Chronological Objects for Data Analysis

David A. James
Bell Labs, Lucent Technologies
Murray Hill, NJ 07974
dj@bell-labs.com

Daryl Pregibon
AT&T Laboratories
Florham Park, NJ 07932
daryl@research.att.com

Abstract: This paper concerns ideas and methods for manipulating data occuring in time that supplements traditional times series analysis. We propose a general mechanism that defines time variables in terms of classes of chronological objects and provides a natural means of integrating time into data analysis. This approach is based on the object-oriented paradigm and includes methods for date arithmetic and logical operations, graphics, data reduction, printing, and seamless integration with existing statistical modeling software. We illustrate these ideas through an implementation in the **S** language for data analysis and graphics. Implementation in other languages e.g., LISP-STAT, is straight forward. Data collected during fabrication of optical fiber and software development are used to motivate the ideas.

1 Time in Data Analysis

Time is an important variable in most data analyses. Quite frequently time is expressed in a form unsuitable for computations and analysis; for instance, instead of times between events, one is given dates or clock times at which events occurred. The analyst is faced with the tedious job of manipulating text strings and converting them to either days or seconds to finally compute times between events, only to find that once in numerical form, important temporal ancillary information such as time of day, day of the week, holidays etc., is difficult to retrieve.

We describe a general approach based on object-oriented programming to smoothly integrate time variables into data analysis. We have implemented these ideas in the **S** language (Becker et al., 1988), but the approach is implementable in other systems, e.g., LISP-STAT (Tierney, 1990). With the exception of time-series, nominally defined as a set of equally spaced measurements on a variable, data analysis systems treat time variables as ordinary numerical data, i.e., time variables do not exist per se. This is undesirable for effective data analysis. By exploiting the object-oriented paradigm, time variables can be integrated smoothly into data analysis through *methods* for arithmetic, summaries, printing, and graphics that are called automatically by the system; the design of such a class of objects should be carefully planned in order to ensure that existing and future classes of objects integrate into the data analysis environment.

In his Statlib[1] posting (1992), Therneau independently developed **S** routines that implement a subset of what we propose here. Both implementations define methods for *date* arithmetic, printing, plotting, and summaries. In addition to this, chronological objects represent a richer class of objects that includes time-of-day information, that integrates with existing objects, e.g., dataframes, factors, and that coexists with user-defined objects and methods. In particular, chronological objects fit comfortably with the new statistical modeling software in **S** (Chambers and Hastie, 1992).

2 Chronological Objects

Chronological objects are numeric vectors that represent number of days. There are three classes of chronological objects: `times`, `dates` and `chron`. A `times` object represents elapsed number of days while `dates` and `chron` objects represent number of days from a specified *origin*. The origin for `dates` and `chron` objects is arbitrary, and therefore may be specified by the user; there is a function, `origin()`, to query and to specify the origin of these objects. The difference between `dates` and `chron` objects is that the latter represents time-of-day in addition to dates. A `chron` object inherits from `dates`, and `dates` objects inherit from `times`.

Chronological objects have a `format` attribute that stores the output formatting style for dates and times of day. During the creation of a `dates` object, one may specify the date formatting style through an argument, the default being `"m/d/y"`; any permutation of the letters `"m"`, `"d"` and `"y"` separated by any printing character (possibly null) are recognized; similarly any permutation of the words `"day"`, `"month"` and `"year"` is recognized and produces something like `14 July 1992`. Users may specify formatting functions for specialized styles.

Extensions to chronological objects, for instance to include fiscal calendars, time-zones or daylight savings times information, can be accommodated with extra attributes; existing methods could then be modified to use

[1] Statlib is a repository of public-domain mathematical and statistical software administered by Carnegie-Mellon University.

this extra information.

2.1 Creating Chronological Objects

The function `dates()` creates a `dates` object from a character representation of dates:

```
> dts <- dates(c("02/27/92","02/27/92",
          "01/14/92","02/28/92","02/01/92"))
> dts
[1] 02/27/92 02/27/92 01/14/92
[4] 02/28/92 02/01/92
```

Although the character and `chron` representations of dates look similar, they are quite different; with the latter we can perform "date" arithmetic (see next subsection).

The function `times()` creates a `times` object from a character representation of clock times:

```
> tms <- times( c("23:03:20","22:29:56",
      "01:03:30","18:21:03","16:56:26"))
> tms
[1] 23:03:20 22:29:56 01:03:30
[4] 18:21:03 16:56:26
```

The function `chron()` creates objects with either `dates` or `times` components or both.

```
> x <- chron(dates=dts, times=tms)
> x
 [1] (02/27/92 23:03:20) (02/27/92 22:29:56)
 [3] (01/14/92 01:03:30) (02/28/92 18:21:03)
 [5] (02/01/92 16:56:26)
```

Chronological objects can also be created from integer vectors by interpreting them as Julian dates.

