8 Probability distributions

8.1 R as a set of statistical tables

One convenient use of R is to provide a comprehensive set of statistical tables. Functions are provided to evaluate the cumulative distribution function $P(X \le x)$, the probability density function and the quantile function (given q, the smallest x such that $P(X \le x) > q$), and to simulate from the distribution.

Distribution	**R name**	**additional arguments**
beta	`beta`	`shape1, shape2, ncp`
binomial	`binom`	`size, prob`
Cauchy	`cauchy`	`location, scale`
chi-squared	`chisq`	`df, ncp`
exponential	`exp`	`rate`
F	`f`	`df1, df2, ncp`
gamma	`gamma`	`shape, scale`
geometric	`geom`	`prob`
hypergeometric	`hyper`	`m, n, k`
log-normal	`lnorm`	`meanlog, sdlog`
logistic	`logis`	`location, scale`
negative binomial	`nbinom`	`size, prob`
normal	`norm`	`mean, sd`
Poisson	`pois`	`lambda`
signed rank	`signrank`	`n`
Student's t	`t`	`df, ncp`
uniform	`unif`	`min, max`
Weibull	`weibull`	`shape, scale`
Wilcoxon	`wilcox`	`m, n`

Prefix the name given here by '`d`' for the density, '`p`' for the CDF, '`q`' for the quantile function and '`r`' for simulation (*r*andom deviates). The first argument is `x` for `dxxx`, `q` for `pxxx`, `p` for `qxxx` and `n` for `rxxx` (except for `rhyper`, `rsignrank` and `rwilcox`, for which it is `nn`). In not quite all cases is the non-centrality parameter `ncp` currently available: see the on-line help for details.

The `pxxx` and `qxxx` functions all have logical arguments `lower.tail` and `log.p` and the `dxxx` ones have `log`. This allows, e.g., getting the cumulative (or "integrated") *hazard* function, $H(t) = -\log(1 - F(t))$, by

```
- pxxx(t, ..., lower.tail = FALSE, log.p = TRUE)
```

or more accurate log-likelihoods (by `dxxx(..., log = TRUE)`), directly.

In addition there are functions `ptukey` and `qtukey` for the distribution of the studentized range of samples from a normal distribution, and `dmultinom` and `rmultinom` for the multinomial distribution. Further distributions are available in contributed packages, notably **SuppDists** (`https://CRAN.R-project.org/package=SuppDists`).

Here are some examples

```
> ## 2-tailed p-value for t distribution
> 2*pt(-2.43, df = 13)
> ## upper 1% point for an F(2, 7) distribution
> qf(0.01, 2, 7, lower.tail = FALSE)
```

See the on-line help on `RNG` for how random-number generation is done in R.

8.2 Examining the distribution of a set of data

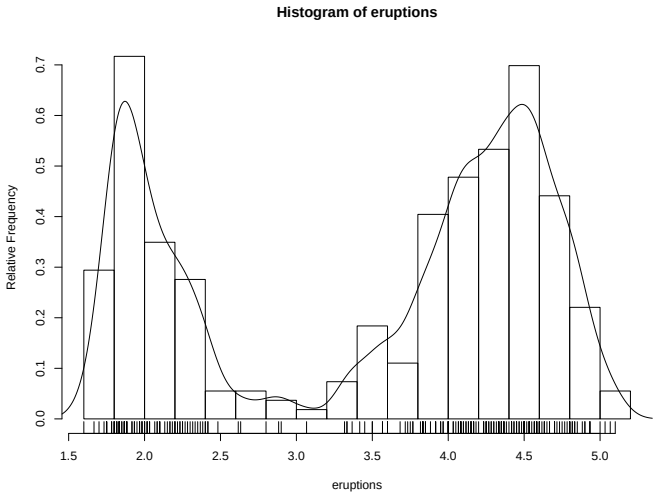
Given a (univariate) set of data we can examine its distribution in a large number of ways. The simplest is to examine the numbers. Two slightly different summaries are given by `summary` and `fivenum` and a display of the numbers by `stem` (a "stem and leaf" plot).

```
> attach(faithful)
> summary(eruptions)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.600   2.163   4.000   3.488   4.454   5.100
> fivenum(eruptions)
[1] 1.6000 2.1585 4.0000 4.4585 5.1000
> stem(eruptions)

  The decimal point is 1 digit(s) to the left of the |

  16 | 070355555588
  18 | 000022233333335577777777888822335777888
  20 | 00002223378800035778
  22 | 0002335578023578
  24 | 00228
  26 | 23
  28 | 080
  30 | 7
  32 | 2337
  34 | 250077
  36 | 0000823577
  38 | 2333335582225577
  40 | 000000335778888800223355557778
  42 | 033355557788002333335555777 78
  44 | 022223355577800000002333335777 8888
  46 | 00002333577000000 23578
  48 | 00000022335800333
  50 | 0370
```

A stem-and-leaf plot is like a histogram, and R has a function `hist` to plot histograms.

```
> hist(eruptions)
## make the bins smaller, make a plot of density
> hist(eruptions, seq(1.6, 5.2, 0.2), prob=TRUE)
> lines(density(eruptions, bw=0.1))
> rug(eruptions) # show the actual data points
```

More elegant density plots can be made by `density`, and we added a line produced by `density` in this example. The bandwidth `bw` was chosen by trial-and-error as the default gives too much smoothing (it usually does for "interesting" densities). (Better automated methods of bandwidth choice are available, and in this example `bw = "SJ"` gives a good result.)

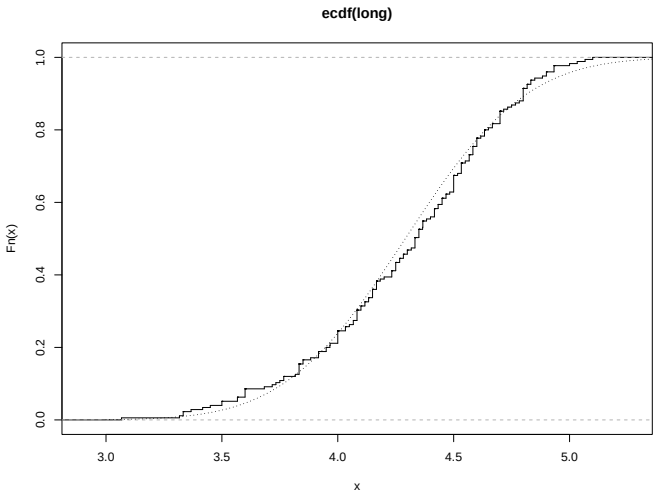


We can plot the empirical cumulative distribution function by using the function `ecdf`.

```
> plot(ecdf(eruptions), do.points=FALSE, verticals=TRUE)
```

This distribution is obviously far from any standard distribution. How about the right-hand mode, say eruptions of longer than 3 minutes? Let us fit a normal distribution and overlay the fitted CDF.

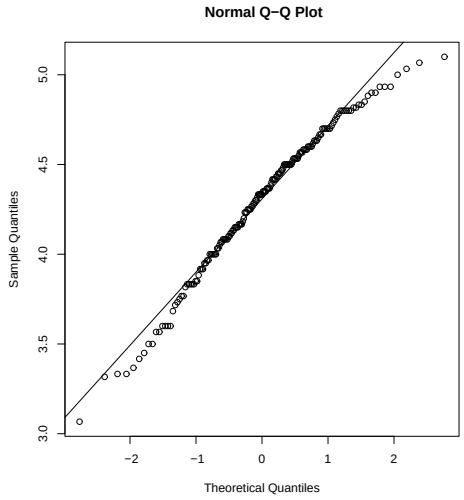
```
> long <- eruptions[eruptions > 3]
> plot(ecdf(long), do.points=FALSE, verticals=TRUE)
> x <- seq(3, 5.4, 0.01)
> lines(x, pnorm(x, mean=mean(long), sd=sqrt(var(long))), lty=3)
```



Quantile-quantile (Q-Q) plots can help us examine this more carefully.

```
par(pty="s")       # arrange for a square figure region
qqnorm(long); qqline(long)
```

which shows a reasonable fit but a shorter right tail than one would expect from a normal distribution. Let us compare this with some simulated data from a t distribution




```
x <- rt(250, df = 5)
qqnorm(x); qqline(x)
```

which will usually (if it is a random sample) show longer tails than expected for a normal. We can make a Q-Q plot against the generating distribution by

```
qqplot(qt(ppoints(250), df = 5), x, xlab = "Q-Q plot for t dsn")
qqline(x)
```

Finally, we might want a more formal test of agreement with normality (or not). R provides the Shapiro-Wilk test

```
> shapiro.test(long)

        Shapiro-Wilk normality test

data:  long
W = 0.9793, p-value = 0.01052
```

and the Kolmogorov-Smirnov test

```
> ks.test(long, "pnorm", mean = mean(long), sd = sqrt(var(long)))

        One-sample Kolmogorov-Smirnov test

data:  long
D = 0.0661, p-value = 0.4284
alternative hypothesis: two.sided
```

(Note that the distribution theory is not valid here as we have estimated the parameters of the normal distribution from the same sample.)

8.3 One- and two-sample tests

So far we have compared a single sample to a normal distribution. A much more common operation is to compare aspects of two samples. Note that in R, all "classical" tests including the ones used below are in package **stats** which is normally loaded.

Consider the following sets of data on the latent heat of the fusion of ice (*cal/gm*) from Rice (1995, p.490)

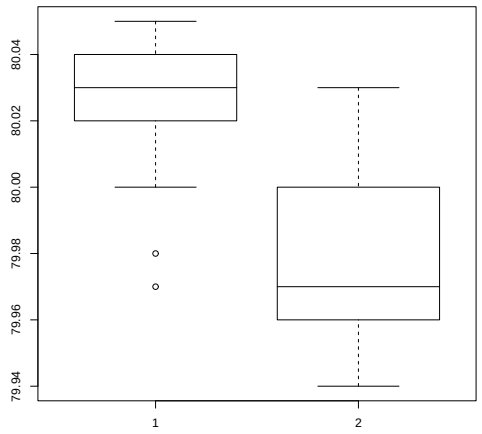
```
Method A: 79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
          80.05 80.03 80.02 80.00 80.02
```

```
Method B: 80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97
```

Boxplots provide a simple graphical comparison of the two samples.

```
A <- scan()
79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
80.05 80.03 80.02 80.00 80.02

B <- scan()
80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97

boxplot(A, B)
```

which indicates that the first group tends to give higher results than the second.



To test for the equality of the means of the two examples, we can use an *unpaired* t-test by

```
> t.test(A, B)

        Welch Two Sample t-test

data:  A and B
t = 3.2499, df = 12.027, p-value = 0.00694
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.01385526 0.07018320
sample estimates:
mean of x mean of y
 80.02077  79.97875
```

which does indicate a significant difference, assuming normality. By default the R function does not assume equality of variances in the two samples. We can use the F test to test for equality in the variances, provided that the two samples are from normal populations.

```
> var.test(A, B)

        F test to compare two variances

data:  A and B
F = 0.5837, num df = 12, denom df =  7, p-value = 0.3938
alternative hypothesis: true ratio of variances is not equal to 1
```

```
95 percent confidence interval:
 0.1251097 2.1052687
sample estimates:
ratio of variances
         0.5837405
```

which shows no evidence of a significant difference, and so we can use the classical t-test that assumes equality of the variances.

```
> t.test(A, B, var.equal=TRUE)

         Two Sample t-test

data:  A and B
t = 3.4722, df = 19, p-value = 0.002551
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.01669058 0.06734788
sample estimates:
mean of x mean of y
 80.02077  79.97875
```

All these tests assume normality of the two samples. The two-sample Wilcoxon (or Mann-Whitney) test only assumes a common continuous distribution under the null hypothesis.

```
> wilcox.test(A, B)

         Wilcoxon rank sum test with continuity correction

data:  A and B
W = 89, p-value = 0.007497
alternative hypothesis: true location shift is not equal to 0

Warning message:
Cannot compute exact p-value with ties in: wilcox.test(A, B)
```

Note the warning: there are several ties in each sample, which suggests strongly that these data are from a discrete distribution (probably due to rounding).

