Plotting Expensive or Fast Varying Smooth Functions

Martin B. Maechler
Bell Communications Research, 2M-343
445 South Street
Morristown, NJ 07962-1910, USA.
e-mail: maechler@bellcore.com

July 15, 1991

Abstract

This paper describes a new approach for automatically determining a proper set of abscissa values in order to plot smooth functions that are expensive or that vary fast over part of the interval of interest. Given a smooth (e.g., piecewise differentiable) function $f : R \to \mathbb{R}$ (for $R \subset \mathbb{R}$), and an interval $[a, b] \subset R$, we are looking for a minimal set of values $a = x_1 \leq x_2 \leq \cdots \leq x_n = b$, such that the polyline given by (its corners) $\{(x_i, f(x_i))\}$, $i = 1, \ldots, n$, "looks like" an infinite precision plot of the function f.

In other words, the corners of the polyline should (almost) be "invisible" (whenever the derivative of f is continuous). We implemented a greedy algorithm that starts from a small set of abscissa values and adds more points wherever necessary. New points are added in an interval if (at least) one of the adjacent corners is jagged and if the line segment is long enough. Our main criterion for "visible jaggedness" of a corner is its angle.

Not only is our algorithm efficient when the number of evaluations of an expensive function has to be kept as low as possible, but it is also very useful for automatically representing and drawing functions with infinite poles or other singularities.

1 Motivation: Abscissa Choice for Function Plotting

A typical low-level task arising in data analysis and in mathematical and statistical experiments is plotting a function. Very often, and in our context, we want to draw the graph of a smooth function $f : R \to \mathbb{R}$ (for $R \subset \mathbb{R}$). We will not rigorously define "smooth", but we have in mind a piecewise continuously differentiable function with (at most) a few discontinuities in the function or its first derivative.

Given an interval $[a, b] \subset R$, it is customary just to evaluate f at an equispaced grid of n (several dozen to a few hundred) points $x_i = a + \frac{i-1}{n-1}(b - a)$, for $i = 1, 2, \ldots, n$. This is okay as long as evaluation of f is cheap and f is not varying too rapidly anywhere. It may be necessary to determine the set of abscissa values x_i more diligently in the following situations:

1. Evaluation of $f(\cdot)$ is expensive, e.g., $f(x)$ is the result of an optimization or a complicated simulation with parameter x.

2. The function f varies rapidly around some points. This behavior may not be revealed if the chosen grid is too course. A typical example is a function with a pole, i.e., x_p with $\lim_{x \to x_p} 1/f(x) = 0$.

In the above situations, we are looking for a minimal set of values $a = x_1 \le x_2 \le \cdots \le x_n = b$, such that the polyline given by (its corners) $\{(x_i, f(x_i))\}$, $i = 1, \ldots, n$, "looks like" an infinite precision plot of the function f. In other words, the corners of the polyline should (almost) be "invisible" whenever f', the derivative of f is continuous. This is similar to saying that f should be uniformly well approximated by the "linear spline" determined by $(x_1, f(x_1)), \ldots, (x_n, f(x_n))$. Stated like this, we have a well known problem in spline approximation theory with variable knots. Finding an optimal solution is a difficult nonlinear problem that would not be reasonable to address for the purpose of just plotting our function f.

Instead, we implemented a greedy algorithm that starts from a small set of abscissa values and adds more points wherever necessary:
The interval $[x_i, x_{i+1}]$ is subdivided, i.e., an interior abscissa value is added if: Either of its corresponding corner points $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$ is jagged and the corresponding line segment is long enough. Our main criterion for "visible jaggedness" is the angle at the corner of the (current) polyline.

In order to assess length and angle, we assume the following: The original x–y coordinate system is not the one in which the plot will be drawn; rather, the plotting coordinate system will be such that the plot region (determined by the ranges of x_i and $y_i = f(x_i)$) is approximately square.

The rest of the paper is organized as follows: In section 2, we discuss the basic algorithmic components. Section 3 gives more specific remarks regarding our implementation, and section 4 discusses possible extensions and improvements for the future. Appendix A contains the S `help()` information, whereas appendix B gives both the full S code for the main S function `plot.function` and the test procedure used to produce the featureful set of examples given in appendix C.

2 Our Algorithm

Starting with a (small) set of initial abscissa values x_i, we iteratively add points, until no long line segments with jagged corners remain or until the number of function evaluations of $f(\cdot)$ hits a preset maximum.

We have implemented our algorithm in S, which encourages vectorization (to some extent). We therefore treat all the line segments and corners simultaneously, and in each iteration add a new set of abscissas "in parallel". (Note that this indicates possibilities for even more efficient implementations on parallel hardware architectures).

Here we describe the main algorithmic components; appendix B.1 contains the (source code of) the detailed algorithm including error checks, treatment of extreme cases and statements producing diagnostic output.

2.1 Coordinate system; rescaling of y

The original x–y coordinate system is not the one in which the plot will be drawn. Rather, for given points x_i, $y_i = f(x_i)$, the plot will rescale the axes such that the plot region will be approximately square. For our consideration of long line segments and jagged corners, we will use such a coordinate system by rescaling the y_i before computing the geometric attributes:

$$y_i := y_i(b-a)/(\max_j y_j - \min_j y_j).$$

2.2 Angles at corners

An elementary geometrical sub-problem is the determination of the (absolute) angle between two directed line segments, i.e., vectors. This angle is function of the two slopes of the lines (and their order). Assume that we are given the slopes m_1 and m_2 of the line segments l_1 and l_2 and we want to determine the (absolute value of the) angle that one turns "going" along l_1 to l_2. There are two good solutions to this problem.

1. The first (probably more straightforward) one makes use of the geometric property of the scalar product:
$$\boldsymbol{u}\cdot\boldsymbol{v} = \|\boldsymbol{u}\|\,\|\boldsymbol{v}\|\cos(\alpha_{\boldsymbol{u},\boldsymbol{v}}),$$
where $\alpha_{\boldsymbol{u},\boldsymbol{v}}$ is the angle between $\boldsymbol{u}$ and $\boldsymbol{v}$, in the interval $[0,\pi)$. In our case, with $\boldsymbol{u} = (1, m_1)$ and $\boldsymbol{v} = (1, m_2)$, we get
$$\text{angle}\ (m_1, m_2) = \arccos\frac{1+m_1 m_2}{\sqrt{(1+m_1^2)(1+m_2^2)}}. \tag{1}$$

2. The second method proved to be about 25% faster to compute. Here, we use the elementary formula for the tangent of the angle (of rotation of l_1 into l_2):
$$\tan(\alpha_{m_1,m_2}) = \frac{m_2 - m_1}{1 + m_1 m_2}$$
Note that here $\alpha_{m_1,m_2} \in (-\pi, \pi)$, i.e., the angle has a sign that we will disregard.

   Computationally, we use the S function `atan(x,y)`, cf. Becker et al. (1988), or equivalently the C function `atan2(y, x)`, cf. Kernighan and Ritchie (1988). Both return the argument or phase (angle) θ of a polar representation (r, θ) of the cartesian point (x, y).

Corners are considered jagged iff their absolute angles are bigger than a user specified parameter `angle.max` which by default is set to 3 degrees, i.e, $\pi/60$ radians. [Using degrees instead of radians is easier and more intuitive for most users — even for the mathematically educated ones.]

2.3 Long segments

As mentioned before, acute corners are only visually problematic if they are delimited by (two or one) long line segments. To assess this, we will use a translation and rotation invariant distance (norm) in the rescaled coordinate system specified in section 2.1.

The line segment with endpoints (x_i, y_i) and (x_{i+1}, y_{i+1}) is considered long iff

$$\left\|\begin{pmatrix}x_{i+1}\\ y_{i+1}\end{pmatrix} - \begin{pmatrix}x_i\\ y_i\end{pmatrix}\right\| > (b-a)\,l_{\min}, \qquad \text{where}\ \ l_{\min} = \texttt{length.min}, \tag{2}$$

and `length.min` is a parameter in our S implementation. Currently, the default value for `length.min` is 1/64 which means a segment is considered long if it is longer than 1/64 of the length (or height) of our (rescaled) plot region.

Currently, we use the L_1 norm $\left\| \begin{pmatrix} u \\ v \end{pmatrix} \right\| = |u| + |v|$. This is more robust to extreme values and easier to compute than the Euclidean distance, or L_2 norm, ($\left\|(u,v)^{\mathsf{T}}\right\| = \sqrt{u^2+v^2}$).

2.4 Constructing new points

We prefer to consider the problem for **long** line segments which have at least one **jagged** corner. The alternative approach, namely to consider the problem for each **jagged** corner which borders a **long** segment is somewhat more complicated.

3 Remarks on our Implementation in S

The source code for our S function `plot.function` is in appendix B.1 and a short user description (as with `help()`) is in appendix A.

An implementation in S is necessarily computationally inefficient because S code is interpreted and the algorithm is iterative.

There are, however, other advantages to choosing S:

- It is relatively easy to experiment with different variations of the algorithm.
- It is straightforward to experiment with many different functions since they can be passed as arguments to `plot.function`.
- The built-in graphical capabilities of S enable one to inspect the outcome of experiments graphically.

A considerable fraction of the source code is dedicated to checks (and remedies) for possible anomalies of all different kinds:

- Inappropriately specified user parameters.
- Overflow of function values $f(x_i)$, including *`NA`*'s returned at locations x where $f(x)$ is undefined.
- Intervals $[x_i, x_{i+1})$ which are too short (leading to unstable computation of the slopes).
- New x_i values duplicating old ones because of other extreme situations (as $f(x) \to \infty$).

Another part of the code is dedicated to diagnostic output which is of primary interest only to a person investigating the workings of our algorithm. We provide a parameter `verbose.level` that indicates the amount of diagnostic output that is desired. For `verbose.level`≤ 1, only 0 to 2 lines of diagnostic output is produced whereas for `verbose.level`≥ 2, at least one line of status information is given per iteration.

4 Possible Extensions and Enhancements

The following ideas might enhance and extend the effectiveness and applicability of our algorithm, but are currently not implemented:

1. It is desirable to extend the above ideas to functions of more than one variable and maybe even to vector-valued functions. The most immediate application would be to (scalar) functions of two variables and perspective (and contour) plot representations of them.