2.2 `chron` Arithmetic and Boolean Operations

Standard arithmetic is defined for `times` objects but not for `dates` and `chron` objects since they represent time relative to an origin. We can add or subtract scalars (which represents days) to `dates` or `chron` objects but multiplication and division are disabled:

```
> c(dts[1], dts[1] + 10)
[1] 02/27/92 03/08/92

> dts[1] - 31
[1] 01/27/92
```

The only meaningful dyadic arithmetic operator is subtraction, which results in a `times` object that represents days between the two operands:

```
> dts[1] - dts[3]
Time in days:
[1] 44
```

Logical comparisons to scalars and to conforming `chron` objects work as expected:

```
> dts[ dts > "01/25/92"]
[1] 02/27/92 02/27/92 02/28/92 02/01/92

> dts > dts[3]
[1] T T F T T
```

In summary, all operations which are sensible for the respective objects are permitted and work as expected; operations that are not sensible are explicitly disabled. This implies that the behavior of functions that use these basic arithmetic and logical operators is predictable:

```
> range(dts)
[1] 01/14/92 02/28/92

> diff(x)
Time in days:
[1]  -0.02319444 -44.89335648  45.72052083
[4] -27.05876157

> sort(dts)[1:3]
[1] 01/14/92 02/01/92 02/27/92
```

2.3 Integration with Other Methods including Graphics

Arithmetic and logical operations are formally handled in object-oriented systems by a *generic* function that invokes another function, called a *method*, according to the class of the operands; the method then carries out the arithmetic and logical operation as defined for that particular class of objects. The previous section illustrated the leverage gained by tailoring a special method for chronological objects as all functions which perform arithmetic operations invoke the method. Other methods used implicitly above are the subscript, `"["()`, printing methods, `print()`, and combine `c()`.

Chronological objects should coexist naturally with existing and user-defined objects. In **S**, a `dataframe` is a generalization of a matrix whereby columns need not be of the same mode or class, and it plays an important role in the tools for statistical modeling. The **S** implementation of chronological objects anticipates the need for integration and ensures that methods for browsing, printing, plotting, and summarizing individual objects in a dataframe behave appropriately; also it provides simple means to generate common objects such as `factors` from chronological objects. Section 3 provides several examples of this integration.

Graphics is important in data analysis — it allows our human vision system to quickly perceive structure or lack thereof. Our implementation reflects the emphasis on graphics by providing a suite of methods which result in natural and expected behavior of standard graphical functions. We later demonstrate the use of the `plot()` method for chronological objects. This method is identical to the usual default method except that for a single supplied argument, the empirical distribution function is plotted rather than a sequence-order plot. A special function `axis.times()` ensures that axes are labeled appropriately for chronological objects. The same holds for other graphical functions such as `points()`, `lines()`, `identify()`, and `hist()`.

2.4 Special Date Handling Functions

Often it is useful to view data that occurs through time by aggregating them over selected intervals or time units; `days()`, `weekdays()`, `months()`, and `years()` take a `dates` object and return an ordered factor (categorical variable with fixed levels) denoting the corresponding component of a dates vector. Strictly speaking, some of these functions should return a cyclic or periodic factor; however, such a class of objects has yet to be created for our **S** implementation. The objects produced by the above functions can be used to carry out the partitioning of a primary variable prior to aggregation (e.g., as the second argument to the S function `split()`):

```
> weekdays(dts)
[1] Thu Thu Tue Fri Sat
Levels:
   Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

The method for `cut()` can be used to create an ordered factor from a `dates` object with an arbitrary set of break points through a `breaks=` argument. This argument can take one of three forms: it can be a `dates` object specifying variable-width intervals, a number indicating the number of equally spaced intervals, or one of the strings `"days"`, `"weeks"`, `"months"`, or `"years"`. In contrast to the above functions, `cut()` simply partitions the time interval instead of creating periodic subintervals (like weekdays).