There are several ways to compare graphically the two samples. We have already seen a pair of boxplots. The following

```
> plot(ecdf(A), do.points=FALSE, verticals=TRUE, xlim=range(A, B))
> plot(ecdf(B), do.points=FALSE, verticals=TRUE, add=TRUE)
```

will show the two empirical CDFs, and `qqplot` will perform a Q-Q plot of the two samples. The Kolmogorov-Smirnov test is of the maximal vertical distance between the two ecdfs, assuming a common continuous distribution:

```
> ks.test(A, B)

         Two-sample Kolmogorov-Smirnov test

data:  A and B
D = 0.5962, p-value = 0.05919
alternative hypothesis: two-sided

Warning message:
cannot compute correct p-values with ties in: ks.test(A, B)
```

9 Grouping, loops and conditional execution

9.1 Grouped expressions

R is an expression language in the sense that its only command type is a function or expression which returns a result. Even an assignment is an expression whose result is the value assigned, and it may be used wherever any expression may be used; in particular multiple assignments are possible.

Commands may be grouped together in braces, {*expr_1*; ...; *expr_m*}, in which case the value of the group is the result of the last expression in the group evaluated. Since such a group is also an expression it may, for example, be itself included in parentheses and used as part of an even larger expression, and so on.

9.2 Control statements

9.2.1 Conditional execution: `if` statements

The language has available a conditional construction of the form

```
> if (expr_1) expr_2 else expr_3
```

where *expr_1* must evaluate to a single logical value and the result of the entire expression is then evident.

The "short-circuit" operators `&&` and `||` are often used as part of the condition in an `if` statement. Whereas `&` and `|` apply element-wise to vectors, `&&` and `||` apply to vectors of length one, and only evaluate their second argument if necessary.

There is a vectorized version of the `if`/`else` construct, the `ifelse` function. This has the form `ifelse(condition, a, b)` and returns a vector of the same length as `condition`, with elements `a[i]` if `condition[i]` is true, otherwise `b[i]` (where `a` and `b` are recycled as necessary).

9.2.2 Repetitive execution: `for` loops, `repeat` and `while`

There is also a `for` loop construction which has the form

```
> for (name in expr_1) expr_2
```

where *name* is the loop variable. *expr_1* is a vector expression, (often a sequence like `1:20`), and *expr_2* is often a grouped expression with its sub-expressions written in terms of the dummy *name*. *expr_2* is repeatedly evaluated as *name* ranges through the values in the vector result of *expr_1*.

As an example, suppose `ind` is a vector of class indicators and we wish to produce separate plots of `y` versus `x` within classes. One possibility here is to use `coplot()`,[1] which will produce an array of plots corresponding to each level of the factor. Another way to do this, now putting all plots on the one display, is as follows:

```
> xc <- split(x, ind)
> yc <- split(y, ind)
> for (i in 1:length(yc)) {
    plot(xc[[i]], yc[[i]])
    abline(lsfit(xc[[i]], yc[[i]]))
  }
```

(Note the function `split()` which produces a list of vectors obtained by splitting a larger vector according to the classes specified by a factor. This is a useful function, mostly used in connection with boxplots. See the `help` facility for further details.)

[1] to be discussed later, or use `xyplot` from package **lattice** (`https://CRAN.R-project.org/package=lattice`).

Warning: `for()` loops are used in R code much less often than in compiled languages. Code that takes a 'whole object' view is likely to be both clearer and faster in R.

Other looping facilities include the

```
> repeat expr
```

statement and the

```
> while (condition) expr
```

statement.

The `break` statement can be used to terminate any loop, possibly abnormally. This is the only way to terminate `repeat` loops.

The `next` statement can be used to discontinue one particular cycle and skip to the "next".

Control statements are most often used in connection with *functions* which are discussed in Chapter 10 [Writing your own functions], page 41, and where more examples will emerge.

10 Writing your own functions

As we have seen informally along the way, the R language allows the user to create objects of mode *function*. These are true R functions that are stored in a special internal form and may be used in further expressions and so on. In the process, the language gains enormously in power, convenience and elegance, and learning to write useful functions is one of the main ways to make your use of R comfortable and productive.

It should be emphasized that most of the functions supplied as part of the R system, such as `mean()`, `var()`, `postscript()` and so on, are themselves written in R and thus do not differ materially from user written functions.

A function is defined by an assignment of the form

```
> name <- function(arg_1, arg_2, ...) expression
```

The *expression* is an R expression, (usually a grouped expression), that uses the arguments, *arg_i*, to calculate a value. The value of the expression is the value returned for the function.

A call to the function then usually takes the form `name(expr_1, expr_2, ...)` and may occur anywhere a function call is legitimate.

10.1 Simple examples

As a first example, consider a function to calculate the two sample t-statistic, showing "all the steps". This is an artificial example, of course, since there are other, simpler ways of achieving the same end.

The function is defined as follows:

```
> twosam <- function(y1, y2) {
    n1  <- length(y1); n2  <- length(y2)
    yb1 <- mean(y1);   yb2 <- mean(y2)
    s1  <- var(y1);    s2  <- var(y2)
    s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
    tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
    tst
  }
```

With this function defined, you could perform two sample t-tests using a call such as

```
> tstat <- twosam(data$male, data$female); tstat
```

As a second example, consider a function to emulate directly the MATLAB backslash command, which returns the coefficients of the orthogonal projection of the vector y onto the column space of the matrix, X. (This is ordinarily called the least squares estimate of the regression coefficients.) This would ordinarily be done with the `qr()` function; however this is sometimes a bit tricky to use directly and it pays to have a simple function such as the following to use it safely.

Thus given a n by 1 vector y and an n by p matrix X then $X\backslash y$ is defined as $(X^TX)^-X^Ty$, where $(X^TX)^-$ is a generalized inverse of $X'X$.

```
> bslash <- function(X, y) {
  X <- qr(X)
  qr.coef(X, y)
}
```

After this object is created it may be used in statements such as

```
> regcoeff <- bslash(Xmat, yvar)
```

and so on.

The classical R function `lsfit()` does this job quite well, and more[1]. It in turn uses the functions `qr()` and `qr.coef()` in the slightly counterintuitive way above to do this part of the calculation. Hence there is probably some value in having just this part isolated in a simple to use function if it is going to be in frequent use. If so, we may wish to make it a matrix binary operator for even more convenient use.

10.2 Defining new binary operators

Had we given the `bslash()` function a different name, namely one of the form

%anything%

it could have been used as a *binary operator* in expressions rather than in function form. Suppose, for example, we choose `!` for the internal character. The function definition would then start as

```
> "%!%" <- function(X, y) { ... }
```

(Note the use of quote marks.) The function could then be used as `X %!% y`. (The backslash symbol itself is not a convenient choice as it presents special problems in this context.)

The matrix multiplication operator, `%*%`, and the outer product matrix operator `%o%` are other examples of binary operators defined in this way.

10.3 Named arguments and defaults

As first noted in Section 2.3 [Generating regular sequences], page 8, if arguments to called functions are given in the "*name=object*" form, they may be given in any order. Furthermore the argument sequence may begin in the unnamed, positional form, and specify named arguments after the positional arguments.

Thus if there is a function `fun1` defined by

```
> fun1 <- function(data, data.frame, graph, limit) {
    [function body omitted]
  }
```

then the function may be invoked in several ways, for example

```
> ans <- fun1(d, df, TRUE, 20)
> ans <- fun1(d, df, graph=TRUE, limit=20)
> ans <- fun1(data=d, limit=20, graph=TRUE, data.frame=df)
```

are all equivalent.

In many cases arguments can be given commonly appropriate default values, in which case they may be omitted altogether from the call when the defaults are appropriate. For example, if `fun1` were defined as

```
> fun1 <- function(data, data.frame, graph=TRUE, limit=20) { ... }
```

it could be called as

```
> ans <- fun1(d, df)
```

which is now equivalent to the three cases above, or as

```
> ans <- fun1(d, df, limit=10)
```

which changes one of the defaults.

It is important to note that defaults may be arbitrary expressions, even involving other arguments to the same function; they are not restricted to be constants as in our simple example here.

[1] See also the methods described in Chapter 11 [Statistical models in R], page 50

10.4 The '...' argument

Another frequent requirement is to allow one function to pass on argument settings to another. For example many graphics functions use the function `par()` and functions like `plot()` allow the user to pass on graphical parameters to `par()` to control the graphical output. (See Section 12.4.1 [The par() function], page 67, for more details on the `par()` function.) This can be done by including an extra argument, literally '`...`', of the function, which may then be passed on. An outline example is given below.

```
fun1 <- function(data, data.frame, graph=TRUE, limit=20, ...) {
  [omitted statements]
  if (graph)
    par(pch="*", ...)
  [more omissions]
}
```

Less frequently, a function will need to refer to components of '`...`'. The expression `list(...)` evaluates all such arguments and returns them in a named list, while `..1`, `..2`, etc. evaluate them one at a time, with '`..n`' returning the n-th unmatched argument.

10.5 Assignments within functions

Note that *any ordinary assignments done within the function are local and temporary and are lost after exit from the function.* Thus the assignment `X <- qr(X)` does not affect the value of the argument in the calling program.

To understand completely the rules governing the scope of R assignments the reader needs to be familiar with the notion of an evaluation *frame*. This is a somewhat advanced, though hardly difficult, topic and is not covered further here.

If global and permanent assignments are intended within a function, then either the 'superassignment' operator, `<<-` or the function `assign()` can be used. See the `help` document for details.

10.6 More advanced examples

10.6.1 Efficiency factors in block designs

As a more complete, if a little pedestrian, example of a function, consider finding the efficiency factors for a block design. (Some aspects of this problem have already been discussed in Section 5.3 [Index matrices], page 19.)

A block design is defined by two factors, say `blocks` (`b` levels) and `varieties` (`v` levels). If R and K are the v by v and b by b *replications* and *block size* matrices, respectively, and N is the b by v incidence matrix, then the efficiency factors are defined as the eigenvalues of the matrix

$$E = I_v - R^{-1/2} N^T K^{-1} N R^{-1/2} = I_v - A^T A,$$

where $A = K^{-1/2} N R^{-1/2}$. One way to write the function is given below.

```
> bdeff <- function(blocks, varieties) {
    blocks <- as.factor(blocks)             # minor safety move
    b <- length(levels(blocks))
    varieties <- as.factor(varieties)       # minor safety move
    v <- length(levels(varieties))
    K <- as.vector(table(blocks))           # remove dim attr
    R <- as.vector(table(varieties))        # remove dim attr
    N <- table(blocks, varieties)
```

```
    A <- 1/sqrt(K) * N * rep(1/sqrt(R), rep(b, v))
    sv <- svd(A)
    list(eff=1 - sv$d^2, blockcv=sv$u, varietycv=sv$v)
}
```

It is numerically slightly better to work with the singular value decomposition on this occasion rather than the eigenvalue routines.

The result of the function is a list giving not only the efficiency factors as the first component, but also the block and variety canonical contrasts, since sometimes these give additional useful qualitative information.

10.6.2 Dropping all names in a printed array

For printing purposes with large matrices or arrays, it is often useful to print them in close block form without the array names or numbers. Removing the `dimnames` attribute will not achieve this effect, but rather the array must be given a `dimnames` attribute consisting of empty strings. For example to print a matrix, `X`

```
> temp <- X
> dimnames(temp) <- list(rep("", nrow(X)), rep("", ncol(X)))
> temp; rm(temp)
```

This can be much more conveniently done using a function, `no.dimnames()`, shown below, as a "wrap around" to achieve the same result. It also illustrates how some effective and useful user functions can be quite short.

```
no.dimnames <- function(a) {
  ## Remove all dimension names from an array for compact printing.
  d <- list()
  l <- 0
  for(i in dim(a)) {
    d[[l <- l + 1]] <- rep("", i)
  }
  dimnames(a) <- d
  a
}
```

With this function defined, an array may be printed in close format using

```
> no.dimnames(X)
```

This is particularly useful for large integer arrays, where patterns are the real interest rather than the values.

10.6.3 Recursive numerical integration

Functions may be recursive, and may themselves define functions within themselves. Note, however, that such functions, or indeed variables, are not inherited by called functions in higher evaluation frames as they would be if they were on the search path.

The example below shows a naive way of performing one-dimensional numerical integration. The integrand is evaluated at the end points of the range and in the middle. If the one-panel trapezium rule answer is close enough to the two panel, then the latter is returned as the value. Otherwise the same process is recursively applied to each panel. The result is an adaptive integration process that concentrates function evaluations in regions where the integrand is farthest from linear. There is, however, a heavy overhead, and the function is only competitive with other algorithms when the integrand is both smooth and very difficult to evaluate.