2. Currently, the attributes jagged and long are considered independently and a new pointed is only added were both are present. We feel that this does not model accurately enough the human visual perception of jaggedness of long segments. Optimally, we would rather make the decision "jagged and long" dependent on the values of both attributes length and angle. For example, a (relatively) small angle on a quite long segment may be much more problematic than a larger angle of an only moderately long segment.

3. It may be that (rotation and translation invariant) angles do not exactly correspond to the desired visual impact. For example, it is plausible that an angle on an almost horizontal line segment may be more visually disturbing than the same angle on a (45 degrees) slanted segment.

4. `NA` (for "not available") values returned by $f(\cdot)$ are currently carefully removed, including their corresponding x_i's. For two reasons, this may be undesirable. First, we are disregarding information about f and not returning it. Second, a removed point does not "attract" other points in its "neighborhood" whereas this may be desired. A better method would keep all the `NA` values, automatically declare these points as jagged and adapt all the other parts of the algorithm (lengths, slopes, etc.) in order to work with `NA`'s.

5. Currently, new abscissa values are added only one at a time possibly on each side of a jagged corner. For functions with only a few singularities (as in the second part of our example set, see appendix C, and figures 8–12), this often results in many iterations each of which adds only a few new abscissa values around the same jagged corners (see appendix C, the output of the fourth example, $\arcsin(x)$, and figure 4). It may be more efficient in these cases to add several new x_i around a given corner at once. This would be of particular importance in a parallel environment when parallel evaluation of several $f(x_i)$'s may not be much more expensive than one evaluation.

6. Parallelization of parts of the algorithm is already done somewhat in our S implementation. It may be extremely efficient on a parallel computer architecture.

References

Becker, R. A., Chambers, J. M., and Wilks, A. R. (1988). *The New S Language*. Wadsworth and Brooks/Cole, Pacific Grove, CA.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall software series. Prentice Hall, Englewood Cliffs, N.J., 2nd edition.

A S `help()` documentation

plot.function	plot a function, automatically determining abscissae	**plot.function**

```
plot.function(f, a, b, n0= 17, x= seq(a, b, length= n0), angle.max= 3,
length.min= 1/64, max.f.eval=999, log.ax="", dx.fact= 0.25,
min.dx= 10*.Machine$double.eps, do.plot= T, XLAB="", verbose.level= 1)
```

ARGUMENTS

- `f` the function to plot. Must be of mode `function(x)`.
- `a` smallest argument for `f` (leftmost point to be plotted).
- `b` largest argument for `f` (rightmost point to be plotted).
- `n0` initial number of abscissa values. Default: 17.
- `x` numeric vector of initial abscissae values. If specified, the arguments `a`, `b`, and `n0` are disgarded. This is useful if you want to start off with a very non-equispaced design.
- `angle.max` maximal angle [in degrees] that is considered tolerable. Default: 3, i.e. $3 * \pi/180$ radians.
- `length.min` minimal length of a line segment (as fraction of the plot region width).
- `max.f.eval` integer; maximal number of function evaluations allowed. Since `f` is evaluated "in parallel" on possibly many arguments, the actual number may be somewhat higher.
- `log.ax` optional character, the `log` parameter in `plot()`, i.e., any of `""`,`"x"`,`"y"` or `"xy"`. If `"x"` or `"y"` is present, a log scale is used for the corresponding axis. Only in the case of `"y"`, the algorithm computes angle and length in the transformed coordinate system. Default `""`.
- `dx.fact` optional fraction of an interval length, must be $\in (0, 0.5)$. Determines how far from a jagged corner a new abscissa is placed.
- `min.dx` optional numeric; minimal length of $x[i+1] - x[i]$ (for new points), as fraction of $b - a$. Default: 10 * .Machine $ double.eps
- `do.plot` optional logical, if `T` (default), the plot is done on the current device.
- `XLAB` optional character to be used as `xlab=` in `plot` if `do.plot` is `T`. Default `""`, for convenience.
- `verbose.level`
  optional integer in 0:3. Determines the amount of diagnostic output that is written to the terminal. 0 means "no output", 1 (default) gives reason for stop and a 1–line summary. 2 (and more) gives (at least) a line per iteration.

VALUE

list with components `x` and `y`, corresponding to {x[i]}, {f(x[i])}. If `do.plot` then `plot(x,y, log=lag.ax, type="l", xlab = XLAB)` is executed.

EXAMPLES

```
plot.function (sin, 0, pi)
plot.function (function(x) exp(-x^2), -3, 4, verb=2)

pl.gamma <- plot.function(gamma, -5, 5, verb = 2)
plot(pl.gamma, ylim = c(-50, 200), type = "b"); abline(h = 0, lty = 3)
```

```
## This is a tough one :
## lgamma  ==  ln(gamma(.))  is  NA in [-5,-4], [-3,-2] and [-1,0]
pl.lgamma <- plot.function( lgamma, -6, 5, verb = 2)
plot(pl.lgamma, ylim = c(-2, 4), type = "b"); abline(h = 0, lty = 3)
```

SEE ALSO

`tst.plot.function` which calls a set of "interesting" text examples.

B Source of S function and examples

B.1 Main `plot.function`

This is the definition of the S function `plot.function`: For the sake of efficiency, we have written one large function rather than modularizing sections as sub-functions.

Further note that a large part of the code is dedicated to error checking and to diagnostic output (an example of which is seen in appendix C).

```
plot.function <- function( f, a, b, n0 = 17, x = seq(a, b, length= n0),
                           angle.max = 3, length.min = 1/64,
                           max.f.eval=999,  log.ax= "", dx.fact= .25,
                           min.dx = 10 * .Machine$ double.eps,
                           do.plot = T, XLAB="", verbose.level = 1)
{
  if(missing(verbose.level) && exists("v.lev", frame=sys.parent()))
        verbose.level <- get("v.lev", frame=sys.parent())
  if (verbose.level <= 0)  print <- function(...){}
  if (verbose.level <= 1)  CAT <- function(...){}  else  CAT <- cat
  if (verbose.level >= 2) {
    o.op <- options(digits=5)
    if (verbose.level >= 3 && unlist(options("warn")) == 0)
         options(warn=1) # print warning()s immediately
    on.exit(options(o.op))
  }
  if (dx.fact >= .5 || dx.fact<=0) stop("argument `dx.fact' must be in (0, .5)")
  angle.max <- angle.max * pi/180 #-- convert from degrees to radians!
  if (n0 < 3) {n0 <- 3 ;   warning(paste("'n0' must be >= 3; was only",n0))}
  if(!missing(x)) { #-- x is specified differently  --
    n0 <- length(x <- sort(x))
    if (n0 < 3) stop("`x' must have >= 3 elements")
    a <- x[1];  b <- x[n0]
    CAT("x specified explicitely --> New  a=", format(a),"  b=", format(b),
        "  n0=",n0,"\n")
  } else if (a>b) { y <- a; a <- b; b <- y }

  norm.min <- (b-a)*length.min
  min.dx <- min.dx * (b-a)
  if (length(string.match("x", log.ax)))
        warning("log=\"x\" is not implemented optimally")
  do.log <- length(string.match("y", log.ax))
  f.string <- deparse(substitute(f))
  if (substring(f.string[1],1,9)=="function(") f.string <- f.string[2]
  y0 <- f(x)
  if (length(y0) != n0)
```

```
  stop(paste("function ‘", f.string,"’ does not vectorize properly!", sep=""))
old.n0 ← iter.no ← 0
CAT("‘plot.function’: ---- Diagnostics ----\n")

repeat {   iter.no ← iter.no + 1
   dx ← x[-1] - x[-n0]
   if (n0 > max.f.eval) {print("n0 > max.f.eval"); break}
   y ← as.single (if (do.log) log(y0) else  y0)
   if (any(is.na(y))) {
       warning(paste(if(do.log) "log of" else "","function", f.string,
                       "returned NA"))
       ok ← ! is.na(y)
       if (!any(ok)) stop(paste(f.string,"returned  ONLY  NA’s"))

   #- Possible improvement : Try to REPLACE x[!ok], rather then delete them.
   #-      repeat "jitter very little with x[!ok]"  until ok or "too many trys"
   #-- OR even BETTER:
   #--     leave the NA values (this IS informative after all !) and make
   #--     the rest of the code work with NA's
   #--     NA - points should probably always be considered as "jagged"
   #--     For contiguous NA intervals, you may optionally disable the
   #--     insertion of new points (that may be NA's anyway)

       y ← y[ok]; y0 ← y0[ok]
       x ← x[ok]; n0 ← length(x); dx ← x[-1] - x[-n0]
   }
   if (n0 == old.n0[1] || all(n0 < old.n0)){print("n0 did not change"); break}
   sy ← diff(range(y))/(b-a)
   if (sy > 0) y ←  y / sy  #-- Now : range(y) == range(x); use this geometry --
   dy ← y[-1] - y[-n0]
   if (any(is.na(dy))) {#- happens VERY rarely, for y → inf., e.g. f = exp(1/x)
                        #- I did not see this happen since 4/3/91, ~ 14.00
       na.dy ← is.na(dy)
       CAT('####> "plot.function": dy is NA  at', which(na.dy))
       #-- do the diff() BEFORE the scaling :
       if (any(is.na(y))) { CAT(";  y is NA, too !\n")
               print(paste("y[c(", paste(which(is.na(y)), collapse=","),
                            ")]  are NA  after scaling", sep=""))
               break
       }
       else {  CAT(";  ‘y’ is ok \n")
               dy ← diff(if (do.log) log(y0) else  y0) / sy
       }
   }
   f1 ← dy/dx #------- these are the (scaled) slopes -------

   # "jagged" corners  IFF  the angle between two line segments it too big :
   is.jagged ← abs( steep2angle(f1[-n0+1], f1[-1])) > angle.max
   #########  -- attribute of corners 2, .., n0 - 1
   is.long   ← abs(dx) + abs(dy) > norm.min
   #######     -- attribute of segments  2, .., n0
   if ((! any(is.jagged)) || (! any(is.long)))
       {print("No more 'jagged' nor 'long'"); break}
   # formatC : gives extra blanks for short integers → overall nicer format
   CAT('  old n0 = ', formatC(as.integer(n0)),
       ' jagged:', formatC(sum(is.jagged)),
       ' long:',   formatC(sum(is.long)), sep="")
   is.j.Llong ← is.jagged & is.long[-n0+1]
```

```
    is.j.Rlong <- is.jagged & is.long[-1]
    CAT('  j & l:', formatC(
       if(sum({is.j_ is.j.Llong|is.j.Rlong})<=3) x[c(F,is.j,F)] else sum(is.j)
          , 3),"\n")
    if (!any(is.j.Llong|is.j.Rlong))
       {print("No more `jagged' together with `long'");break}

    xc <- x[-c(1,n0)]
    xc <- c(xc[is.j.Llong], xc[is.j.Rlong])
    dxs <- dx.fact* c( - dx[-n0+1][is.j.Llong],          #- left  long segments
                         dx[-1]    [is.j.Rlong] )        #- right long segments
    if( any((sml.dx <- abs(dxs) < min.dx)) ) {
       CAT("  --", sum(sml.dx), "of the 'new dx' are small; corresp. xc[]:",
           formatC(xc[sml.dx],3),"\n")
       if (all(sml.dx)) {print("All new `dx's are too small"); break}
       new.x <- xc[!sml.dx] + dxs[!sml.dx]
    } else  new.x <- xc + dxs
    if (any(new.x < a) || any(new.x > b)) stop("new.x out of bounds")

    dup <- duplicated(new.x)
    if (any(dup)) { #-- this really happens rarely, but e.g., for 1/x  --
                    #-- only happens when  dx[.] ~= < .Machine$double.eps
        CAT(" --##> duplicated(new.x):", formatC(new.x[dup],3), "  dx[]:",
            formatC(c(dx[-n0+1][is.jagged], dx[-1][is.jagged])[dup],3),"\n")
        new.x <- new.x[!dup]
    }
    if (!length(new.x))  {print("new.x: length 0 from 'unique'"); break}
    not.match.x <- is.na(match(new.x, x))
    if (!all(not.match.x)) {
          CAT("--> \"plot.function\": new.x matches x ", sum(!not.match.x),
                "times"," dx=", formatC(
                c(dx[-n0+1][is.jagged], dx[-1][is.jagged])[!dup][!not.match.x]
                        ,3), "\n")
          new.x <- new.x[not.match.x]
          if (0 == length(new.x))
                {print("new.x has length 0 from matching x"); break}
    }
    x <- c(x, new.x)[ sort.x <- sort.list(c(x, new.x))]
    if (any(dup <- duplicated(x)))  ## --- should NEVER happen  --- ##
        stop(paste("--#####> \"plot.function\": duplicated(x)=",
                paste(mformat(x[dup],3),concatenate=",")))
    old.n0 <- c(n0, old.n0[1])
    n0 <- length(x)
    y0 <- c(y0, f(new.x) )[ sort.x ]

  } #-- end 'repeat'

  if (verbose.level >= 1)
    cat("`plot.function', after", iter.no,"iterations: n0 =", n0,
        ' "jagged":', sum(is.jagged), ' "long":', sum(is.long),"\n")
  if (do.plot)
        plot(x, y0, type='l', log=log.ax, ylab=f.string, xlab=XLAB)
  invisible(return(x = x, y = y0))
}

steep2angle <- function(m1,m2)
{
  #-- angle between to lines of steepness m1 & m2;  vectorizes
```

```
  atan( m2 - m1, 1 + m1*m2)
}
```