```
> cut(dts, "months")
[1] Feb 92 Feb 92 Jan 92 Feb 92 Feb 92
Levels:
   Jan 92 < Feb 92
```

`is.weekend()`, `is.holiday()` and `leap.year()` return logical vectors indicating whether the corresponding element of the dates vector is a weekend, a holiday or a leap year, respectively. In either case, these logical variables provide yet another way to subscript `chron` objects:

```
> is.weekend(dts)
[1] F F F F T
```

3 Applications

3.1 Software Development and Modification Requests

Development of large systems requires a highly disciplined way of introducing features and changes. Changes to a system may be needed due to faults in existing software, new functionality, changes in customer requirements, emergence of new standards, porting to new environments, etc. At AT&T software organizations, development is controlled through a process based on modification requests (MRs). A modification request is a formal request to change something in a system; it can refer to changes in software, firmware, hardware, documentation, or even development processes themselves. For each MR, additional information such as its severity, type, subsystems it affects, etc., is recorded. As development progresses MRs visit various states (under investigation, being fixed, under testing, pending approval, etc.) until they are "closed", i.e., incorporated into the product. Dates associated with state transitions are also recorded and provide the basic data to model MR time-to-close as a function of type, severity, etc. A subset of MRs was extracted from a project management database that includes information on MRs. The first 3 lines of the (ASCII) file are given by:

```
               opened type sev state  entered
mr91-00301 01/03/91 enh  3   c      03/11/91
mr91-01402 01/14/91 doc  4   c      07/11/91
mr91-01804 01/18/91 sw   2   ui     01/18/91
```

Time-to-close is censored for MRs that are outstanding. The MR data are read into S by:

```
> mrs <- read.table("mr.data", header=T,
                    as.is=c(F,T,F,F,F,T))
> mrs$opened <- dates(mrs$opened)
> mrs$entered <- dates(mrs$entered)
> mrs$sev <- ordered(mrs$sev,levels=4:1,
                 labels=c("low","med",
                     "high","critical"))
```

The last expression creates an ordered factor for MR severities. A summary of the data is simply obtained and shows the integration of `dates` methods with `dataframe` methods:

```
> summary(mrs)
   opened             type         sev
Min.   :01/03/91   doc:32   low      :13
1st Qu.:03/03/91   enh: 5   med      :43
Median :07/03/91   hw :12   high     :22
Mean   :07/01/91   sw :35   critical: 6
3rd Qu.:10/05/91
Max.   :12/19/91

state         entered
bf:13   Min.   :01/18/91
c :47   1st Qu.:07/22/91
ui:24   Median :09/30/91
        Mean   :09/21/91
        3rd Qu.:12/03/91
        Max.   :02/14/92
```

Plotting these data is natural and direct:

```
> attach(mrs)
> plot(opened, xlab="MR Opening Date",
       ylab="Cumulative Number of MRs")
```

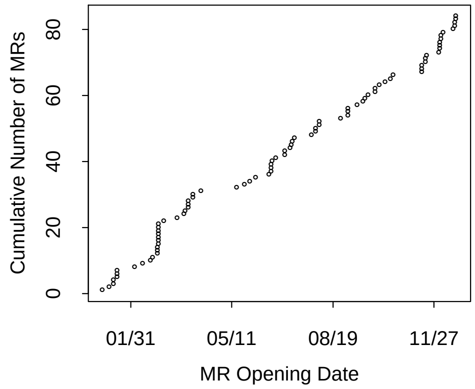

Figure 1: `dates` *plotting method. The plot shows cumulative number of MRs against opening dates.*

We can easily count events for specific time periods:

```
> table(months(opened))
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
  7  14   7   3   3   7   7   5   9   4   6  12
```

A well-designed class of objects and its methods should coexist with other user-defined objects and methods. The next example shows the seamless interface of chronological objects and user-defined censored objects. We compute time-to-close for each MR, taking into consideration that some MRs have not yet been closed:

```
> day.of.analysis <- dates("02/14/92")
> mr.time <- entered - opened
> # identify MRs not yet closed
> i <- state != "c"
> mr.time[i] <- day.of.analysis-opened[i]
> mr.time <- censored(mr.time, !i)
```

We have defined the variable `mr.time` as a `censored` object. The time-to-close data are conveniently displayed in an event chart(Goldman, 1992):