The example is also given partly as a little puzzle in R programming.

```
area <- function(f, a, b, eps = 1.0e-06, lim = 10) {
```

```
  fun1 <- function(f, a, b, fa, fb, a0, eps, lim, fun) {
    ## function 'fun1' is only visible inside 'area'
    d <- (a + b)/2
    h <- (b - a)/4
    fd <- f(d)
    a1 <- h * (fa + fd)
    a2 <- h * (fd + fb)
    if(abs(a0 - a1 - a2) < eps || lim == 0)
      return(a1 + a2)
    else {
      return(fun(f, a, d, fa, fd, a1, eps, lim - 1, fun) +
             fun(f, d, b, fd, fb, a2, eps, lim - 1, fun))
    }
  }
  fa <- f(a)
  fb <- f(b)
  a0 <- ((fa + fb) * (b - a))/2
  fun1(f, a, b, fa, fb, a0, eps, lim, fun1)
}
```

10.7 Scope

The discussion in this section is somewhat more technical than in other parts of this document. However, it details one of the major differences between S-PLUS and R.

The symbols which occur in the body of a function can be divided into three classes; formal parameters, local variables and free variables. The formal parameters of a function are those occurring in the argument list of the function. Their values are determined by the process of *binding* the actual function arguments to the formal parameters. Local variables are those whose values are determined by the evaluation of expressions in the body of the functions. Variables which are not formal parameters or local variables are called free variables. Free variables become local variables if they are assigned to. Consider the following function definition.

```
f <- function(x) {
  y <- 2*x
  print(x)
  print(y)
  print(z)
}
```

In this function, `x` is a formal parameter, `y` is a local variable and `z` is a free variable.

In R the free variable bindings are resolved by first looking in the environment in which the function was created. This is called *lexical scope*. First we define a function called `cube`.

```
cube <- function(n) {
  sq <- function() n*n
  n*sq()
}
```

The variable `n` in the function `sq` is not an argument to that function. Therefore it is a free variable and the scoping rules must be used to ascertain the value that is to be associated with it. Under static scope (S-PLUS) the value is that associated with a global variable named `n`. Under lexical scope (R) it is the parameter to the function `cube` since that is the active binding for the variable `n` at the time the function `sq` was defined. The difference between evaluation in R and evaluation in S-PLUS is that S-PLUS looks for a global variable called `n` while R first looks for a variable called `n` in the environment created when `cube` was invoked.

```
## first evaluation in S
S> cube(2)
Error in sq(): Object "n" not found
Dumped
S> n <- 3
S> cube(2)
[1] 18
## then the same function evaluated in R
R> cube(2)
[1] 8
```

Lexical scope can also be used to give functions *mutable state*. In the following example we show how R can be used to mimic a bank account. A functioning bank account needs to have a balance or total, a function for making withdrawals, a function for making deposits and a function for stating the current balance. We achieve this by creating the three functions within `account` and then returning a list containing them. When `account` is invoked it takes a numerical argument `total` and returns a list containing the three functions. Because these functions are defined in an environment which contains `total`, they will have access to its value.

The special assignment operator, `<<-`, is used to change the value associated with `total`. This operator looks back in enclosing environments for an environment that contains the symbol `total` and when it finds such an environment it replaces the value, in that environment, with the value of right hand side. If the global or top-level environment is reached without finding the symbol `total` then that variable is created and assigned to there. For most users `<<-` creates a global variable and assigns the value of the right hand side to it[2]. Only when `<<-` has been used in a function that was returned as the value of another function will the special behavior described here occur.

```
open.account <- function(total) {
  list(
    deposit = function(amount) {
      if(amount <= 0)
        stop("Deposits must be positive!\n")
      total <<- total + amount
      cat(amount, "deposited.  Your balance is", total, "\n\n")
    },
    withdraw = function(amount) {
      if(amount > total)
        stop("You don't have that much money!\n")
      total <<- total - amount
      cat(amount, "withdrawn.  Your balance is", total, "\n\n")
    },
    balance = function() {
      cat("Your balance is", total, "\n\n")
    }
  )
}

ross <- open.account(100)
robert <- open.account(200)

ross$withdraw(30)
```

[2] In some sense this mimics the behavior in S-Plus since in S-Plus this operator always creates or assigns to a global variable.

```
ross$balance()
robert$balance()

ross$deposit(50)
ross$balance()
ross$withdraw(500)
```

10.8 Customizing the environment

Users can customize their environment in several different ways. There is a site initialization file and every directory can have its own special initialization file. Finally, the special functions `.First` and `.Last` can be used.

The location of the site initialization file is taken from the value of the `R_PROFILE` environment variable. If that variable is unset, the file `Rprofile.site` in the R home subdirectory `etc` is used. This file should contain the commands that you want to execute every time R is started under your system. A second, personal, profile file named `.Rprofile`[3] can be placed in any directory. If R is invoked in that directory then that file will be sourced. This file gives individual users control over their workspace and allows for different startup procedures in different working directories. If no `.Rprofile` file is found in the startup directory, then R looks for a `.Rprofile` file in the user's home directory and uses that (if it exists). If the environment variable `R_PROFILE_USER` is set, the file it points to is used instead of the `.Rprofile` files.

Any function named `.First()` in either of the two profile files or in the `.RData` image has a special status. It is automatically performed at the beginning of an R session and may be used to initialize the environment. For example, the definition in the example below alters the prompt to `$` and sets up various other useful things that can then be taken for granted in the rest of the session.

Thus, the sequence in which files are executed is, `Rprofile.site`, the user profile, `.RData` and then `.First()`. A definition in later files will mask definitions in earlier files.

```
> .First <- function() {
  options(prompt="$ ", continue="+\t")  # $ is the prompt
  options(digits=5, length=999)         # custom numbers and printout
  x11()                                 # for graphics
  par(pch = "+")                        # plotting character
  source(file.path(Sys.getenv("HOME"), "R", "mystuff.R"))
                                        # my personal functions
  library(MASS)                         # attach a package
}
```

Similarly a function `.Last()`, if defined, is (normally) executed at the very end of the session. An example is given below.

```
> .Last <- function() {
  graphics.off()                        # a small safety measure.
  cat(paste(date(),"\nAdios\n"))        # Is it time for lunch?
}
```

10.9 Classes, generic functions and object orientation

The class of an object determines how it will be treated by what are known as *generic* functions. Put the other way round, a generic function performs a task or action on its arguments *specific to the class of the argument itself.* If the argument lacks any `class` attribute, or has a class

[3] So it is hidden under UNIX.

not catered for specifically by the generic function in question, there is always a *default action* provided.

An example makes things clearer. The class mechanism offers the user the facility of designing and writing generic functions for special purposes. Among the other generic functions are `plot()` for displaying objects graphically, `summary()` for summarizing analyses of various types, and `anova()` for comparing statistical models.

The number of generic functions that can treat a class in a specific way can be quite large. For example, the functions that can accommodate in some fashion objects of class `"data.frame"` include

```
[      [[<-     any     as.matrix
[<-    mean     plot    summary
```

A currently complete list can be got by using the `methods()` function:

```
> methods(class="data.frame")
```

Conversely the number of classes a generic function can handle can also be quite large. For example the `plot()` function has a default method and variants for objects of classes `"data.frame"`, `"density"`, `"factor"`, and more. A complete list can be got again by using the `methods()` function:

```
> methods(plot)
```

For many generic functions the function body is quite short, for example

```
> coef
function (object, ...)
UseMethod("coef")
```

The presence of `UseMethod` indicates this is a generic function. To see what methods are available we can use `methods()`

```
> methods(coef)
[1] coef.aov*          coef.Arima*        coef.default*      coef.listof*
[5] coef.nls*          coef.summary.nls*

   Non-visible functions are asterisked
```

In this example there are six methods, none of which can be seen by typing its name. We can read these by either of

```
> getAnywhere("coef.aov")
A single object matching 'coef.aov' was found
It was found in the following places
  registered S3 method for coef from namespace stats
  namespace:stats
with value

function (object, ...)
{
    z <- object$coef
    z[!is.na(z)]
}

> getS3method("coef", "aov")
function (object, ...)
{
    z <- object$coef
    z[!is.na(z)]
```

```
}
```

A function named *`gen.cl`* will be invoked by the generic *`gen`* for class *`cl`*, so do not name functions in this style unless they are intended to be methods.

The reader is referred to the *R Language Definition* for a more complete discussion of this mechanism.

11 Statistical models in R

This section presumes the reader has some familiarity with statistical methodology, in particular with regression analysis and the analysis of variance. Later we make some rather more ambitious presumptions, namely that something is known about generalized linear models and nonlinear regression.

The requirements for fitting statistical models are sufficiently well defined to make it possible to construct general tools that apply in a broad spectrum of problems.

R provides an interlocking suite of facilities that make fitting statistical models very simple. As we mention in the introduction, the basic output is minimal, and one needs to ask for the details by calling extractor functions.

11.1 Defining statistical models; formulae

The template for a statistical model is a linear regression model with independent, homoscedastic errors

$$y_i = \sum_{j=0}^{p} \beta_j x_{ij} + e_i, \qquad e_i \sim \text{NID}(0, \sigma^2), \qquad i = 1, \ldots, n$$

In matrix terms this would be written

$$y = X\beta + e$$

where the y is the response vector, X is the *model matrix* or *design matrix* and has columns $x_0, x_1, \ldots, x_p$, the determining variables. Very often x_0 will be a column of ones defining an *intercept* term.

Examples

Before giving a formal specification, a few examples may usefully set the picture.

Suppose `y`, `x`, `x0`, `x1`, `x2`, ... are numeric variables, `X` is a matrix and `A`, `B`, `C`, ... are factors. The following formulae on the left side below specify statistical models as described on the right.

`y ~ x`
`y ~ 1 + x` Both imply the same simple linear regression model of y on x. The first has an implicit intercept term, and the second an explicit one.

`y ~ 0 + x`
`y ~ -1 + x`
`y ~ x - 1` Simple linear regression of y on x through the origin (that is, without an intercept term).

`log(y) ~ x1 + x2`
Multiple regression of the transformed variable,$\log(y)$, on $x1$ and $x2$ (with an implicit intercept term).

`y ~ poly(x,2)`
`y ~ 1 + x + I(x^2)`
Polynomial regression of y on x of degree 2. The first form uses orthogonal polynomials, and the second uses explicit powers, as basis.

`y ~ X + poly(x,2)`
Multiple regression y with model matrix consisting of the matrix X as well as polynomial terms in x to degree 2.

`y ~ A` Single classification analysis of variance model of y, with classes determined by A.

`y ~ A + x` Single classification analysis of covariance model of y, with classes determined by A, and with covariate x.

```
y ~ A*B
y ~ A + B + A:B
y ~ B %in% A
```
`y ~ A/B` Two factor non-additive model of y on A and B. The first two specify the same crossed classification and the second two specify the same nested classification. In abstract terms all four specify the same model subspace.

```
y ~ (A + B + C)^2
y ~ A*B*C - A:B:C
```
Three factor experiment but with a model containing main effects and two factor interactions only. Both formulae specify the same model.

```
y ~ A * x
y ~ A/x
y ~ A/(1 + x) - 1
```
Separate simple linear regression models of y on x within the levels of A, with different codings. The last form produces explicit estimates of as many different intercepts and slopes as there are levels in A.

```
y ~ A*B + Error(C)
```
An experiment with two treatment factors, A and B, and error strata determined by factor C. For example a split plot experiment, with whole plots (and hence also subplots), determined by factor C.

The operator `~` is used to define a *model formula* in R. The form, for an ordinary linear model, is

```
response ~ op_1 term_1 op_2 term_2 op_3 term_3 ...
```

where

response is a vector or matrix, (or expression evaluating to a vector or matrix) defining the response variable(s).

op_i is an operator, either `+` or `-`, implying the inclusion or exclusion of a term in the model, (the first is optional).

term_i is either

- a vector or matrix expression, or `1`,
- a factor, or
- a *formula expression* consisting of factors, vectors or matrices connected by *formula operators*.

In all cases each term defines a collection of columns either to be added to or removed from the model matrix. A `1` stands for an intercept column and is by default included in the model matrix unless explicitly removed.