B.2 `tst.plot.function()`: The testing code

Following is the testing function that produced all the example plots. For part of its (default) output and the plots produced, see appendix C.

```
# To test the 'plot.function' with an interesting set of examples.
# For printing you may want to add "stamp" :
# postscript(); stamp()

tst.plot.function <- function(plot.f= plot.function, do.wait = F, verbose=T,
                        tst.sets= 1:2)
{
  par (mfrow = c(2,1))
  assign("d.wait",do.wait, frame= sys.nframe())
  assign("v.lev", if (verbose) 2 else 1, frame= sys.nframe())

  plo <- function(call,...) {
        its.call <- sys.call()[[2]]
        func <- its.call[[2]]
        if (is.function(func))  func <- func[[length(func)]]
        cat("\nCall: ");   dput(its.call)
        cat("Function: "); dput(func)
        cat("          ~~~~~~~~~~~~~~~~~~~~~~~~~\n")
        # if (!get("verbose", frame = sys.parent())) cat("\n")
        plot(call,..., type = "b", ylab="",
                xlab = paste(deparse(its.call), collapse=" "),
                main = paste(deparse(func), collapse=" "))
        subtit(paste(length(call$x)," points"))
        wait(get("d.wait", frame=sys.parent()))
  }
  cat("Current defaults for  `plot.function()' :\n")
  aargs(plot.f); cat("\n")

  if (any( 1 == tst.sets)) {
   plo(plot.f( sin, 0, 10))                              # 1
   plo(plot.f(function(x) sin(1/x), 0, 2))               # 2
   plo(plot.f(function(x) sin(1/x), 0, 2, n0=101))       # 3
   plo(plot.f(asin,-1,1))                                # 4
   plo(plot.f(atan,-10,10))                              # 5
   plo(plot.f(function(x) exp(-x^2), -3, 4))             # 6
   plo(plot.f(function(x) exp(-1/x^2), -.75, 2))         # 7
  }
  if (any( 2 == tst.sets)) { ## ---- Here, we have poles ------
   plo(plot.f(function(x) exp(1/x), -1, 2, n0=101), ylim=c(0,10))      # 8
   plo(plot.f(function(x) log(abs(tan(1/x))), 0,1, angle=15))          # 9
   abline ( v = 2/(pi*1:12), lty = 3, lwd = .5)
   plo(plot.f(function(x) log(abs(tan(1/x))), 0,1, angle=15, length.min=1/20)) # 10
   abline ( v = 2/(pi*1:12), lty = 3, lwd = .5)
   plo(plot.f(function(x) log(abs(tan(1/x))), 0,.75, ang=15, log='x'), log='x')# 11
   abline ( v = 2/(pi*1:16), lty = 3, lwd = .5)
   plo(plot.f(function(x) log(abs(sin(pi*x))), 0,6, angle=15))         # 12
  }
}
```

C A Set of Examples

The following set of examples is mainly for illustrative purposes. The examples were chosen for "interestingness" and to show some of the properties of the algorithm. The following pictures were created by executing the S function `tst.plot.function()`. The S source of this test function was given in B.2, and the plots are given in at the end of this section.

Here is the output produced in the first part of `tst.plot.function()`:

```
Fri Jul 12 16:47:48 EDT 1991
> tst.plot.function()
Current defaults for  'plot.function()' :
function(f, a, b, n0= 17, x= seq(a, b, length = n0), angle.max= 3,
         length.min= 1/64, max.f.eval= 999, log.ax= "", dx.fact= 0.25,
         min.dx= 10 * .Machine$double.eps, do.plot= T, XLAB= "", verbose.level= 1)


Call: plot.function(sin, 0, 10)
Function: sin
          ~~~~~~~~~~~~~~~~~~~~~~~~~
'plot.function': ---- Diagnostics ----
  old n0 =  17 jagged: 12 long: 16  j & l:   12
  old n0 =  41 jagged: 24 long: 40  j & l:   24
  old n0 =  89 jagged: 43 long: 57  j & l:   31
  old n0 = 126 jagged: 38 long: 46  j & l:
[1] "No more 'jagged' together with 'long'"
'plot.function', after 4 iterations: n0 = 126  "jagged": 38  "long": 46
Warning in axes: No room for ylab

Call: plot.function(function(x) sin(1/x), 0, 2)
Function: sin(1/x)
          ~~~~~~~~~~~~~~~~~~~~~~~~~
'plot.function': ---- Diagnostics ----
  old n0 =  16 jagged:  5 long: 15  j & l:    5
  old n0 =  26 jagged: 11 long: 25  j & l:   11
  old n0 =  48 jagged: 20 long: 27  j & l:   12
  old n0 =  65 jagged: 17 long: 28  j & l: 0.195 0.201 0.213
  old n0 =  70 jagged: 17 long: 27  j & l:
[1] "No more 'jagged' together with 'long'"
'plot.function', after 5 iterations: n0 = 70  "jagged": 17  "long": 27
Warning in axes: No room for ylab

Call: plot.function(function(x) sin(1/x), 0, 2, n0 = 101)
Function: sin(1/x)
          ~~~~~~~~~~~~~~~~~~~~~~~~~
'plot.function': ---- Diagnostics ----
  old n0 = 100 jagged: 23 long: 28  j & l:   17
  old n0 = 133 jagged: 26 long: 38  j & l:   11
  old n0 = 153 jagged: 28 long: 48  j & l:   11
  old n0 = 172 jagged: 30 long: 60  j & l:   10
  old n0 = 189 jagged: 34 long: 71  j & l:   11
  old n0 = 205 jagged: 35 long: 77  j & l:    6
  old n0 = 213 jagged: 35 long: 82  j & l:    4
```

```
  old n0 = 220 jagged: 37 long: 87  j & l:    5
  old n0 = 228 jagged: 37 long: 91  j & l: 0.022 0.0236 0.0254
  old n0 = 232 jagged: 38 long: 93  j & l: 0.022 0.0255
  old n0 = 235 jagged: 38 long: 93  j & l:
[1] "No more ‘jagged’ together with ‘long’"
‘plot.function’, after 11 iterations: n0 = 235  "jagged": 38  "long": 93
Warning in axes: No room for ylab

Call: plot.function(asin, -1, 1)
Function: asin
          ~~~~~~~~~~~~~~~~~~~~~~~~~
‘plot.function’: ---- Diagnostics ----
  old n0 =  17 jagged:  6 long: 16  j & l:    6
  old n0 =  29 jagged:  6 long: 28  j & l:    6
  old n0 =  41 jagged:  2 long: 32  j & l: -0.93 0.93
  old n0 =  45 jagged:  2 long: 34  j & l: -0.947 0.947
  old n0 =  49 jagged:  2 long: 36  j & l: -0.96 0.96
  old n0 =  53 jagged:  2 long: 36  j & l: -0.97 0.97
  old n0 =  57 jagged:  2 long: 34  j & l: -0.978 0.978
  old n0 =  59 jagged:  2 long: 34  j & l: -0.983 0.983
  old n0 =  61 jagged:  2 long: 34  j & l: -0.987 0.987
  old n0 =  63 jagged:  2 long: 34  j & l: -0.991 0.991
  old n0 =  65 jagged:  2 long: 34  j & l: -0.993 0.993
  old n0 =  67 jagged:  2 long: 34  j & l: -0.995 0.995
  old n0 =  69 jagged:  2 long: 34  j & l: -0.996 0.996
  old n0 =  71 jagged:  2 long: 34  j & l: -0.997 0.997
  old n0 =  73 jagged:  2 long: 34  j & l: -0.998 0.998
[1] "No more ’jagged’ nor ’long’"
‘plot.function’, after 16 iterations: n0 = 75  "jagged": 0  "long": 34
Warning in axes: No room for ylab

Call: plot.function(atan, -10, 10)
Function: atan
          ~~~~~~~~~~~~~~~~~~~~~~~~~
‘plot.function’: ---- Diagnostics ----
  old n0 =  17 jagged: 10 long: 16  j & l:   10
  old n0 =  37 jagged: 20 long: 36  j & l:   20
  old n0 =  77 jagged: 10 long: 36  j & l:   10
[1] "No more ’jagged’ nor ’long’"
‘plot.function’, after 4 iterations: n0 = 93  "jagged": 0  "long": 36
Warning in axes: No room for ylab

Call: plot.function(function(x) exp( - x^2), -3, 4)
Function: exp( - x^2)
          ~~~~~~~~~~~~~~~~~~~~~~~~~
‘plot.function’: ---- Diagnostics ----
  old n0 =  17 jagged: 12 long: 16  j & l:   12
  old n0 =  41 jagged: 23 long: 40  j & l:   23
  old n0 =  87 jagged: 30 long: 48  j & l:   21
  old n0 = 115 jagged: 26 long: 40  j & l: -0.184 -0.129
  old n0 = 117 jagged: 26 long: 39  j & l:
[1] "No more ‘jagged’ together with ‘long’"
‘plot.function’, after 5 iterations: n0 = 117  "jagged": 26  "long": 39
```

```
Warning in axes: No room for ylab

Call: plot.function(function(x) exp(-1/x^2), -0.75, 2)
Function: exp(-1/x^2)
          ~~~~~~~~~~~~~~~~~~~~~~~~~
'plot.function': ---- Diagnostics ----
  old n0 =  17 jagged: 10 long: 16  j & l:   10
  old n0 =  37 jagged: 13 long: 36  j & l:   13
  old n0 =  63 jagged: 20 long: 36  j & l:    8
  old n0 =  74 jagged: 17 long: 32  j & l: 0.528
  old n0 =  75 jagged: 16 long: 31  j & l:
[1] "No more 'jagged' together with 'long'"
'plot.function', after 5 iterations: n0 = 75  "jagged": 16  "long": 31
Warning in axes: No room for ylab
```

