALIFORNIA, LOS ANGELES, CA 90095-1554

E-mail address, Jan de Leeuw: deleeuw@stat.ucla.edu

URL, Jan de Leeuw: http://gifi.stat.ucla.edu