The *formula operators* are similar in effect to the Wilkinson and Rogers notation used by such programs as Glim and Genstat. One inevitable change is that the operator '`.`' becomes '`:`' since the period is a valid name character in R.

The notation is summarized below (based on Chambers & Hastie, 1992, p.29):

`Y ~ M` Y is modeled as M.

`M_1 + M_2` Include *M_1* and *M_2*.

M_1 - *M_2* Include *M_1* leaving out terms of *M_2*.

M_1 : *M_2* The tensor product of *M_1* and *M_2*. If both terms are factors, then the "subclasses" factor.

M_1 %in% *M_2*
Similar to *M_1*:*M_2*, but with a different coding.

M_1 * *M_2* *M_1* + *M_2* + *M_1*:*M_2*.

M_1 / *M_2* *M_1* + *M_2* %in% *M_1*.

M^n All terms in M together with "interactions" up to order n

I(M) Insulate M. Inside M all operators have their normal arithmetic meaning, and that term appears in the model matrix.

Note that inside the parentheses that usually enclose function arguments all operators have their normal arithmetic meaning. The function `I()` is an identity function used to allow terms in model formulae to be defined using arithmetic operators.

Note particularly that the model formulae specify the *columns of the model matrix*, the specification of the parameters being implicit. This is not the case in other contexts, for example in specifying nonlinear models.

11.1.1 Contrasts

We need at least some idea how the model formulae specify the columns of the model matrix. This is easy if we have continuous variables, as each provides one column of the model matrix (and the intercept will provide a column of ones if included in the model).

What about a k-level factor `A`? The answer differs for unordered and ordered factors. For *unordered* factors $k-1$ columns are generated for the indicators of the second, ..., k-th levels of the factor. (Thus the implicit parameterization is to contrast the response at each level with that at the first.) For *ordered* factors the $k-1$ columns are the orthogonal polynomials on $1, \ldots, k$, omitting the constant term.

Although the answer is already complicated, it is not the whole story. First, if the intercept is omitted in a model that contains a factor term, the first such term is encoded into k columns giving the indicators for all the levels. Second, the whole behavior can be changed by the `options` setting for `contrasts`. The default setting in R is

```
options(contrasts = c("contr.treatment", "contr.poly"))
```

The main reason for mentioning this is that R and S have different defaults for unordered factors, S using Helmert contrasts. So if you need to compare your results to those of a textbook or paper which used S-PLUS, you will need to set

```
options(contrasts = c("contr.helmert", "contr.poly"))
```

This is a deliberate difference, as treatment contrasts (R's default) are thought easier for newcomers to interpret.

We have still not finished, as the contrast scheme to be used can be set for each term in the model using the functions `contrasts` and `C`.

We have not yet considered interaction terms: these generate the products of the columns introduced for their component terms.

Although the details are complicated, model formulae in R will normally generate the models that an expert statistician would expect, provided that marginality is preserved. Fitting, for example, a model with an interaction but not the corresponding main effects will in general lead to surprising results, and is for experts only.

11.2 Linear models

The basic function for fitting ordinary multiple models is `lm()`, and a streamlined version of the call is as follows:

```
> fitted.model <- lm(formula, data = data.frame)
```

For example

```
> fm2 <- lm(y ~ x1 + x2, data = production)
```

would fit a multiple regression model of y on $x1$ and $x2$ (with implicit intercept term).

The important (but technically optional) parameter `data = production` specifies that any variables needed to construct the model should come first from the `production` *data frame. This is the case regardless of whether data frame* `production` *has been attached on the search path or not.*

11.3 Generic functions for extracting model information

The value of `lm()` is a fitted model object; technically a list of results of class `"lm"`. Information about the fitted model can then be displayed, extracted, plotted and so on by using generic functions that orient themselves to objects of class `"lm"`. These include

```
add1     deviance   formula      predict  step
alias    drop1      kappa        print    summary
anova    effects    labels       proj     vcov
coef     family     plot         residuals
```

A brief description of the most commonly used ones is given below.

`anova(`*object_1*`, `*object_2*`)`
: Compare a submodel with an outer model and produce an analysis of variance table.

`coef(`*object*`)`
: Extract the regression coefficient (matrix).

  Long form: `coefficients(`*object*`)`.

`deviance(`*object*`)`
: Residual sum of squares, weighted if appropriate.

`formula(`*object*`)`
: Extract the model formula.

`plot(`*object*`)`
: Produce four plots, showing residuals, fitted values and some diagnostics.

`predict(`*object*`, newdata=`*data.frame*`)`
: The data frame supplied must have variables specified with the same labels as the original. The value is a vector or matrix of predicted values corresponding to the determining variable values in *data.frame*.

`print(`*object*`)`
: Print a concise version of the object. Most often used implicitly.

`residuals(`*object*`)`
: Extract the (matrix of) residuals, weighted as appropriate.

  Short form: `resid(`*object*`)`.

`step(`*object*`)`
: Select a suitable model by adding or dropping terms and preserving hierarchies. The model with the smallest value of AIC (Akaike's An Information Criterion) discovered in the stepwise search is returned.

`summary(object)`
: Print a comprehensive summary of the results of the regression analysis.

`vcov(object)`
: Returns the variance-covariance matrix of the main parameters of a fitted model object.

11.4 Analysis of variance and model comparison

The model fitting function `aov(formula, data=data.frame)` operates at the simplest level in a very similar way to the function `lm()`, and most of the generic functions listed in the table in Section 11.3 [Generic functions for extracting model information], page 53, apply.

It should be noted that in addition `aov()` allows an analysis of models with multiple error strata such as split plot experiments, or balanced incomplete block designs with recovery of inter-block information. The model formula

```
response ~ mean.formula + Error(strata.formula)
```

specifies a multi-stratum experiment with error strata defined by the *strata.formula*. In the simplest case, *strata.formula* is simply a factor, when it defines a two strata experiment, namely between and within the levels of the factor.

For example, with all determining variables factors, a model formula such as that in:

```
> fm <- aov(yield ~ v + n*p*k + Error(farms/blocks), data=farm.data)
```

would typically be used to describe an experiment with mean model `v + n*p*k` and three error strata, namely "between farms", "within farms, between blocks" and "within blocks".

11.4.1 ANOVA tables

Note also that the analysis of variance table (or tables) are for a sequence of fitted models. The sums of squares shown are the decrease in the residual sums of squares resulting from an inclusion of *that term* in the model at *that place* in the sequence. Hence only for orthogonal experiments will the order of inclusion be inconsequential.

For multistratum experiments the procedure is first to project the response onto the error strata, again in sequence, and to fit the mean model to each projection. For further details, see Chambers & Hastie (1992).

A more flexible alternative to the default full ANOVA table is to compare two or more models directly using the `anova()` function.

```
> anova(fitted.model.1, fitted.model.2, ...)
```

The display is then an ANOVA table showing the differences between the fitted models when fitted in sequence. The fitted models being compared would usually be an hierarchical sequence, of course. This does not give different information to the default, but rather makes it easier to comprehend and control.

11.5 Updating fitted models

The `update()` function is largely a convenience function that allows a model to be fitted that differs from one previously fitted usually by just a few additional or removed terms. Its form is

```
> new.model <- update(old.model, new.formula)
```

In the *new.formula* the special name consisting of a period, '`.`', only, can be used to stand for "the corresponding part of the old model formula". For example,

```
> fm05 <- lm(y ~ x1 + x2 + x3 + x4 + x5, data = production)
> fm6  <- update(fm05, . ~ . + x6)
> smf6 <- update(fm6, sqrt(.) ~ .)
```

would fit a five variate multiple regression with variables (presumably) from the data frame `production`, fit an additional model including a sixth regressor variable, and fit a variant on the model where the response had a square root transform applied.

Note especially that if the `data=` argument is specified on the original call to the model fitting function, this information is passed on through the fitted model object to `update()` and its allies.

The name '`.`' can also be used in other contexts, but with slightly different meaning. For example

```
> fmfull <- lm(y ~ . , data = production)
```

would fit a model with response `y` and regressor variables *all other variables in the data frame* `production`.

Other functions for exploring incremental sequences of models are `add1()`, `drop1()` and `step()`. The names of these give a good clue to their purpose, but for full details see the on-line help.

11.6 Generalized linear models

Generalized linear modeling is a development of linear models to accommodate both non-normal response distributions and transformations to linearity in a clean and straightforward way. A generalized linear model may be described in terms of the following sequence of assumptions:

- There is a response, y, of interest and stimulus variables x_1, x_2, ..., whose values influence the distribution of the response.
- The stimulus variables influence the distribution of y through *a single linear function, only.* This linear function is called the *linear predictor*, and is usually written

  $$\eta = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p,$$

  hence x_i has no influence on the distribution of y if and only if $\beta_i = 0$.
- The distribution of y is of the form

  $$f_Y(y; \mu, \varphi) = \exp\left[\frac{A}{\varphi}\left\{y\lambda(\mu) - \gamma\left(\lambda(\mu)\right)\right\} + \tau(y, \varphi)\right]$$

  where φ is a *scale parameter* (possibly known), and is constant for all observations, A represents a prior weight, assumed known but possibly varying with the observations, and μ is the mean of y.

  So it is assumed that the distribution of y is determined by its mean and possibly a scale parameter as well.
- The mean, μ, is a smooth invertible function of the linear predictor:

  $$\mu = m(\eta), \qquad \eta = m^{-1}(\mu) = \ell(\mu)$$

  and this inverse function, $\ell()$, is called the *link function.*

These assumptions are loose enough to encompass a wide class of models useful in statistical practice, but tight enough to allow the development of a unified methodology of estimation and inference, at least approximately. The reader is referred to any of the current reference works on the subject for full details, such as McCullagh & Nelder (1989) or Dobson (1990).

11.6.1 Families

The class of generalized linear models handled by facilities supplied in R includes *gaussian*, *binomial*, *poisson*, *inverse gaussian* and *gamma* response distributions and also *quasi-likelihood* models where the response distribution is not explicitly specified. In the latter case the *variance function* must be specified as a function of the mean, but in other cases this function is implied by the response distribution.

Each response distribution admits a variety of link functions to connect the mean with the linear predictor. Those automatically available are shown in the following table:

Family name	**Link functions**
`binomial`	`logit`, `probit`, `log`, `cloglog`
`gaussian`	`identity`, `log`, `inverse`
`Gamma`	`identity`, `inverse`, `log`
`inverse.gaussian`	`1/mu^2`, `identity`, `inverse`, `log`
`poisson`	`identity`, `log`, `sqrt`
`quasi`	`logit`, `probit`, `cloglog`, `identity`, `inverse`, `log`, `1/mu^2`, `sqrt`

The combination of a response distribution, a link function and various other pieces of information that are needed to carry out the modeling exercise is called the *family* of the generalized linear model.

11.6.2 The `glm()` function

Since the distribution of the response depends on the stimulus variables through a single linear function *only*, the same mechanism as was used for linear models can still be used to specify the linear part of a generalized model. The family has to be specified in a different way.

The R function to fit a generalized linear model is `glm()` which uses the form

```
> fitted.model <- glm(formula, family=family.generator, data=data.frame)
```

The only new feature is the *family.generator*, which is the instrument by which the family is described. It is the name of a function that generates a list of functions and expressions that together define and control the model and estimation process. Although this may seem a little complicated at first sight, its use is quite simple.

The names of the standard, supplied family generators are given under "Family Name" in the table in Section 11.6.1 [Families], page 56. Where there is a choice of links, the name of the link may also be supplied with the family name, in parentheses as a parameter. In the case of the `quasi` family, the variance function may also be specified in this way.

Some examples make the process clear.

The `gaussian` family

A call such as

```
> fm <- glm(y ~ x1 + x2, family = gaussian, data = sales)
```

achieves the same result as

```
> fm <- lm(y ~ x1+x2, data=sales)
```

but much less efficiently. Note how the gaussian family is not automatically provided with a choice of links, so no parameter is allowed. If a problem requires a gaussian family with a nonstandard link, this can usually be achieved through the `quasi` family, as we shall see later.

The `binomial` family

Consider a small, artificial example, from Silvey (1970).

On the Aegean island of Kalythos the male inhabitants suffer from a congenital eye disease, the effects of which become more marked with increasing age. Samples of islander males of various ages were tested for blindness and the results recorded. The data is shown below:

Age:	20	35	45	55	70
No. tested:	50	50	50	50	50
No. blind:	6	17	26	37	44

The problem we consider is to fit both logistic and probit models to this data, and to estimate for each model the LD50, that is the age at which the chance of blindness for a male inhabitant is 50%.