Following are the plots as produced by `tst.plot.function()`, see B.2 for the S code.

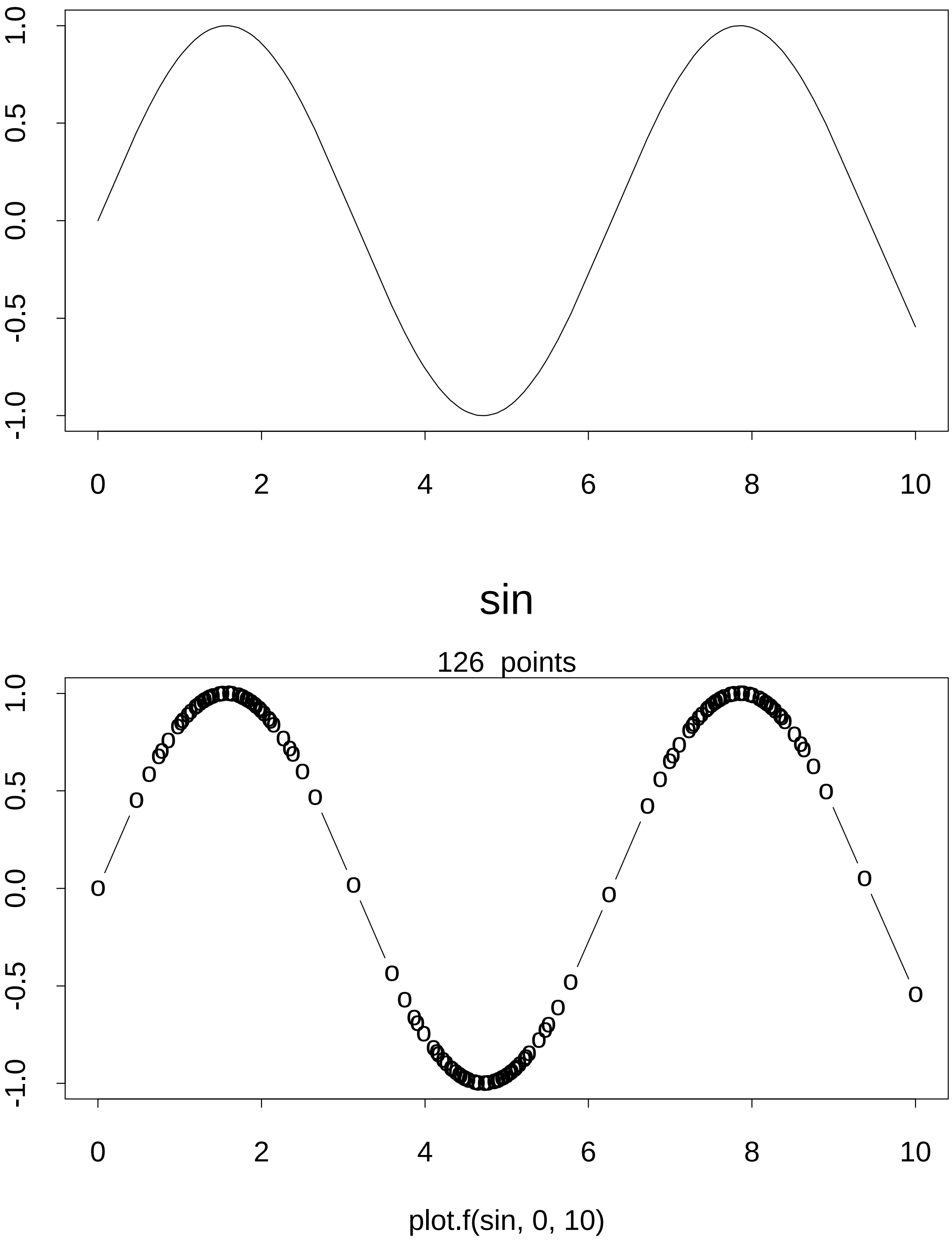
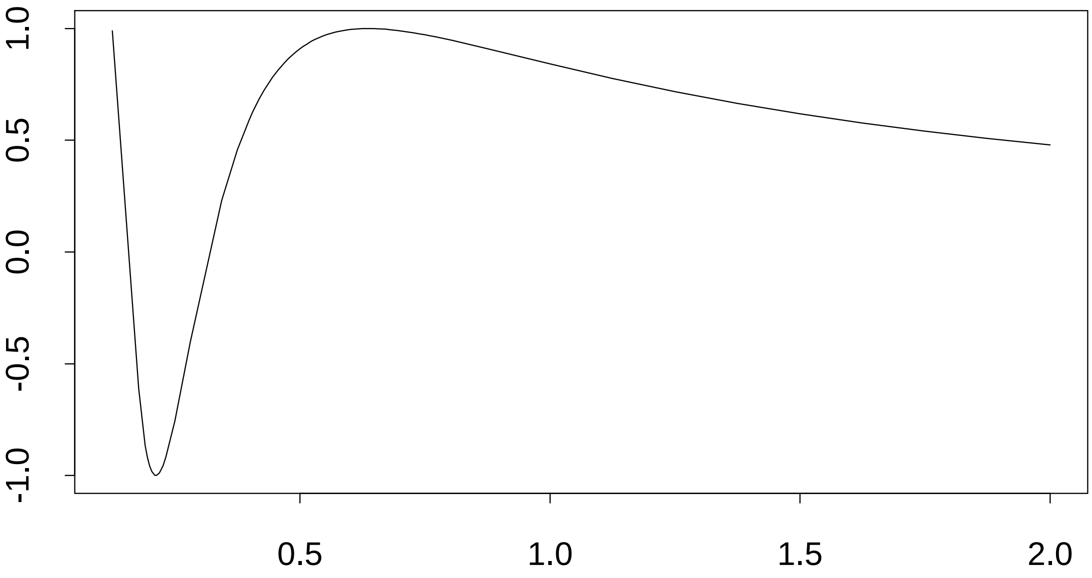


Figure 1: The simple sine function. No corners are visible in spite of relatively long stretches between some abscissa values.



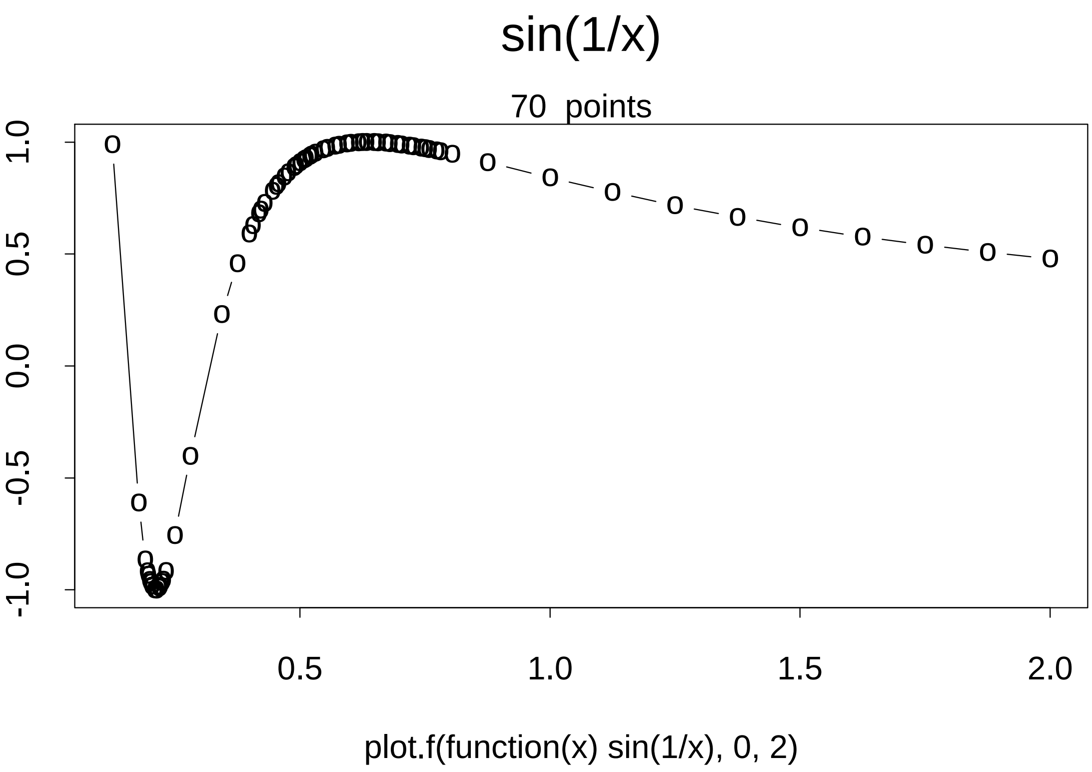


Figure 2: $sin(1/x)$. With only 74 points, a very smooth appearance, but see fig. 3 !