```
> i <- sev=="high"
> event.chart(mr.time[i], opened[i],
     xlab="Time to Close (Days)",
     ylab="Date of Opening (High Severity)")
```

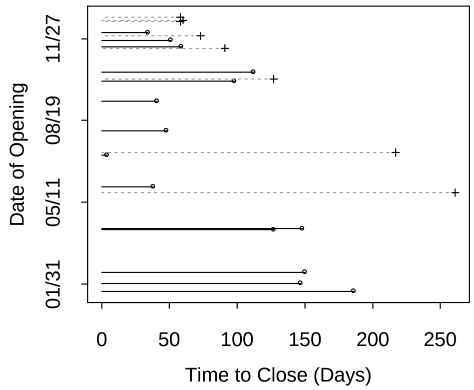

Figure 2: *Event chart for MR time-to-close for severity High. The solid lines depict time-to-close, and the dotted lines depict censored observations.*

The `event.chart()` function is straightforward given the available functions for `chron` arithmetic and plotting.

3.2 Manufacturing of Optical Fiber

The first stage in the manufacture of optical fiber by the modified chemical vapor deposition (MCVD) process consists of depositing vaporized chemicals on the interior of a glass tube and collapsing the tube to a solid rod, called a preform. Measurements on a preform are made immediately in order to characterize preform geometry and the ability of the subsequent fiber to guide light. The measurements are saved in a commercial relational database.

Our second data set consists of measurements derived from the refractive index profile of preforms manufactured over a two month period. Variables of interest included `lathe`, `date`, and `ccratio`. A lathe is a machine where preforms are manufactured, `date` refers to the manufacturing date, and the `ccratio` is the ratio of the overall diameter to the core diameter of the preform. As part of exploratory data analysis we produced a plot of the within-lathe standard deviation versus the within-lathe mean for `ccratio`; Figure 3 (top panel) displays this plot. Lathes with large standard deviations are of interest in understanding variability in preform processing. Perusal of the lathes with the largest standard deviations consistently shows that the variance is dominated by one or more outlying preforms. Lathes with moderately large standard deviations tend to have more interesting behavior. Figure 3 (bottom panel) shows a boxplot(Chambers et al., 1983) of `ccratio` by week for lathe 129. Apparently the lathe was calibrated during week 4 (where it showed high variance) with the net result that the *level* of `ccratio` increased by 0.15 units after the change.

```
> i <- lathe == "129"
> plot(x = cut(date[i], breaks="week"),
       y = ccratio[i],
       xlab = "Production Week",
       ylab="ccratio (lathe 129)")
```

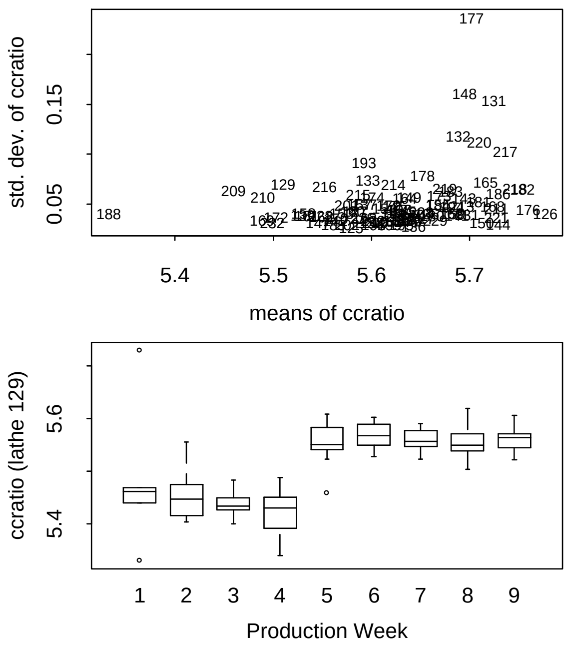

Figure 3: *Mean-dispersion plot (top panel) of* `ccratio` *by lathe. Boxplot (bottom panel) of* `ccratio` *by weeks for lathe 129.*

The above expression, which creates a boxplot, illustrates the integration of new objects with existing ones: The function `cut()` takes a `dates` object and produces an `ordered` factor; the **S** dispatcher then invokes the plotting method for factors, which produces a boxplot of `ccratio` for each level of the factor, i.e., weeks.

3.3 Relaxation of Optical Fiber

Optical fiber is drawn from a preform by melting the tip of the preform at high temperature. The thin strand of glass is rapidly cooled, coated, and wound onto spools. Measurement on the resulting fibers follows usually within a few hours, but current manufacturing trends aim toward reducing work-in-progress by measuring fibers immediately after being drawn. To ascertain whether this would have an impact on measurements of absorptive loss (dB/km), an important quality characteristic of optical fiber, several fibers were systematically remeasured at varying time intervals after draw. The data were recorded on a personal computer connected to an optical time-domain reflectometer; the data was read into the data frame `relax`, and it includes day and time of measurements, and loss measured at two wave lengths, 1310nm and 1550nm.

It is important to characterize the nature and extent of relaxation of fiber loss from draw time in order to predict "steady state" loss from an initial early measurement. Figure 4 depicts the extent of fiber relaxation at both wave lengths.