If y is the number of blind at age x and n the number tested, both models have the form

$$y \sim \mathrm{B}(n, F(\beta_0 + \beta_1 x))$$

where for the probit case, $F(z) = \Phi(z)$ is the standard normal distribution function, and in the logit case (the default), $F(z) = e^z/(1 + e^z)$. In both cases the LD50 is

$$\mathrm{LD50} = -\beta_0/\beta_1$$

that is, the point at which the argument of the distribution function is zero.

The first step is to set the data up as a data frame

```
> kalythos <- data.frame(x = c(20,35,45,55,70), n = rep(50,5),
                         y = c(6,17,26,37,44))
```

To fit a binomial model using `glm()` there are three possibilities for the response:

- If the response is a *vector* it is assumed to hold *binary* data, and so must be a 0/1 vector.
- If the response is a *two-column matrix* it is assumed that the first column holds the number of successes for the trial and the second holds the number of failures.
- If the response is a *factor*, its first level is taken as failure (0) and all other levels as 'success' (1).

Here we need the second of these conventions, so we add a matrix to our data frame:

```
> kalythos$Ymat <- cbind(kalythos$y, kalythos$n - kalythos$y)
```

To fit the models we use

```
> fmp <- glm(Ymat ~ x, family = binomial(link=probit), data = kalythos)
> fml <- glm(Ymat ~ x, family = binomial, data = kalythos)
```

Since the logit link is the default the parameter may be omitted on the second call. To see the results of each fit we could use

```
> summary(fmp)
> summary(fml)
```

Both models fit (all too) well. To find the LD50 estimate we can use a simple function:

```
> ld50 <- function(b) -b[1]/b[2]
> ldp <- ld50(coef(fmp)); ldl <- ld50(coef(fml)); c(ldp, ldl)
```

The actual estimates from this data are 43.663 years and 43.601 years respectively.

Poisson models

With the Poisson family the default link is the `log`, and in practice the major use of this family is to fit surrogate Poisson log-linear models to frequency data, whose actual distribution is often multinomial. This is a large and important subject we will not discuss further here. It even forms a major part of the use of non-gaussian generalized models overall.

Occasionally genuinely Poisson data arises in practice and in the past it was often analyzed as gaussian data after either a log or a square-root transformation. As a graceful alternative to the latter, a Poisson generalized linear model may be fitted as in the following example:

```
> fmod <- glm(y ~ A + B + x, family = poisson(link=sqrt),
              data = worm.counts)
```

Quasi-likelihood models

For all families the variance of the response will depend on the mean and will have the scale parameter as a multiplier. The form of dependence of the variance on the mean is a characteristic of the response distribution; for example for the Poisson distribution $\mathrm{Var}[y] = \mu$.

For quasi-likelihood estimation and inference the precise response distribution is not specified, but rather only a link function and the form of the variance function as it depends on the mean. Since quasi-likelihood estimation uses formally identical techniques to those for the gaussian distribution, this family provides a way of fitting gaussian models with non-standard link functions or variance functions, incidentally.

For example, consider fitting the non-linear regression

$$y = \frac{\theta_1 z_1}{z_2 - \theta_2} + e$$

which may be written alternatively as

$$y = \frac{1}{\beta_1 x_1 + \beta_2 x_2} + e$$

where $x_1 = z_2/z_1$, $x_2 = -1/z_1$, $\beta_1 = 1/\theta_1$ and $\beta_2 = \theta_2/\theta_1$. Supposing a suitable data frame to be set up we could fit this non-linear regression as

```
> nlfit <- glm(y ~ x1 + x2 - 1,
               family = quasi(link=inverse, variance=constant),
               data = biochem)
```

The reader is referred to the manual and the help document for further information, as needed.

11.7 Nonlinear least squares and maximum likelihood models

Certain forms of nonlinear model can be fitted by Generalized Linear Models (`glm()`). But in the majority of cases we have to approach the nonlinear curve fitting problem as one of nonlinear optimization. R's nonlinear optimization routines are `optim()`, `nlm()` and `nlminb()`, We seek the parameter values that minimize some index of lack-of-fit, and they do this by trying out various parameter values iteratively. Unlike linear regression for example, there is no guarantee that the procedure will converge on satisfactory estimates. All the methods require initial guesses about what parameter values to try, and convergence may depend critically upon the quality of the starting values.

11.7.1 Least squares

One way to fit a nonlinear model is by minimizing the sum of the squared errors (SSE) or residuals. This method makes sense if the observed errors could have plausibly arisen from a normal distribution.

Here is an example from Bates & Watts (1988), page 51. The data are:

```
> x <- c(0.02, 0.02, 0.06, 0.06, 0.11, 0.11, 0.22, 0.22, 0.56, 0.56,
         1.10, 1.10)
```

```
> y <- c(76, 47, 97, 107, 123, 139, 159, 152, 191, 201, 207, 200)
```

The fit criterion to be minimized is:

```
> fn <- function(p) sum((y - (p[1] * x)/(p[2] + x))^2)
```

In order to do the fit we need initial estimates of the parameters. One way to find sensible starting values is to plot the data, guess some parameter values, and superimpose the model curve using those values.

```
> plot(x, y)
> xfit <- seq(.02, 1.1, .05)
> yfit <- 200 * xfit/(0.1 + xfit)
> lines(spline(xfit, yfit))
```

We could do better, but these starting values of 200 and 0.1 seem adequate. Now do the fit:

```
> out <- nlm(fn, p = c(200, 0.1), hessian = TRUE)
```

After the fitting, `out$minimum` is the SSE, and `out$estimate` are the least squares estimates of the parameters. To obtain the approximate standard errors (SE) of the estimates we do:

```
> sqrt(diag(2*out$minimum/(length(y) - 2) * solve(out$hessian)))
```

The 2 which is subtracted in the line above represents the number of parameters. A 95% confidence interval would be the parameter estimate $\pm$ 1.96 SE. We can superimpose the least squares fit on a new plot:

```
> plot(x, y)
> xfit <- seq(.02, 1.1, .05)
> yfit <- 212.68384222 * xfit/(0.06412146 + xfit)
> lines(spline(xfit, yfit))
```

The standard package **stats** provides much more extensive facilities for fitting non-linear models by least squares. The model we have just fitted is the Michaelis-Menten model, so we can use

```
> df <- data.frame(x=x, y=y)
> fit <- nls(y ~ SSmicmen(x, Vm, K), df)
> fit
Nonlinear regression model
  model:  y ~ SSmicmen(x, Vm, K)
   data:  df
         Vm            K
212.68370711   0.06412123
 residual sum-of-squares:  1195.449
> summary(fit)

Formula: y ~ SSmicmen(x, Vm, K)

Parameters:
    Estimate Std. Error t value Pr(>|t|)
Vm 2.127e+02  6.947e+00  30.615 3.24e-11
K  6.412e-02  8.281e-03   7.743 1.57e-05

Residual standard error: 10.93 on 10 degrees of freedom

Correlation of Parameter Estimates:
      Vm
K 0.7651
```

11.7.2 Maximum likelihood

Maximum likelihood is a method of nonlinear model fitting that applies even if the errors are not normal. The method finds the parameter values which maximize the log likelihood, or equivalently which minimize the negative log-likelihood. Here is an example from Dobson (1990), pp. 108–111. This example fits a logistic model to dose-response data, which clearly could also be fit by `glm()`. The data are:

```
> x <- c(1.6907, 1.7242, 1.7552, 1.7842, 1.8113,
         1.8369, 1.8610, 1.8839)
> y <- c( 6, 13, 18, 28, 52, 53, 61, 60)
> n <- c(59, 60, 62, 56, 63, 59, 62, 60)
```

The negative log-likelihood to minimize is:

```
> fn <- function(p)
   sum( - (y*(p[1]+p[2]*x) - n*log(1+exp(p[1]+p[2]*x))
           + log(choose(n, y)) ))
```

We pick sensible starting values and do the fit:

```
> out <- nlm(fn, p = c(-50,20), hessian = TRUE)
```

After the fitting, `out$minimum` is the negative log-likelihood, and `out$estimate` are the maximum likelihood estimates of the parameters. To obtain the approximate SEs of the estimates we do:

```
> sqrt(diag(solve(out$hessian)))
```

A 95% confidence interval would be the parameter estimate ± 1.96 SE.

11.8 Some non-standard models

We conclude this chapter with just a brief mention of some of the other facilities available in R for special regression and data analysis problems.

- **Mixed models.** The recommended **nlme** (`https://CRAN.R-project.org/package=nlme`) package provides functions `lme()` and `nlme()` for linear and non-linear mixed-effects models, that is linear and non-linear regressions in which some of the coefficients correspond to random effects. These functions make heavy use of formulae to specify the models.
- **Local approximating regressions.** The `loess()` function fits a nonparametric regression by using a locally weighted regression. Such regressions are useful for highlighting a trend in messy data or for data reduction to give some insight into a large data set.

  Function `loess` is in the standard package **stats**, together with code for projection pursuit regression.
- **Robust regression.** There are several functions available for fitting regression models in a way resistant to the influence of extreme outliers in the data. Function `lqs` in the recommended package **MASS** (`https://CRAN.R-project.org/package=MASS`) provides state-of-art algorithms for highly-resistant fits. Less resistant but statistically more efficient methods are available in packages, for example function `rlm` in package **MASS** (`https://CRAN.R-project.org/package=MASS`).
- **Additive models.** This technique aims to construct a regression function from smooth additive functions of the determining variables, usually one for each determining variable. Functions `avas` and `ace` in package **acepack** (`https://CRAN.R-project.org/package=acepack`) and functions `bruto` and `mars` in package **mda** (`https://CRAN.R-project.org/package=mda`) provide some examples of these techniques in user-contributed packages to R. An extension is **Generalized Additive Models**, implemented in user-contributed packages **gam** (`https://CRAN.R-project.org/package=gam`) and **mgcv** (`https://CRAN.R-project.org/package=mgcv`).

- **Tree-based models.** Rather than seek an explicit global linear model for prediction or interpretation, tree-based models seek to bifurcate the data, recursively, at critical points of the determining variables in order to partition the data ultimately into groups that are as homogeneous as possible within, and as heterogeneous as possible between. The results often lead to insights that other data analysis methods tend not to yield.

  Models are again specified in the ordinary linear model form. The model fitting function is `tree()`, but many other generic functions such as `plot()` and `text()` are well adapted to displaying the results of a tree-based model fit in a graphical way.

  Tree models are available in R *via* the user-contributed packages **rpart** (`https://CRAN.R-project.org/package=rpart`) and **tree** (`https://CRAN.R-project.org/package=tree`).

12 Graphical procedures

Graphical facilities are an important and extremely versatile component of the R environment. It is possible to use the facilities to display a wide variety of statistical graphs and also to build entirely new types of graph.

The graphics facilities can be used in both interactive and batch modes, but in most cases, interactive use is more productive. Interactive use is also easy because at startup time R initiates a graphics *device driver* which opens a special *graphics window* for the display of interactive graphics. Although this is done automatically, it may useful to know that the command used is `X11()` under UNIX, `windows()` under Windows and `quartz()` under macOS. A new device can always be opened by `dev.new()`.

Once the device driver is running, R plotting commands can be used to produce a variety of graphical displays and to create entirely new kinds of display.

Plotting commands are divided into three basic groups:

- **High-level** plotting functions create a new plot on the graphics device, possibly with axes, labels, titles and so on.
- **Low-level** plotting functions add more information to an existing plot, such as extra points, lines and labels.
- **Interactive** graphics functions allow you interactively add information to, or extract information from, an existing plot, using a pointing device such as a mouse.

In addition, R maintains a list of *graphical parameters* which can be manipulated to customize your plots.

This manual only describes what are known as 'base' graphics. A separate graphics subsystem in package **grid** coexists with base – it is more powerful but harder to use. There is a recommended package **lattice** (`https://CRAN.R-project.org/package=lattice`) which builds on **grid** and provides ways to produce multi-panel plots akin to those in the *Trellis* system in S.

12.1 High-level plotting commands

High-level plotting functions are designed to generate a complete plot of the data passed as arguments to the function. Where appropriate, axes, labels and titles are automatically generated (unless you request otherwise.) High-level plotting commands always start a new plot, erasing the current plot if necessary.