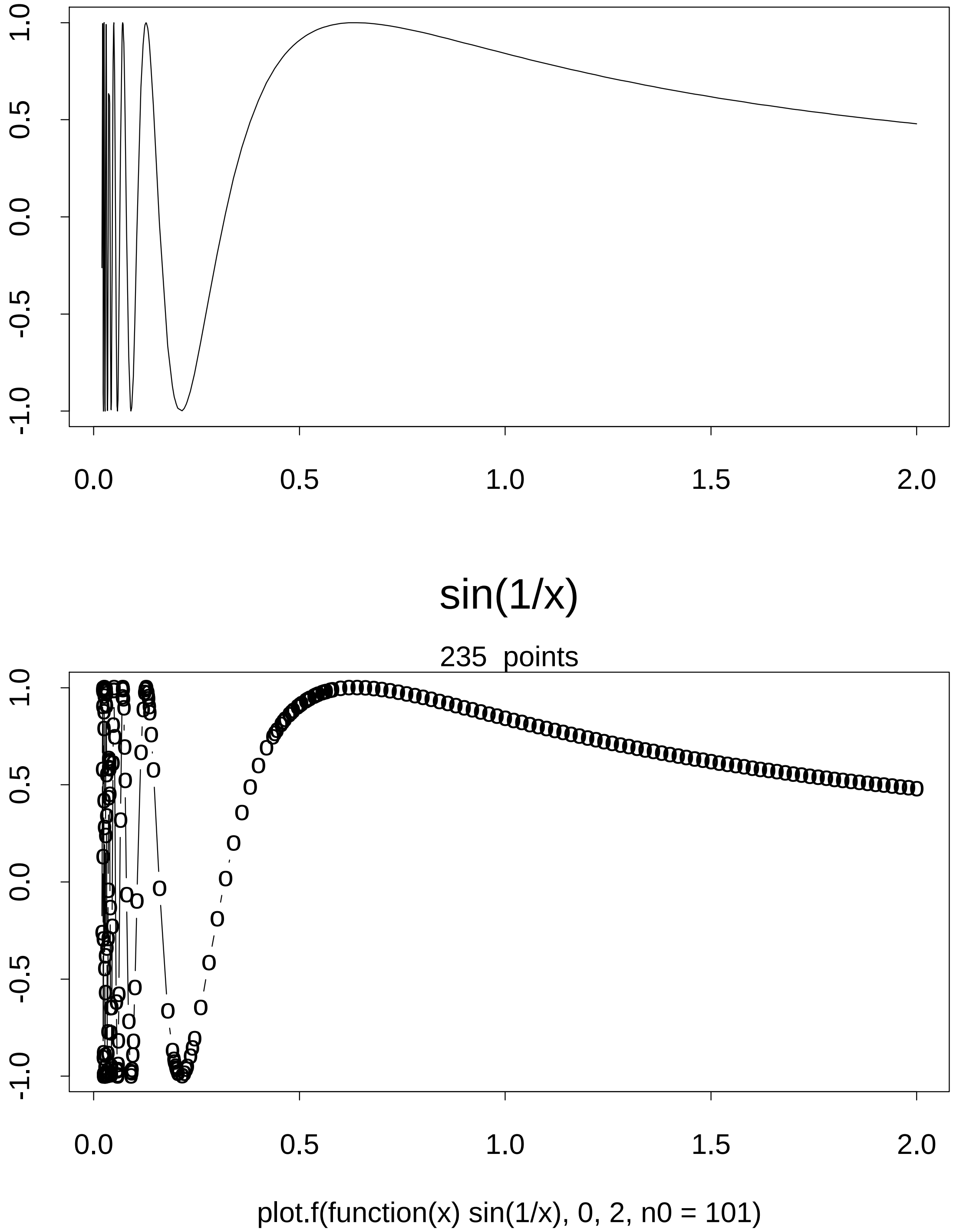


Figure 3: $sin(1/x)$, with more starting points in order to detect the "chaos" around 0 ($sin(1/x)$ oscillates infinitely fast for $x \to 0$ and is therefore discontinuous at $x = 0$).

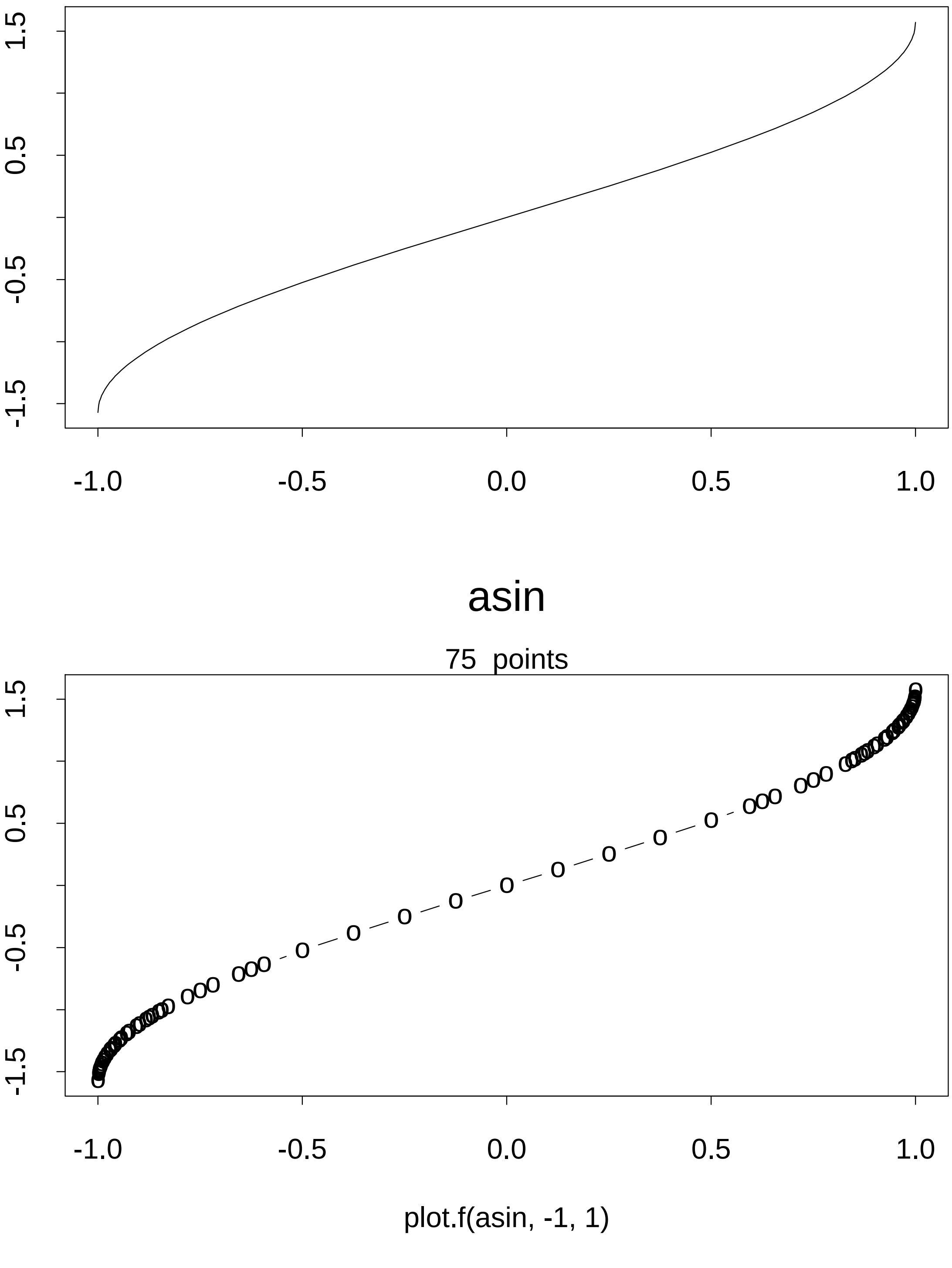


Figure 4: $\arcsin(x)$; the points with infinite slope are well modeled (but currently need many iterations, see output above).