```
> attach(relax)
> plot(time, w1550, ylab="change in loss",
    xlab="Time of Measurement")
> points(time, w1310)
```

Next we need a model to capture the smooth decrease in loss exhibited in Figure 4. One such model is

$$loss(t) = \alpha + \beta e^{-\theta t} + error$$

This model nicely captures the boundary conditions at elapsed times of 0 and ∞ where $loss(0) = \alpha + \beta$ and $loss(\infty) = \alpha$. Thus β is the total amount of relaxation, α the steady state loss, and θ the rate constant. Specifying the model, setting starting values, and fitting are illustrated for loss at 1310nm:

```
> draw <- chron("05/22/92","14:00:00")
> f1310 <- formula(w1310 ~ alpha +
             beta * exp(-theta*(time-draw)))
> s1310 <- list(alpha = w1310[1], beta =
             diff(range(w1310)), theta = 1)
> m1310 <- nls(f1310, start = s1310)
```

Note that the behavior of the computations did not change when a `chron` object appeared in the model formula. The results of these computations and identical ones for 1550nm are best examined graphically:

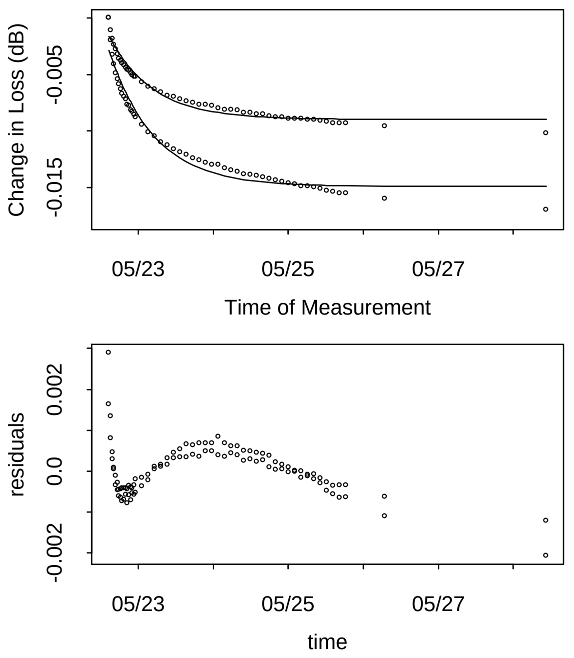

Figure 4: *Change in loss versus measurement time after draw (top panel) at 1310nm and 1550nm. The solid lines are obtained by nonlinear least squares. Residuals from the nonlinear model are displayed in the bottom panel.*

```
> plot(time, residuals(m1550),
         ylab="residuals")
> points(time, residuals(m1310))
```

We note that while the model has some theoretical appeal, systematic lack of fit is apparent. However, in the absence of alternative models, this one might be useful in providing a conservative estimate of steady state loss.

Our implementation of `chron` objects integrates smoothly into the statistical modeling software distributed in the current release of **S**, as illustrated with the nonlinear least-squares fit of the previous example. In (James and Pregibon, 1992) we also describe how to decompose a times series into seasonal, trend, and remainder series (Cleveland et al., 1990) This further illustrates the integration of `chron` objects into the modeling language and the use of the `months()` function to pull out a seasonal effect.

4 Internals of Chronological Objects

Dates are stored as Julian dates, i.e., number of days since a fixed point in time called the *origin*; the origin is arbitrary and by default is midnight of January 1, 1970. Each object has its own origin, and the function `origin()` displays it or modifies it:

```
> origin(dts)
month day year
    1   1 1970
> dts[10]
[1] 02/23/92
> origin(dts) <- c(month=1, day=1, year=1990)
> dts[10]
[1] 02/23/92
```

Note that the change in origin only affects the internal representation and that subtraction of two `dates` or `chron` objects returns a `times` object, i.e., the origin is dropped.

The functions `julian()` and `month.day.year()`, taken from the library `examples` (Becker et al., 1988), were modified to take an extra optional argument `origin`. See the detailed documentation in Appendix B.

The function `as.numeric()` returns the Julian date of a `dates` object while `as.character()` returns its character representation:

```
> as.numeric(dts[1:5])
[1] 8092 8092 8048 8093 8066

> as.character(dts[1:5])
[1] "02/27/92" "02/27/92" "01/14/92"
[4] "02/28/92" "02/01/92"
```

In the case of a `chron` object, clock times are represented as fractions of a day. So `as.numeric()` returns the Julian day plus the fractional part:

```
> as.numeric(dts.tms[1:5])
[1] 8092.961 8092.937 8048.044 8093.765
[5] 8066.706
```

All chronological objects have a `format` attribute that stores the output formatting style for dates and times of day. During the creation of a `dates` object, you may specify the date formatting style through the argument `format`, the default being `"m/d/y"`; any permutation of the letters `"m", "d"` and `"y"` separated by any printing character (possibly null) are recognized. Thus `"d-m-y"` produces `14-07-92` and `"ymd"` produces `920714`. It is also possible to specify dates in "full" e.g., `July 14 1992`; any permutation of the words `"month"`, `"day"` and `"year"` is recognized. For instance `"day month year"` produces `14 July 1992`. Note that the output date format may be different than the input format—use the `out.format` argument to specify a different one. You may also use `format()` or `chron()` to change the output date style of a chronological object:

```
> format(dt2) <- 'y.m.d'
> dts2[1:5]
[1] 92.01.01 92.02.27 92.01.14
[4] 92.02.28 92.02.01
```

Although the above styles are the most common, there are many other styles not covered; for those you may specify the name of a formatting function to either `format` or `out.format`. The format functions should take as input a vector of characters and return a vector of Julian dates; the output format function should take a vector of Julian dates and produce a character vector. See Appendix A for an example.

Printing of `chron` objects is handled by an implicit or explicit call to the `print()` method. The `print()` method has an argument, `simplify`, that controls printing: if `FALSE` the full date and/or times are printed, otherwise the year component is dropped from dates that span only one calendar year and seconds are dropped from `times` objects. The option `chron.simplify` has the same effect:

```
> options(chron.simplify=T)
> dts.tms[1:5]
[1] (01/01 23:03) (02/27 22:29) (01/14 01:03)
[4] (02/28 18:21) (02/01 16:56)

> print(dts[1:5], simplify=F)
[1] 01/01/92 02/27/92 01/14/92 02/28/92
[5] 02/01/92
```

As can be seen from the last example, the `simplify` argument to `print` takes precedence over the global option `chron.simplify`.

Extensions to these objects, e.g., to include time-zone information or fiscal calendars, could be accommodated with extra attributes; existing methods could then be modified to use the extra attributes.

5 Conclusions

Time variables are common in data analysis. Often they dictate the primary course of the analysis (e.g., with failure time data) and other times they simply provide an important way to slice or aggregate the primary variables. Implementations of time variables in statistical computing environments should facilitate computations with time variables in order to fully exploit the temporal structure in data.

Our implementation of chronological objects in the **S** language was designed to integrate smoothly into its statistical modeling software.

We believe that similar implementations could be easily written in other systems; however such implementations may depend on object-oriented principles for operations to be intuitive and consistent, in particular arithmetic, aggregation, printing, data summaries, and plotting.

References

Becker, R. A., Chambers, J. W., and Wilks, A. R. (1988). *The New S Language*. Wadsworth, Pacific Grove, California.

Chambers, J. W., Cleveland, W. S., Kleiner, B., and Tukey, P. A. (1983). *Graphical Methods for Data Analysis*. Wadsworth, Pacific Grove, California.

Chambers, J. W. and Hastie, T., editors (1992). *Statistical Models in S*. Wadsworth, Pacific Grove, California.

Cleveland, R. B., Clevelenad, W. S., McRae, J. E., and Terpening, I. (1990). Stl: A seasonal-trend decomposition procedure based on loess. *Journal of Official Statistics*, 6:3–73.

Goldman, A. I. (1992). Eventcharts: Visualizing survival and other timed-events data. *The American Statistician*, 46(1):13–18.

James, D. A. and Pregibon, D. (1992). Chronological objects in S. AT&T Bell Laboratories Technical Report (available electronically `ftp://cm.bell-labs.com/ms/departments-/sia/dj/papers/chron92.ps`.

Tierney, L. (1990). *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*. Wiley, New York.

A Specifying Functions for Date Formatting

The simple mechanism described in Section 2 to specify formatting styles is adequate for many situations, but there will be occasions in which existing formats will not suffice. Note that formatting is relevant both for reading as well as printing. In the following example we show how we parse dates in a format that does not correspond to the specifications in Section 2. Suppose the data is stored as "14-jul-92", "15-jul-92", etc. The following function returns the Julian date for such data:

```
> in.fmt <- function(str){
      d <- as.integer(substring(str, 1, 2))
      m <- substring(str, 4, 6)
      mon <- lower.case(month.abb)
      m <- match(m, mon)
      y <- as.integer(substring(str, 8, 9))
      julian(m, d, 1900+y)
      }
> dts3.str
[1] "01-jan-92" "27-feb-92" "14-jan-92" "28-feb-92" "01-feb-92"
[6] "30-jan-92" "08-aug-92" "10-jan-92" "19-jan-92" "23-feb-92"