12.1.1 The `plot()` function

One of the most frequently used plotting functions in R is the `plot()` function. This is a *generic* function: the type of plot produced is dependent on the type or *class* of the first argument.

`plot(x, y)`
`plot(xy)` If x and y are vectors, `plot(x, y)` produces a scatterplot of y against x. The same effect can be produced by supplying one argument (second form) as either a list containing two elements x and y or a two-column matrix.

`plot(x)` If x is a time series, this produces a time-series plot. If x is a numeric vector, it produces a plot of the values in the vector against their index in the vector. If x is a complex vector, it produces a plot of imaginary versus real parts of the vector elements.

`plot(f)`
`plot(f, y)`
f is a factor object, y is a numeric vector. The first form generates a bar plot of f; the second form produces boxplots of y for each level of f.

`plot(`*df*`)`
`plot(~ `*expr*`)`
`plot(`*y* `~ `*expr*`)`
: *df* is a data frame, *y* is any object, *expr* is a list of object names separated by '`+`' (e.g., `a + b + c`). The first two forms produce distributional plots of the variables in a data frame (first form) or of a number of named objects (second form). The third form plots *y* against every object named in *expr*.

12.1.2 Displaying multivariate data

R provides two very useful functions for representing multivariate data. If `X` is a numeric matrix or data frame, the command

```
> pairs(X)
```

produces a pairwise scatterplot matrix of the variables defined by the columns of `X`, that is, every column of `X` is plotted against every other column of `X` and the resulting $n(n-1)$ plots are arranged in a matrix with plot scales constant over the rows and columns of the matrix.

When three or four variables are involved a *coplot* may be more enlightening. If `a` and `b` are numeric vectors and `c` is a numeric vector or factor object (all of the same length), then the command

```
> coplot(a ~ b | c)
```

produces a number of scatterplots of `a` against `b` for given values of `c`. If `c` is a factor, this simply means that `a` is plotted against `b` for every level of `c`. When `c` is numeric, it is divided into a number of *conditioning intervals* and for each interval `a` is plotted against `b` for values of `c` within the interval. The number and position of intervals can be controlled with `given.values=` argument to `coplot()`—the function `co.intervals()` is useful for selecting intervals. You can also use two *given* variables with a command like

```
> coplot(a ~ b | c + d)
```

which produces scatterplots of `a` against `b` for every joint conditioning interval of `c` and `d`.

The `coplot()` and `pairs()` function both take an argument `panel=` which can be used to customize the type of plot which appears in each panel. The default is `points()` to produce a scatterplot but by supplying some other low-level graphics function of two vectors `x` and `y` as the value of `panel=` you can produce any type of plot you wish. An example panel function useful for coplots is `panel.smooth()`.

12.1.3 Display graphics

Other high-level graphics functions produce different types of plots. Some examples are:

`qqnorm(x)`
`qqline(x)`
`qqplot(x, y)`
: Distribution-comparison plots. The first form plots the numeric vector `x` against the expected Normal order scores (a normal scores plot) and the second adds a straight line to such a plot by drawing a line through the distribution and data quartiles. The third form plots the quantiles of `x` against those of `y` to compare their respective distributions.

`hist(x)`
`hist(x, nclass=`*n*`)`
`hist(x, breaks=`*b*`, ...)`
: Produces a histogram of the numeric vector `x`. A sensible number of classes is usually chosen, but a recommendation can be given with the `nclass=` argument. Alternatively, the breakpoints can be specified exactly with the `breaks=` argument.

If the `probability=TRUE` argument is given, the bars represent relative frequencies divided by bin width instead of counts.

`dotchart(x, ...)`
: Constructs a dot chart of the data in `x`. In a dot chart the y-axis gives a labelling of the data in `x` and the x-axis gives its value. For example it allows easy visual selection of all data entries with values lying in specified ranges.

`image(x, y, z, ...)`
`contour(x, y, z, ...)`
`persp(x, y, z, ...)`
: Plots of three variables. The `image` plot draws a grid of rectangles using different colours to represent the value of `z`, the `contour` plot draws contour lines to represent the value of `z`, and the `persp` plot draws a 3D surface.

12.1.4 Arguments to high-level plotting functions

There are a number of arguments which may be passed to high-level graphics functions, as follows:

`add=TRUE` Forces the function to act as a low-level graphics function, superimposing the plot on the current plot (some functions only).

`axes=FALSE`
: Suppresses generation of axes—useful for adding your own custom axes with the `axis()` function. The default, `axes=TRUE`, means include axes.

`log="x"`
`log="y"`
`log="xy"` Causes the x, y or both axes to be logarithmic. This will work for many, but not all, types of plot.

`type=` The `type=` argument controls the type of plot produced, as follows:

- `type="p"` Plot individual points (the default)
- `type="l"` Plot lines
- `type="b"` Plot points connected by lines (*both*)
- `type="o"` Plot points overlaid by lines
- `type="h"` Plot vertical lines from points to the zero axis (*high-density*)
- `type="s"`
  `type="S"` Step-function plots. In the first form, the top of the vertical defines the point; in the second, the bottom.
- `type="n"` No plotting at all. However axes are still drawn (by default) and the coordinate system is set up according to the data. Ideal for creating plots with subsequent low-level graphics functions.

`xlab=`*`string`*
`ylab=`*`string`*
: Axis labels for the x and y axes. Use these arguments to change the default labels, usually the names of the objects used in the call to the high-level plotting function.

`main=`*`string`*
: Figure title, placed at the top of the plot in a large font.

`sub=`*`string`*
: Sub-title, placed just below the x-axis in a smaller font.

12.2 Low-level plotting commands

Sometimes the high-level plotting functions don't produce exactly the kind of plot you desire. In this case, low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot.

Some of the more useful low-level plotting functions are:

`points(x, y)`
`lines(x, y)`
: Adds points or connected lines to the current plot. `plot()`'s `type=` argument can also be passed to these functions (and defaults to `"p"` for `points()` and `"l"` for `lines()`.)

`text(x, y, labels, ...)`
: Add text to a plot at points given by `x, y`. Normally `labels` is an integer or character vector in which case `labels[i]` is plotted at point `(x[i], y[i])`. The default is `1:length(x)`.

  Note: This function is often used in the sequence

  ```
  > plot(x, y, type="n"); text(x, y, names)
  ```

  The graphics parameter `type="n"` suppresses the points but sets up the axes, and the `text()` function supplies special characters, as specified by the character vector `names` for the points.

`abline(a, b)`
`abline(h=`*y*`)`
`abline(v=`*x*`)`
`abline(`*lm.obj*`)`
: Adds a line of slope `b` and intercept `a` to the current plot. `h=y` may be used to specify y-coordinates for the heights of horizontal lines to go across a plot, and `v=x` similarly for the x-coordinates for vertical lines. Also *lm.obj* may be list with a `coefficients` component of length 2 (such as the result of model-fitting functions,) which are taken as an intercept and slope, in that order.

`polygon(x, y, ...)`
: Draws a polygon defined by the ordered vertices in (`x`, `y`) and (optionally) shade it in with hatch lines, or fill it if the graphics device allows the filling of figures.

`legend(x, y, legend, ...)`
: Adds a legend to the current plot at the specified position. Plotting characters, line styles, colors etc., are identified with the labels in the character vector `legend`. At least one other argument *v* (a vector the same length as `legend`) with the corresponding values of the plotting unit must also be given, as follows:

  `legend( , fill=`*v*`)`
  : Colors for filled boxes

  `legend( , col=`*v*`)`
  : Colors in which points or lines will be drawn

  `legend( , lty=`*v*`)`
  : Line styles

  `legend( , lwd=`*v*`)`
  : Line widths

  `legend( , pch=`*v*`)`
  : Plotting characters (character vector)

`title(main, sub)`
: Adds a title `main` to the top of the current plot in a large font and (optionally) a sub-title `sub` at the bottom in a smaller font.

`axis(side, ...)`
: Adds an axis to the current plot on the side given by the first argument (1 to 4, counting clockwise from the bottom.) Other arguments control the positioning of the axis within or beside the plot, and tick positions and labels. Useful for adding custom axes after calling `plot()` with the `axes=FALSE` argument.

Low-level plotting functions usually require some positioning information (e.g., x and y coordinates) to determine where to place the new plot elements. Coordinates are given in terms of *user coordinates* which are defined by the previous high-level graphics command and are chosen based on the supplied data.

Where `x` and `y` arguments are required, it is also sufficient to supply a single argument being a list with elements named `x` and `y`. Similarly a matrix with two columns is also valid input. In this way functions such as `locator()` (see below) may be used to specify positions on a plot interactively.

12.2.1 Mathematical annotation

In some cases, it is useful to add mathematical symbols and formulae to a plot. This can be achieved in R by specifying an *expression* rather than a character string in any one of `text`, `mtext`, `axis`, or `title`. For example, the following code draws the formula for the Binomial probability function:

```
> text(x, y, expression(paste(bgroup("(", atop(n, x), ")"), p^x, q^{n-x})))
```

More information, including a full listing of the features available can obtained from within R using the commands:

```
> help(plotmath)
> example(plotmath)
> demo(plotmath)
```

12.2.2 Hershey vector fonts

It is possible to specify Hershey vector fonts for rendering text when using the `text` and `contour` functions. There are three reasons for using the Hershey fonts:

- Hershey fonts can produce better output, especially on a computer screen, for rotated and/or small text.
- Hershey fonts provide certain symbols that may not be available in the standard fonts. In particular, there are zodiac signs, cartographic symbols and astronomical symbols.
- Hershey fonts provide Cyrillic and Japanese (Kana and Kanji) characters.

More information, including tables of Hershey characters can be obtained from within R using the commands:

```
> help(Hershey)
> demo(Hershey)
> help(Japanese)
> demo(Japanese)
```

12.3 Interacting with graphics

R also provides functions which allow users to extract or add information to a plot using a mouse. The simplest of these is the `locator()` function:

`locator(n, type)`
: Waits for the user to select locations on the current plot using the left mouse button. This continues until `n` (default 512) points have been selected, or another mouse button is pressed. The `type` argument allows for plotting at the selected points and has the same effect as for high-level graphics commands; the default is no plotting. `locator()` returns the locations of the points selected as a list with two components `x` and `y`.

`locator()` is usually called with no arguments. It is particularly useful for interactively selecting positions for graphic elements such as legends or labels when it is difficult to calculate in advance where the graphic should be placed. For example, to place some informative text near an outlying point, the command

```
> text(locator(1), "Outlier", adj=0)
```

may be useful. (`locator()` will be ignored if the current device, such as `postscript` does not support interactive pointing.)

`identify(x, y, labels)`
: Allow the user to highlight any of the points defined by `x` and `y` (using the left mouse button) by plotting the corresponding component of `labels` nearby (or the index number of the point if `labels` is absent). Returns the indices of the selected points when another button is pressed.

Sometimes we want to identify particular *points* on a plot, rather than their positions. For example, we may wish the user to select some observation of interest from a graphical display and then manipulate that observation in some way. Given a number of (x, y) coordinates in two numeric vectors `x` and `y`, we could use the `identify()` function as follows:

```
> plot(x, y)
> identify(x, y)
```

The `identify()` functions performs no plotting itself, but simply allows the user to move the mouse pointer and click the left mouse button near a point. If there is a point near the mouse pointer it will be marked with its index number (that is, its position in the `x`/`y` vectors) plotted nearby. Alternatively, you could use some informative string (such as a case name) as a highlight by using the `labels` argument to `identify()`, or disable marking altogether with the `plot = FALSE` argument. When the process is terminated (see above), `identify()` returns the indices of the selected points; you can use these indices to extract the selected points from the original vectors `x` and `y`.

12.4 Using graphics parameters

When creating graphics, particularly for presentation or publication purposes, R's defaults do not always produce exactly that which is required. You can, however, customize almost every aspect of the display using *graphics parameters*. R maintains a list of a large number of graphics parameters which control things such as line style, colors, figure arrangement and text justification among many others. Every graphics parameter has a name (such as '`col`', which controls colors,) and a value (a color number, for example.)

A separate list of graphics parameters is maintained for each active device, and each device has a default set of parameters when initialized. Graphics parameters can be set in two ways: either permanently, affecting all graphics functions which access the current device; or temporarily, affecting only a single graphics function call.