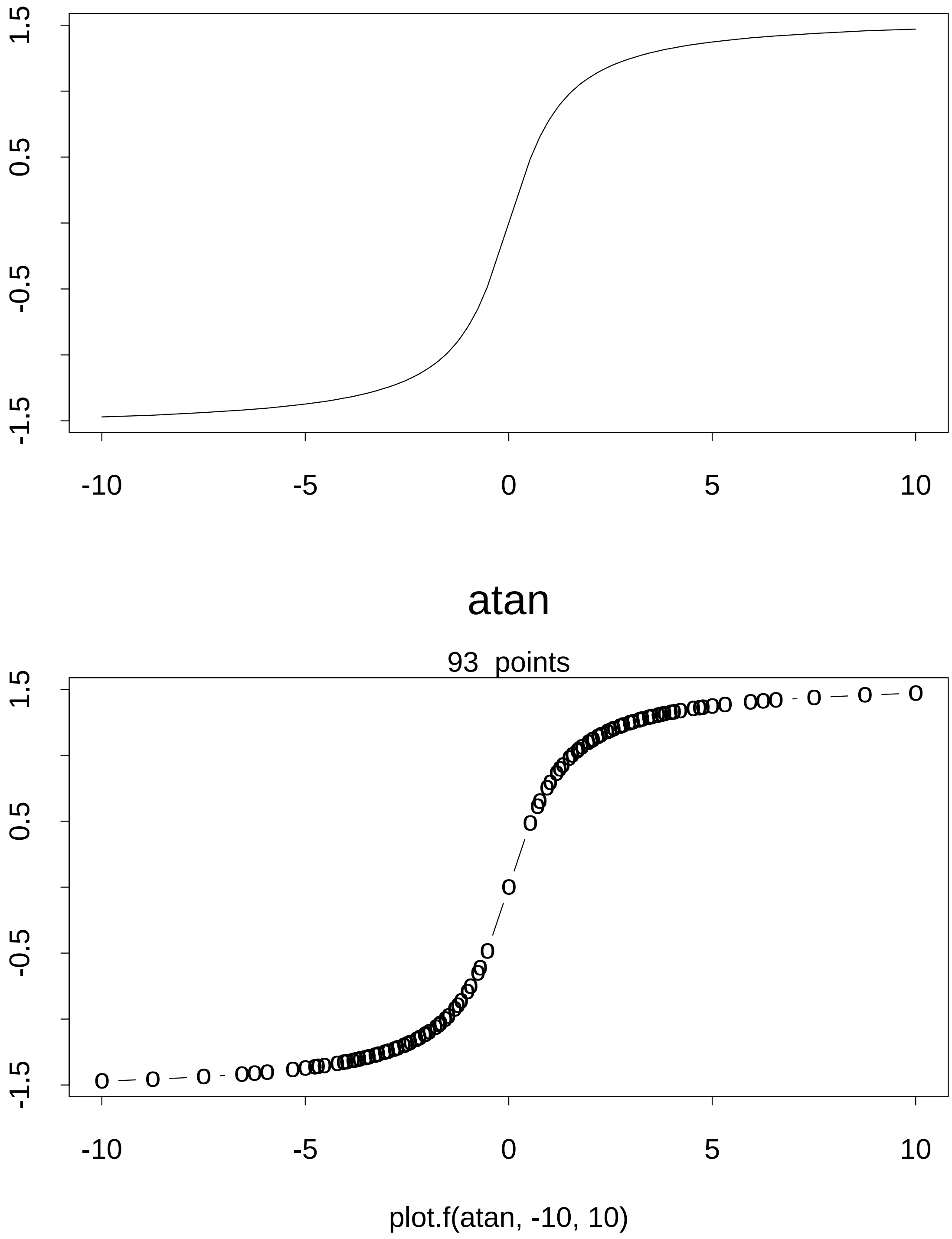


Figure 5: $\arctan(x)$.

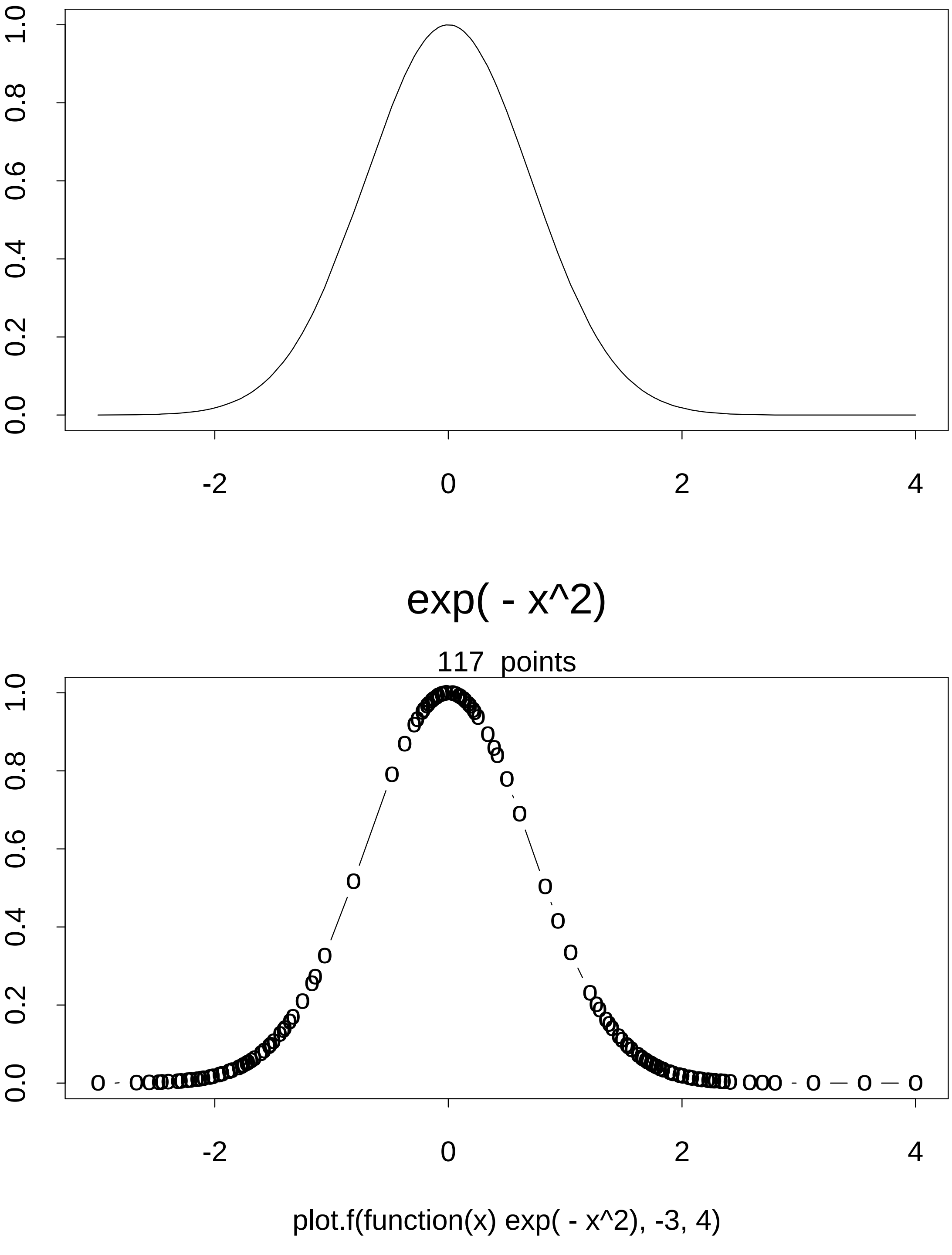


Figure 6: e^{-x^2}, the (unscaled) normal density, “Gauss’ bell curve”

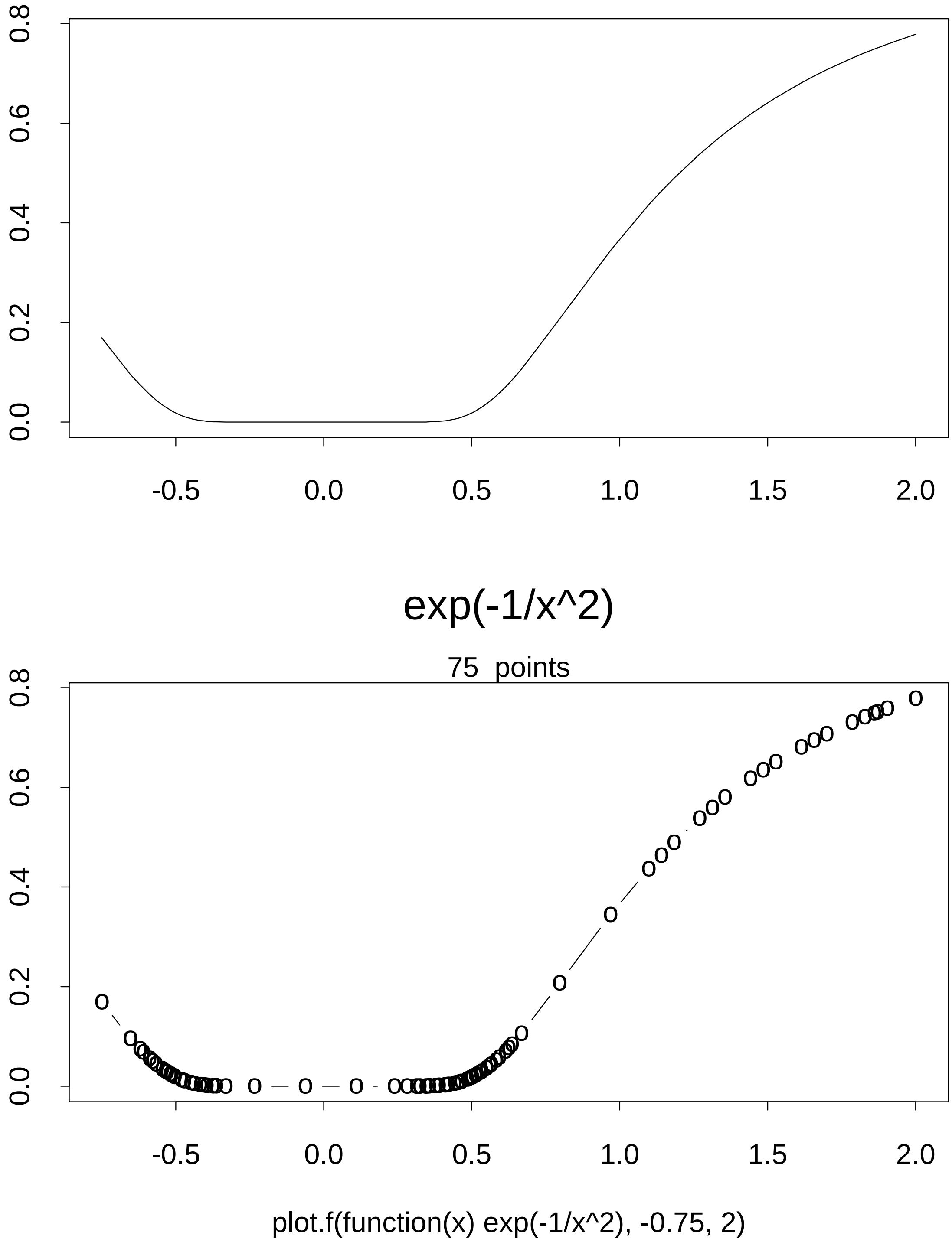


Figure 7: e^{-1/x^2}, *the* calculus example of a function $f(x)$ with no Taylor series expansion at $x = 0$.