> dts3 <- chron(dts3.str, format = as.name("in.fmt"),
                          out.format = "m/d/y")
> dts3
[1] 01/01/92 02/27/92 01/14/92 02/28/92 02/01/92 01/30/92
[7] 08/08/92 01/10/92 01/19/92 02/23/92
```

The argument `out.format` is important in the above example; by default the output format is the same as the input format, thus had we not specified `out.format="m/d/y"` `chron()` would have set the output format to the function `in.fmt` and when we had tried to print the object the function `in.fmt` would not have worked. Notice that since `chron` also takes Julian dates as input, the following expression would have worked as well:

```
> dts3 <- chron( in.fmt(dts3.str) )
```

To print dates out in a format *similar* to the input, we write the following function:

```
> out.fmt <- function(x){
      out <- month.day.year(x)
      mon <- lower.case(month.abb)
      mon <- mon[out$month]
      paste(out$day, mon, out$year-1900, sep="-")
  }
> dts3 <- chron(dts3, out.format=as.name("out.fmt"))
> range(dts3)
[1] 1-jan-92 8-aug-92
```

It is important to coerce `out.fmt` to a name (notice the call to `as.name`). First, if it's given as a character it will be erroneously interpreted as a format according to the rules described in Section 2. Second, if it's given as a function, the complete body of the function is saved with the variable; this is wasteful, especially as the number of such variables increases. Also, by saving the name of the function only, we can modify easily the formatting function without having to recreate all the variables that use it.

B S Functions and Chronological Methods

This appendix contains the detailed documentation of **S** functions that recognize chronological objects, methods defined for them, and other miscellaneous functions.

chron	Create Chronological Object	**chron**

```
chron(dates, times, format = c("m/d/y","h:m:s"), out.format, origin)
```

ARGUMENTS

`dates:` an optional character or numeric vector specifying dates. If character, `dates` are assumed to be in one of the date formats below; if numeric, `dates` are assumed to be Julian dates, i.e., number of days since `origin`.

`times:` optional character or numeric vector specifying times of day. If character, `times` are assumed to be in one of the time formats below; if numeric, `times` are assumed to be fractions of a day.

`format:` two-element vector or list specifying date and time input formats, respectively. The format can be either strings specifying one of the recognized formats below or a list of user-supplied functions to convert dates from character into Julian dates and vice versa.

The dates format can be any permutation of the characters "d", "m", or "y" delimited by a separator (possibly null), e.g., "m/d/y", "d-m-y", "ymd", are all valid; the format can also be permutations of the words "day" "month" and "year", which produces the month name, e.g., "month day year" produces "April 20 1992", "day month year" produces "20 April 1992".

The times format can be any permutation of "h", "m", and "s" separated by any one non-special character. The default is "h:m:s".

`out.format:` two-element vector or list specifying date and time format of the output. Default is same as `format`.

`origin:` a vector specifying the date with respect to which Julian dates are computed. Default is `c(month = 1, day = 1, year = 1970)`.

VALUE

An object of class `times` if only `times` were specified, `dates` if only `dates`, or `chron` if both `dates =` and `times =` were supplied. All these inherit from class `times`.

These objects represent dates and times of day, and allow the following arithmetic and summaries: subtraction `d1-d2`, constant addition `d1+constants`, all logical comparisons, summaries `min()`, `max()`, and `range()` (which drop NAs by default); constants specify days (fractions are converted to time-of-day, e.g., 2.5 represents 2 days and 12 hours). Operations such as sorting, differencing, etc., are automatically handled.

There are methods for `as.character()`, `as.numeric()`, `cut()`, `is.na()`, `print()`, `summary()`, `plot()`, `lines()`, `lag()`, and the usual subsetting functions `[`, `[<-`. The functions `days()`, `months()`, `quarters()`, `years()`, `weeks()` `weekdays()`, `hours()`, `minutes()`, and `seconds()` take any `chron` object as input and extract the corresponding time interval. `cut()` is used to create ordered factors from `chron` objects.

The current implementation of `chron` objects does not handle time zones nor daylight savings time.

SEE ALSO

dates, times, cut.dates, seq.dates

EXAMPLE