12.4.1 Permanent changes: The `par()` function

The `par()` function is used to access and modify the list of graphics parameters for the current graphics device.

`par()` Without arguments, returns a list of all graphics parameters and their values for the current device.

`par(c("col", "lty"))`
With a character vector argument, returns only the named graphics parameters (again, as a list.)

`par(col=4, lty=2)`
With named arguments (or a single list argument), sets the values of the named graphics parameters, and returns the original values of the parameters as a list.

Setting graphics parameters with the `par()` function changes the value of the parameters *permanently*, in the sense that all future calls to graphics functions (on the current device) will be affected by the new value. You can think of setting graphics parameters in this way as setting "default" values for the parameters, which will be used by all graphics functions unless an alternative value is given.

Note that calls to `par()` *always* affect the global values of graphics parameters, even when `par()` is called from within a function. This is often undesirable behavior—usually we want to set some graphics parameters, do some plotting, and then restore the original values so as not to affect the user's R session. You can restore the initial values by saving the result of `par()` when making changes, and restoring the initial values when plotting is complete.

```
> oldpar <- par(col=4, lty=2)
  ... plotting commands ...
> par(oldpar)
```

To save and restore *all* settable[1] graphical parameters use

```
> oldpar <- par(no.readonly=TRUE)
  ... plotting commands ...
> par(oldpar)
```

12.4.2 Temporary changes: Arguments to graphics functions

Graphics parameters may also be passed to (almost) any graphics function as named arguments. This has the same effect as passing the arguments to the `par()` function, except that the changes only last for the duration of the function call. For example:

```
> plot(x, y, pch="+")
```

produces a scatterplot using a plus sign as the plotting character, without changing the default plotting character for future plots.

Unfortunately, this is not implemented entirely consistently and it is sometimes necessary to set and reset graphics parameters using `par()`.

12.5 Graphics parameters list

The following sections detail many of the commonly-used graphical parameters. The R help documentation for the `par()` function provides a more concise summary; this is provided as a somewhat more detailed alternative.

Graphics parameters will be presented in the following form:

`name=value`
A description of the parameter's effect. *name* is the name of the parameter, that is, the argument name to use in calls to `par()` or a graphics function. *value* is a typical value you might use when setting the parameter.

Note that `axes` is **not** a graphics parameter but an argument to a few `plot` methods: see `xaxt` and `yaxt`.

[1] Some graphics parameters such as the size of the current device are for information only.

12.5.1 Graphical elements

R plots are made up of points, lines, text and polygons (filled regions.) Graphical parameters exist which control how these *graphical elements* are drawn, as follows:

`pch="+"` Character to be used for plotting points. The default varies with graphics drivers, but it is usually 'o'. Plotted points tend to appear slightly above or below the appropriate position unless you use `"."` as the plotting character, which produces centered points.

`pch=4` When `pch` is given as an integer between 0 and 25 inclusive, a specialized plotting symbol is produced. To see what the symbols are, use the command

```
> legend(locator(1), as.character(0:25), pch = 0:25)
```

Those from 21 to 25 may appear to duplicate earlier symbols, but can be coloured in different ways: see the help on `points` and its examples.

In addition, `pch` can be a character or a number in the range `32:255` representing a character in the current font.

`lty=2` Line types. Alternative line styles are not supported on all graphics devices (and vary on those that do) but line type 1 is always a solid line, line type 0 is always invisible, and line types 2 and onwards are dotted or dashed lines, or some combination of both.

`lwd=2` Line widths. Desired width of lines, in multiples of the "standard" line width. Affects axis lines as well as lines drawn with `lines()`, etc. Not all devices support this, and some have restrictions on the widths that can be used.

`col=2` Colors to be used for points, lines, text, filled regions and images. A number from the current palette (see `?palette`) or a named colour.

`col.axis`
`col.lab`
`col.main`
`col.sub` The color to be used for axis annotation, x and y labels, main and sub-titles, respectively.

`font=2` An integer which specifies which font to use for text. If possible, device drivers arrange so that `1` corresponds to plain text, `2` to bold face, `3` to italic, `4` to bold italic and `5` to a symbol font (which include Greek letters).

`font.axis`
`font.lab`
`font.main`
`font.sub` The font to be used for axis annotation, x and y labels, main and sub-titles, respectively.

`adj=-0.1` Justification of text relative to the plotting position. `0` means left justify, `1` means right justify and `0.5` means to center horizontally about the plotting position. The actual value is the proportion of text that appears to the left of the plotting position, so a value of `-0.1` leaves a gap of 10% of the text width between the text and the plotting position.

`cex=1.5` Character expansion. The value is the desired size of text characters (including plotting characters) relative to the default text size.

`cex.axis`
`cex.lab`
`cex.main`
`cex.sub` The character expansion to be used for axis annotation, x and y labels, main and sub-titles, respectively.

12.5.2 Axes and tick marks

Many of R's high-level plots have axes, and you can construct axes yourself with the low-level `axis()` graphics function. Axes have three main components: the *axis line* (line style controlled by the `lty` graphics parameter), the *tick marks* (which mark off unit divisions along the axis line) and the *tick labels* (which mark the units.) These components can be customized with the following graphics parameters.

`lab=c(5, 7, 12)`
The first two numbers are the desired number of tick intervals on the x and y axes respectively. The third number is the desired length of axis labels, in characters (including the decimal point.) Choosing a too-small value for this parameter may result in all tick labels being rounded to the same number!

`las=1` Orientation of axis labels. `0` means always parallel to axis, `1` means always horizontal, and `2` means always perpendicular to the axis.

`mgp=c(3, 1, 0)`
Positions of axis components. The first component is the distance from the axis label to the axis position, in text lines. The second component is the distance to the tick labels, and the final component is the distance from the axis position to the axis line (usually zero). Positive numbers measure outside the plot region, negative numbers inside.

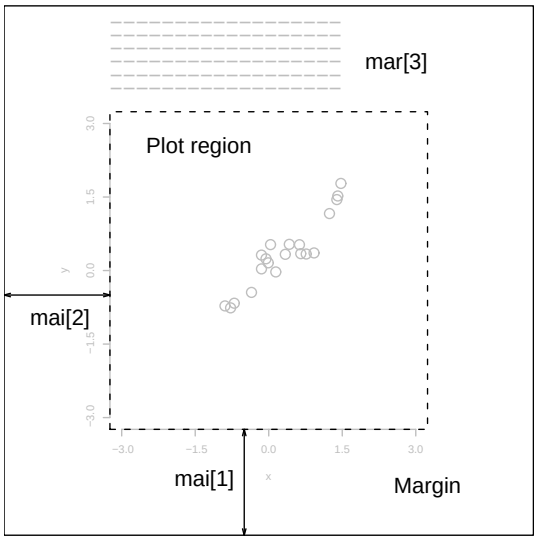
`tck=0.01` Length of tick marks, as a fraction of the size of the plotting region. When `tck` is small (less than 0.5) the tick marks on the x and y axes are forced to be the same size. A value of 1 gives grid lines. Negative values give tick marks outside the plotting region. Use `tck=0.01` and `mgp=c(1,-1.5,0)` for internal tick marks.

`xaxs="r"`
`yaxs="i"` Axis styles for the x and y axes, respectively. With styles `"i"` (internal) and `"r"` (the default) tick marks always fall within the range of the data, however style `"r"` leaves a small amount of space at the edges.

12.5.3 Figure margins

A single plot in R is known as a `figure` and comprises a *plot region* surrounded by margins (possibly containing axis labels, titles, etc.) and (usually) bounded by the axes themselves.

A typical figure is



Graphics parameters controlling figure layout include:

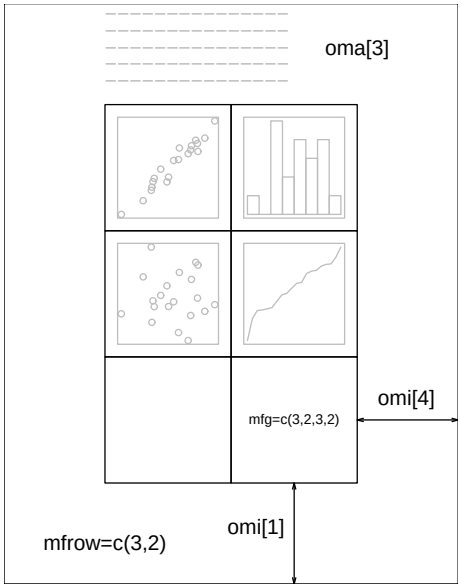
`mai=c(1, 0.5, 0.5, 0)`
: Widths of the bottom, left, top and right margins, respectively, measured in inches.

`mar=c(4, 2, 2, 1)`
: Similar to `mai`, except the measurement unit is text lines.

`mar` and `mai` are equivalent in the sense that setting one changes the value of the other. The default values chosen for this parameter are often too large; the right-hand margin is rarely needed, and neither is the top margin if no title is being used. The bottom and left margins must be large enough to accommodate the axis and tick labels. Furthermore, the default is chosen without regard to the size of the device surface: for example, using the `postscript()` driver with the `height=4` argument will result in a plot which is about 50% margin unless `mar` or `mai` are set explicitly. When multiple figures are in use (see below) the margins are reduced, however this may not be enough when many figures share the same page.

12.5.4 Multiple figure environment

R allows you to create an n by m array of figures on a single page. Each figure has its own margins, and the array of figures is optionally surrounded by an *outer margin*, as shown in the following figure.



The graphical parameters relating to multiple figures are as follows:

`mfcol=c(3, 2)`
`mfrow=c(2, 4)`
: Set the size of a multiple figure array. The first value is the number of rows; the second is the number of columns. The only difference between these two parameters is that setting `mfcol` causes figures to be filled by column; `mfrow` fills by rows.

  The layout in the Figure could have been created by setting `mfrow=c(3,2)`; the figure shows the page after four plots have been drawn.

  Setting either of these can reduce the base size of symbols and text (controlled by `par("cex")` and the pointsize of the device). In a layout with exactly two rows and columns the base size is reduced by a factor of 0.83: if there are three or more of either rows or columns, the reduction factor is 0.66.

`mfg=c(2, 2, 3, 2)`
: Position of the current figure in a multiple figure environment. The first two numbers are the row and column of the current figure; the last two are the number of rows and columns in the multiple figure array. Set this parameter to jump between figures in the array. You can even use different values for the last two numbers than the *true* values for unequally-sized figures on the same page.

`fig=c(4, 9, 1, 4)/10`
: Position of the current figure on the page. Values are the positions of the left, right, bottom and top edges respectively, as a percentage of the page measured from the bottom left corner. The example value would be for a figure in the bottom right of the page. Set this parameter for arbitrary positioning of figures within a page. If you want to add a figure to a current page, use `new=TRUE` as well (unlike S).

`oma=c(2, 0, 3, 0)`
`omi=c(0, 0, 0.8, 0)`
: Size of outer margins. Like `mar` and `mai`, the first measures in text lines and the second in inches, starting with the bottom margin and working clockwise.

Outer margins are particularly useful for page-wise titles, etc. Text can be added to the outer margins with the `mtext()` function with argument `outer=TRUE`. There are no outer margins by default, however, so you must create them explicitly using `oma` or `omi`.

More complicated arrangements of multiple figures can be produced by the `split.screen()` and `layout()` functions, as well as by the **grid** and **lattice** (`https://CRAN.R-project.org/package=lattice`) packages.

12.6 Device drivers

R can generate graphics (of varying levels of quality) on almost any type of display or printing device. Before this can begin, however, R needs to be informed what type of device it is dealing with. This is done by starting a *device driver*. The purpose of a device driver is to convert graphical instructions from R ("draw a line," for example) into a form that the particular device can understand.

Device drivers are started by calling a device driver function. There is one such function for every device driver: type `help(Devices)` for a list of them all. For example, issuing the command

```
> postscript()
```

causes all future graphics output to be sent to the printer in PostScript format. Some commonly-used device drivers are:

`X11()` For use with the X11 window system on Unix-alikes

`windows()`
: For use on Windows

`quartz()` For use on macOS

`postscript()`
: For printing on PostScript printers, or creating PostScript graphics files.

`pdf()` Produces a PDF file, which can also be included into PDF files.

`png()` Produces a bitmap PNG file. (Not always available: see its help page.)

`jpeg()` Produces a bitmap JPEG file, best used for `image` plots. (Not always available: see its help page.)