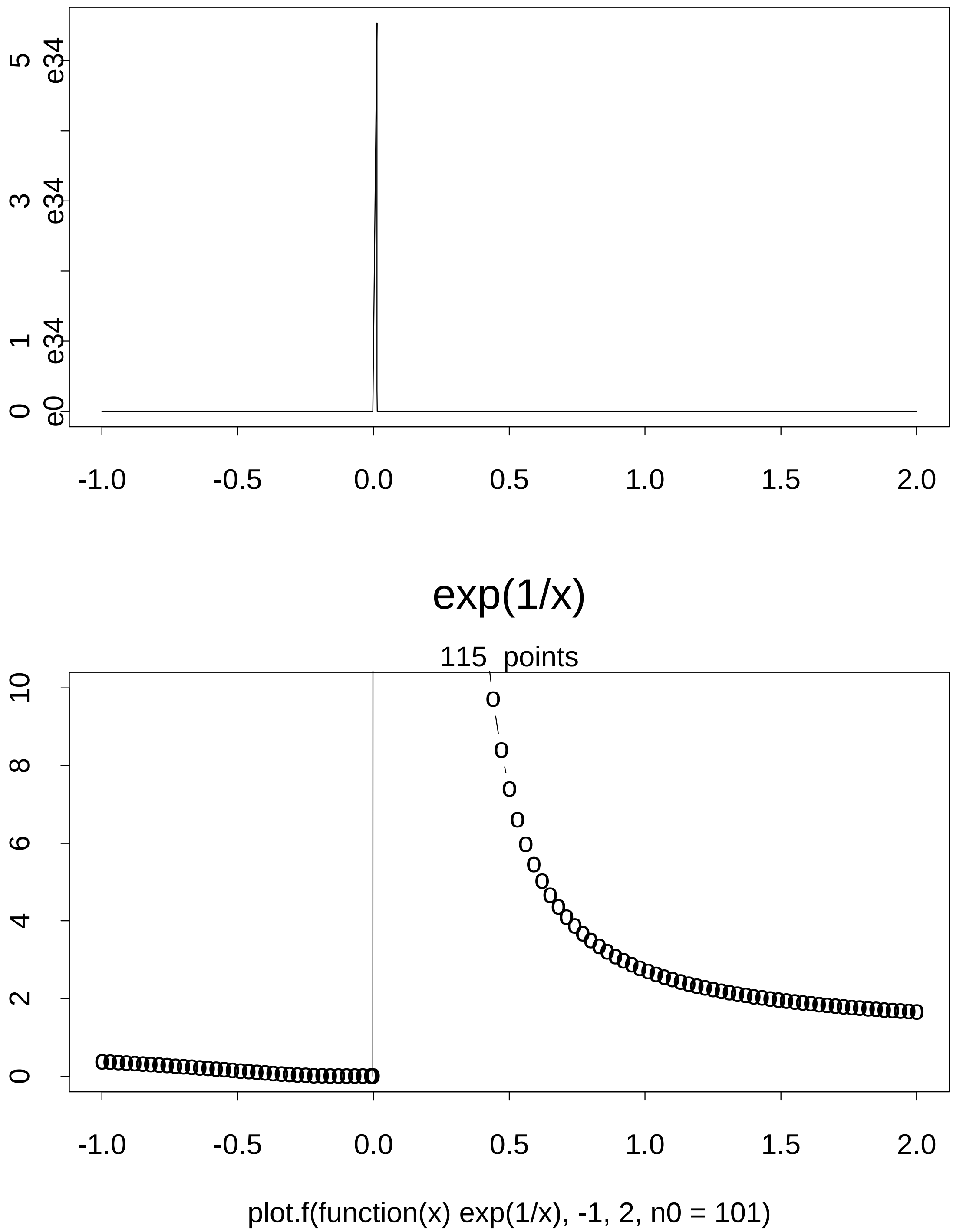


Figure 8: $e^{1/x}$, discontinuous (but left-continuous) at $x = 0$; care had to be taken to deal with "infinite" values and slopes.

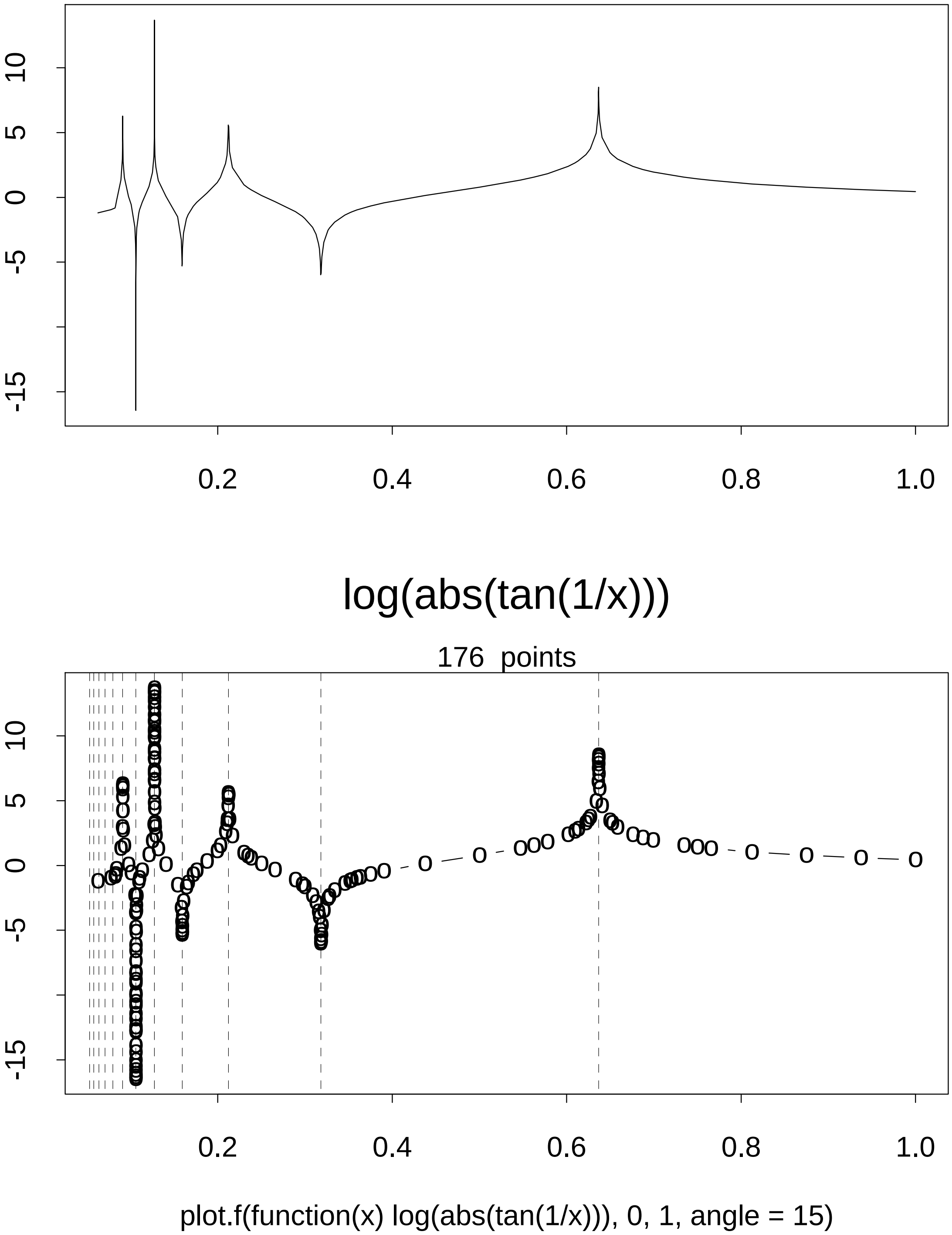


Figure 9: $\log(|\tan(1/x)|)$ with many poles ("chaotic" for $x \to 0$); the vertical lines indicate true pole locations; Note that `angle=15` is rather high and leads to some visible `jagged`ness, (e.g., to the right of the three rightmost positive poles.,) cf. fig. 10 and 11.

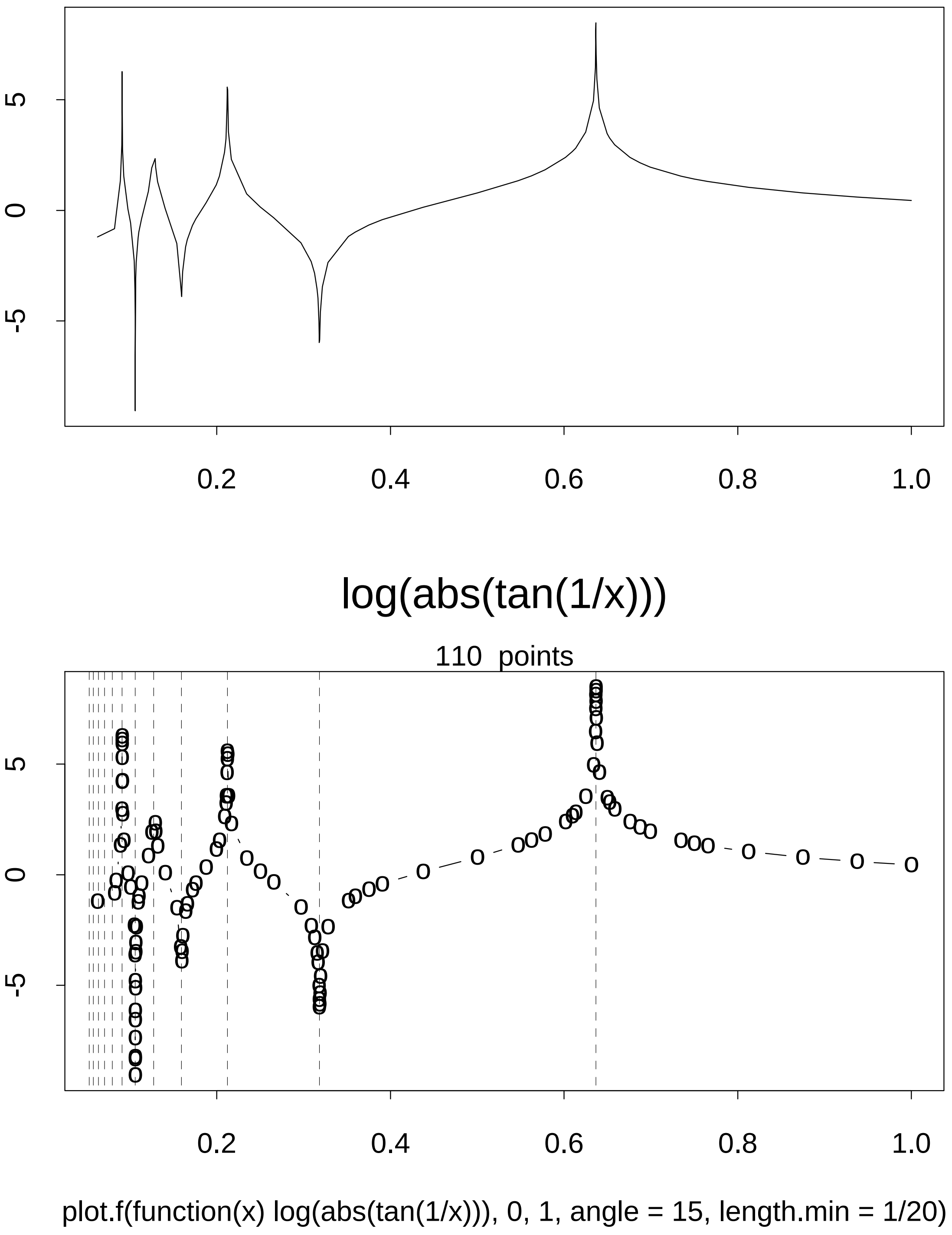


Figure 10: $\log(|\tan(1/x)|)$; setting `length.min` to a (relatively) high value gives too long **jagged** line segments; cf. fig. 9 and 11

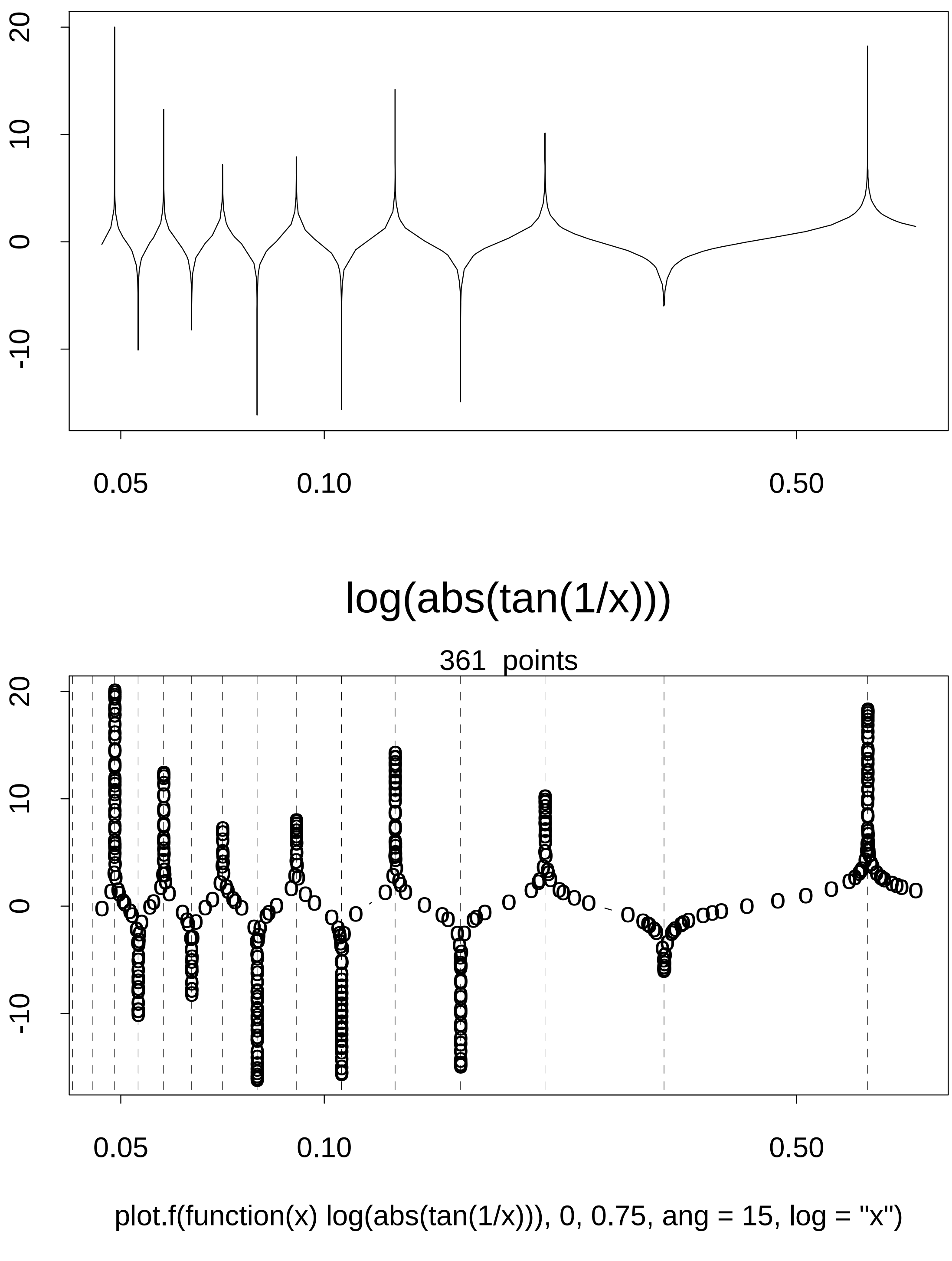


Figure 11: $\log(|\tan(1/x)|)$ with a log scale for x. More poles are automatically detected than in fig. 9, because the x-interval is smaller which increases the resolution; see also fig.10.

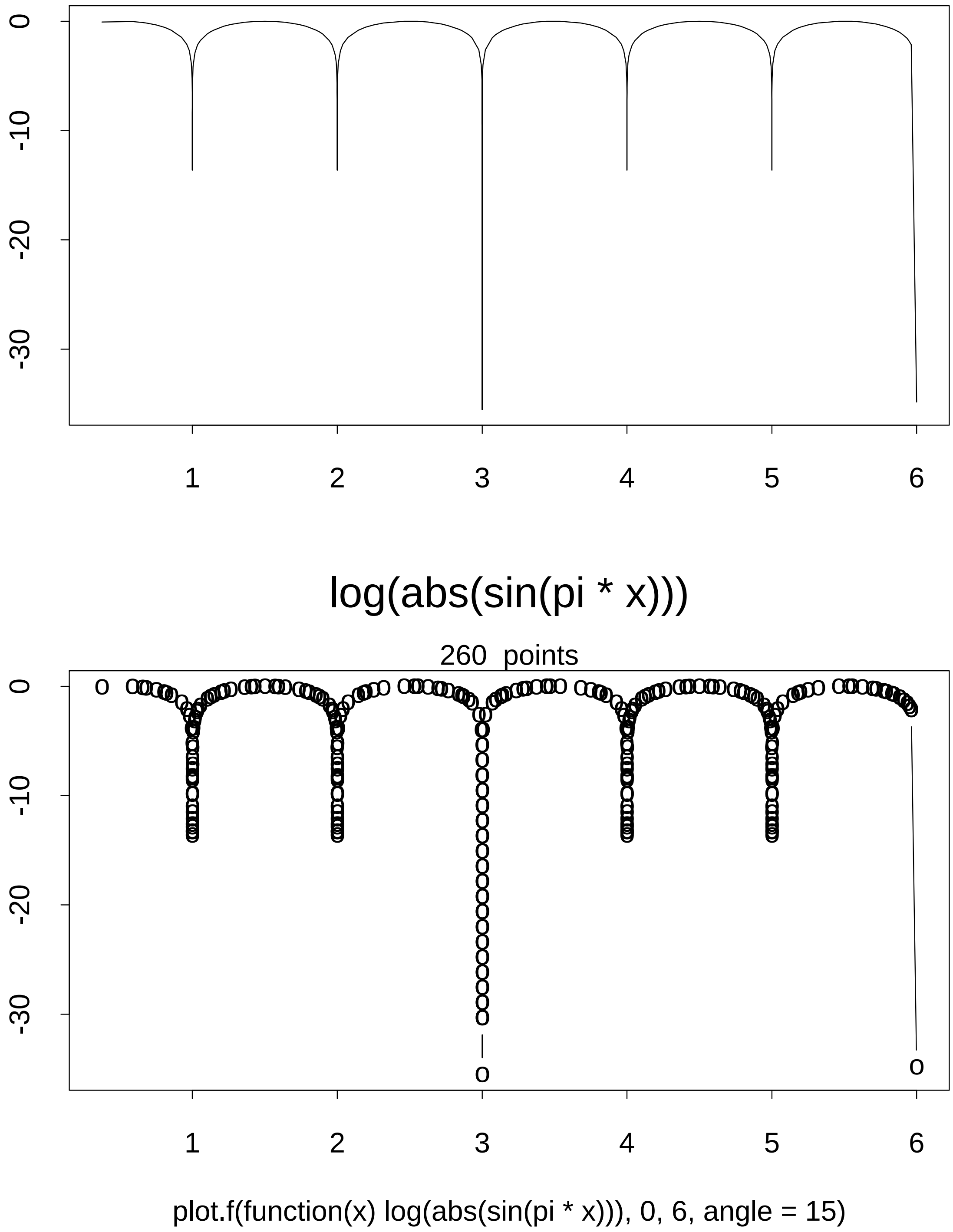


Figure 12: $\log(|\sin(\pi x)|)$, the "Romanic arches". Note that "$f(0) = -\infty$" which (currently) eliminates 0 from the set of abscissas.