When you have finished with a device, be sure to terminate the device driver by issuing the command

```
> dev.off()
```

This ensures that the device finishes cleanly; for example in the case of hardcopy devices this ensures that every page is completed and has been sent to the printer. (This will happen automatically at the normal end of a session.)

12.6.1 PostScript diagrams for typeset documents

By passing the `file` argument to the `postscript()` device driver function, you may store the graphics in PostScript format in a file of your choice. The plot will be in landscape orientation unless the `horizontal=FALSE` argument is given, and you can control the size of the graphic with the `width` and `height` arguments (the plot will be scaled as appropriate to fit these dimensions.) For example, the command

```
> postscript("file.ps", horizontal=FALSE, height=5, pointsize=10)
```

will produce a file containing PostScript code for a figure five inches high, perhaps for inclusion in a document. It is important to note that if the file named in the command already exists, it will be overwritten. This is the case even if the file was only created earlier in the same R session.

Many usages of PostScript output will be to incorporate the figure in another document. This works best when *encapsulated* PostScript is produced: R always produces conformant output, but only marks the output as such when the `onefile=FALSE` argument is supplied. This unusual notation stems from S-compatibility: it really means that the output will be a single page (which is part of the EPSF specification). Thus to produce a plot for inclusion use something like

```
> postscript("plot1.eps", horizontal=FALSE, onefile=FALSE,
             height=8, width=6, pointsize=10)
```

12.6.2 Multiple graphics devices

In advanced use of R it is often useful to have several graphics devices in use at the same time. Of course only one graphics device can accept graphics commands at any one time, and this is known as the *current device*. When multiple devices are open, they form a numbered sequence with names giving the kind of device at any position.

The main commands used for operating with multiple devices, and their meanings are as follows:

`X11()` [UNIX]

`windows()`
`win.printer()`
`win.metafile()`
: [Windows]

`quartz()` [macOS]

`postscript()`
`pdf()`

`png()`

`jpeg()`

`tiff()`

`bitmap()`
`...` Each new call to a device driver function opens a new graphics device, thus extending by one the device list. This device becomes the current device, to which graphics output will be sent.

`dev.list()`
Returns the number and name of all active devices. The device at position 1 on the list is always the *null device* which does not accept graphics commands at all.

`dev.next()`
`dev.prev()`
Returns the number and name of the graphics device next to, or previous to the current device, respectively.

`dev.set(which=`*k*`)`
Can be used to change the current graphics device to the one at position *k* of the device list. Returns the number and label of the device.

`dev.off(`*k*`)`
Terminate the graphics device at point *k* of the device list. For some devices, such as `postscript` devices, this will either print the file immediately or correctly complete the file for later printing, depending on how the device was initiated.

`dev.copy(device, ..., which=`*k*`)`
`dev.print(device, ..., which=`*k*`)`
Make a copy of the device *k*. Here `device` is a device function, such as `postscript`, with extra arguments, if needed, specified by '`...`'. `dev.print` is similar, but the copied device is immediately closed, so that end actions, such as printing hardcopies, are immediately performed.

`graphics.off()`
Terminate all graphics devices on the list, except the null device.

12.7 Dynamic graphics

R does not have builtin capabilities for dynamic or interactive graphics, e.g. rotating point clouds or to "brushing" (interactively highlighting) points. However, extensive dynamic graphics facilities are available in the system GGobi by Swayne, Cook and Buja available from

`http://ggobi.org/`

and these can be accessed from R via the package **rggobi** (`https://CRAN.R-project.org/package=rggobi`), described at `http://ggobi.org/rggobi.html`.

Also, package **rgl** (`https://CRAN.R-project.org/package=rgl`) provides ways to interact with 3D plots, for example of surfaces.

13 Packages

All R functions and datasets are stored in *packages*. Only when a package is loaded are its contents available. This is done both for efficiency (the full list would take more memory and would take longer to search than a subset), and to aid package developers, who are protected from name clashes with other code. The process of developing packages is described in Section "Creating R packages" in *Writing R Extensions*. Here, we will describe them from a user's point of view.

To see which packages are installed at your site, issue the command

```
> library()
```

with no arguments. To load a particular package (e.g., the **boot** (`https://CRAN.R-project.org/package=boot`) package containing functions from Davison & Hinkley (1997)), use a command like

```
> library(boot)
```

Users connected to the Internet can use the `install.packages()` and `update.packages()` functions (available through the `Packages` menu in the Windows and macOS GUIs, see Section "Installing packages" in *R Installation and Administration*) to install and update packages.

To see which packages are currently loaded, use

```
> search()
```

to display the search list. Some packages may be loaded but not available on the search list (see Section 13.3 [Namespaces], page 75): these will be included in the list given by

```
> loadedNamespaces()
```

To see a list of all available help topics in an installed package, use

```
> help.start()
```

to start the HTML help system, and then navigate to the package listing in the `Reference` section.

13.1 Standard packages

The standard (or *base*) packages are considered part of the R source code. They contain the basic functions that allow R to work, and the datasets and standard statistical and graphical functions that are described in this manual. They should be automatically available in any R installation. For a complete list, see Section "R packages" in *R FAQ*.

13.2 Contributed packages and CRAN

There are thousands of contributed packages for R, written by many different authors. Some of these packages implement specialized statistical methods, others give access to data or hardware, and others are designed to complement textbooks. Some (the *recommended* packages) are distributed with every binary distribution of R. Most are available for download from CRAN (`https://CRAN.R-project.org/` and its mirrors) and other repositories such as Bioconductor (`https://www.bioconductor.org/`). The *R FAQ* contains a list of CRAN packages current at the time of release, but the collection of available packages changes very frequently.

13.3 Namespaces

Packages have *namespaces*, which do three things: they allow the package writer to hide functions and data that are meant only for internal use, they prevent functions from breaking when a user (or other package writer) picks a name that clashes with one in the package, and they provide a way to refer to an object within a particular package.

For example, `t()` is the transpose function in R, but users might define their own function named `t`. Namespaces prevent the user's definition from taking precedence, and breaking every function that tries to transpose a matrix.

There are two operators that work with namespaces. The double-colon operator `::` selects definitions from a particular namespace. In the example above, the transpose function will always be available as `base::t`, because it is defined in the `base` package. Only functions that are exported from the package can be retrieved in this way.

The triple-colon operator `:::` may be seen in a few places in R code: it acts like the double-colon operator but also allows access to hidden objects. Users are more likely to use the `getAnywhere()` function, which searches multiple packages.

Packages are often inter-dependent, and loading one may cause others to be automatically loaded. The colon operators described above will also cause automatic loading of the associated package. When packages with namespaces are loaded automatically they are not added to the search list.

14 OS facilities

R has quite extensive facilities to access the OS under which it is running: this allows it to be used as a scripting language and that ability is much used by R itself, for example to install packages.

Because R's own scripts need to work across all platforms, considerable effort has gone into make the scripting facilities as platform-independent as is feasible.

14.1 Files and directories

There are many functions to manipulate files and directories. Here are pointers to some of the more commonly used ones.

To create an (empty) file or directory, use `file.create` or `dir.create`. (These are the analogues of the POSIX utilities `touch` and `mkdir`.) For temporary files and directories in the R session directory see `tempfile`.

Files can be removed by either `file.remove` or `unlink`: the latter can remove directory trees.

For directory listings use `list.files` (also available as `dir`) or `list.dirs`. These can select files using a regular expression: to select by wildcards use `Sys.glob`.

Many types of information on a filepath (including for example if it is a file or directory) can be found by `file.info`.

There are several ways to find out if a file 'exists' (a file can exist on the filesystem and not be visible to the current user). There are functions `file.exists`, `file.access` and `file_test` with various versions of this test: `file_test` is a version of the POSIX `test` command for those familiar with shell scripting.

Function `file.copy` is the R analogue of the POSIX command `cp`.

Choosing files can be done interactively by `file.choose`: the Windows port has the more versatile functions `choose.files` and `choose.dir` and there are similar functions in the **tcltk** package: `tk_choose.files` and `tk_choose.dir`.

Functions `file.show` and `file.edit` will display and edit one or more files in a way appropriate to the R port, using the facilities of a console (such as RGui on Windows or R.app on macOS) if one is in use.

There is some support for *links* in the filesystem: see functions `file.link` and `Sys.readlink`.

14.2 Filepaths

With a few exceptions, R relies on the underlying OS functions to manipulate filepaths. Some aspects of this are allowed to depend on the OS, and do, even down to the version of the OS. There are POSIX standards for how OSes should interpret filepaths and many R users assume POSIX compliance: but Windows does not claim to be compliant and other OSes may be less than completely compliant.

The following are some issues which have been encountered with filepaths.

- POSIX filesystems are case-sensitive, so `foo.png` and `Foo.PNG` are different files. However, the defaults on Windows and macOS are to be case-insensitive, and FAT filesystems (commonly used on removable storage) are not normally case-sensitive (and all filepaths may be mapped to lower case).
- Almost all the Windows' OS services support the use of slash or backslash as the filepath separator, and R converts the known exceptions to the form required by Windows.

- The behaviour of filepaths with a trailing slash is OS-dependent. Such paths are not valid on Windows and should not be expected to work. POSIX-2008 requires such paths to match only directories, but earlier versions allowed them to also match files. So they are best avoided.
- Multiple slashes in filepaths such as `/abc//def` are valid on POSIX filesystems and treated as if there was only one slash. They are *usually* accepted by Windows' OS functions. However, leading double slashes may have a different meaning.
- Windows' UNC filepaths (such as `\\server\dir1\dir2\file` and `\\?\UNC\server\dir1\dir2\file`) are not supported, but they may work in some R functions. POSIX filesystems are allowed to treat a leading double slash specially.
- Windows allows filepaths containing drives and relative to the current directory on a drive, e.g. `d:foo/bar` refers to `d:/a/b/c/foo/bar` if the current directory *on drive* `d:` is `/a/b/c`. It is intended that these work, but the use of absolute paths is safer.

Functions `basename` and `dirname` select parts of a file path: the recommended way to assemble a file path from components is `file.path`. Function `pathexpand` does 'tilde expansion', substituting values for home directories (the current user's, and perhaps those of other users).

On filesystems with links, a single file can be referred to by many filepaths. Function `normalizePath` will find a canonical filepath.

Windows has the concepts of short ('8.3') and long file names: `normalizePath` will return an absolute path using long file names and `shortPathName` will return a version using short names. The latter does not contain spaces and uses backslash as the separator, so is sometimes useful for exporting names from R.

File *permissions* are a related topic. R has support for the POSIX concepts of read/write/execute permission for owner/group/all but this may be only partially supported on the filesystem, so for example on Windows only read-only files (for the account running the R session) are recognized. Access Control Lists (ACLs) are employed on several filesystems, but do not have an agreed standard and R has no facilities to control them. Use `Sys.chmod` to change permissions.

14.3 System commands

Functions `system` and `system2` are used to invoke a system command and optionally collect its output. `system2` is a little more general but its main advantage is that it is easier to write cross-platform code using it.

`system` behaves differently on Windows from other OSes (because the API C call of that name does). Elsewhere it invokes a shell to run the command: the Windows port of R has a function `shell` to do that.

To find out if the OS includes a command, use `Sys.which`, which attempts to do this in a cross-platform way (unfortunately it is not a standard OS service).

Function `shQuote` will quote filepaths as needed for commands in the current OS.

14.4 Compression and Archives

Recent versions of R have extensive facilities to read and write compressed files, often transparently. Reading of files in R is to a very large extent done by *connections*, and the `file` function which is used to open a connection to a file (or a URL) and is able to identify the compression used from the 'magic' header of the file.

The type of compression which has been supported for longest is `gzip` compression, and that remains a good general compromise. Files compressed by the earlier Unix `compress` utility can also be read, but these are becoming rare. Two other forms of compression, those of the

`bzip2` and `xz` utilities are also available. These generally achieve higher rates of compression (depending on the file, much higher) at the expense of slower decompression and much slower compression.

There is some confusion between `xz` and `lzma` compression (see `https://en.wikipedia.org/wiki/Xz` and `https://en.wikipedia.org/wiki/LZMA`): R can read files compressed by most versions of either.

File archives are single files which contain a collection of files, the most common ones being 'tarballs' and zip files as used to distribute R packages. R can list and unpack both (see functions `untar` and `unzip`) and create both (for `zip` with the help of an external program).