```
attach("/usr/dj/s/chron/.Data")
mr.date <- chron(c("09/28/89","02/09/90","09/15/88","08/30/91") )
# printing
> mr.date
> [1] 09/28/89 02/09/90 09/15/88 08/30/91
# chron logic and arithmetic operations
> range(mr.date)
[1] 09/15/88 08/30/91
> mr.date[ mr.date > "01/01/90" ]
[1] 02/09/90 08/30/91
> dts <- c("900928", "901022", "901106", "901119", "901128")
> tms <- c("23:12:55", "10:34:02", "08:30:00", "01:15:00", "20:00:00")
> lathe.date <- chron(dts, tms, format=c("ymd", "h:m:s") )
> lathe.date
[1] (09/28/90 23:12:55) (10/22/90 10:34:02) (11/06/90 08:30:00)
[4] (11/19/90 01:15:00) (11/28/90 20:00:00)
> lathe.date[1] + 2.5        # two and a half days after lathe start
[1] (10/01/90 11:12:55)
> lathe.date[2] - lathe.date[1]        # time between events
[1] (24 days and 11:21:07)
> summary(all.mrs$mr.date)
     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
 10/15/86 02/13/89 03/30/90 02/07/90 02/16/91 10/25/91
# plotting
> plot(mr.date)            # plot cumulative number of events vs date
> hist(lathe.start)        # histogram of time of events
> plot(lathe.start, fiber.loss)
# modeling with chron objects:
# decompose CO2 concentration into long and seasonal trends
> tms <- paste(rep(1:12,32), 1, rep(59:70, rep(12,32)), sep = "/")
> tms <- chron(tms, format = "m/d/y")
> D <- gam(co2 ~ s(tms, 5) + s(tms, 40) + months(tms))
```

cut.dates	Create a Factor from a chron or dates object	**cut.dates**

```
cut.dates(x, breaks, labels)
```

ARGUMENTS

`x`: chron or dates object (see `chron`), character dates such as "10/04/91" or Julian dates).

`breaks`: either a vector of break points (a `dates` vector, character dates such as "10/04/91" or Julian dates), a constant specifying number of equally spaced intervals extending from `min(x)-1` to `max(x)+1`, or one of the strings in `c("days", "weeks", "months", "year")` specifying a time period.

`labels`: character labels for intervals.

VALUE

an ordered factor whose levels represent the various time intervals.

SEE ALSO

seq.dates

EXAMPLE

```
> opened <- chron(dates = opened.str)
> cut(opened, "months")
 [1] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
 [9] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[17] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[25] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[33] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[41] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[49] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[57] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[65] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[73] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[81] Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92 Feb 92
[89] Feb 92 Feb 92 Feb 92
 Jan 92 < Feb 92
# distribution (boxplots) of x weeks
plot(cut(opened, "weeks"), mr.time)
```

dates	Generate Dates and Times Components from Input	**dates**

```
dates(x, ...)
times(x,...)
```

ARGUMENTS

`x`: a chron object, a character vector, or a numeric vector specifying time. If character, it must be in a format recognized by `chron()`. If numeric, it specifies Julian dates, i.e., number of days since an origin.

`...`: parameters for `chron()`.

VALUE

An object of class `dates` or `times`, depending of the function called.

These functions return objects inheriting from dates and times, respectively. They call `chron()` if `x` does not belong to any of the chronological classes.

SEE ALSO

chron

EXAMPLE

```
class(events)
[1] "chron" "dates" "times"
dts <- dates(events)
time.of.day <- times(events)
```

hours — Return Hours, Minutes, or Seconds from a Times Object — hours

```
hours(x)
minutes(x)
seconds(x)
```

ARGUMENTS

`x`: an object inheriting from class "times".

VALUE

the corresponding time period as an ordered factor.

EXAMPLE

```
x <- chron(dates=tmp, times=tms)
hh <- hours(x)
boxplot(split(y, hh))
```

is.weekend — Find Weekends and Holidays in a Chron or Dates Object — is.weekend

```
is.weekend(x)
is.holiday(x, holidays)
```

ARGUMENTS

`x`: an object inheriting from `dates`.

`holidays`: optional `chron` or `dates` object listing holidays. If argument `holidays` is missing, either the value of the object `.Holidays` (if it exists) or `NULL` will be used.

VALUE

a logical object indicating whether the corresponding date is a weekend in the case of `is.weekend()` or a holiday in the case of `is.holiday()`.

SEE ALSO

days, weekdays, months, quarters, years

EXAMPLE

```
# compute daily production averages (delete weekends and holidays)
> dts <- chron(dates=dts.str)
> .Holidays
 New Year Memorial Indepen. Labor day Thanksgiving Christmas
 01/01/92 05/25/92 07/04/92  09/07/92     11/26/92   12/25/92
> working.days <- !is.weekend(dts) & !is.holiday(dts)
> sapply( split(x[working.days], dts[working.days]), mean )
```

julian	Convert between Julian and Calendar Dates	julian

```
julian(m, d, y, origin=c(month=1, day=1, year=1970))
month.day.year(jul, origin=c(month=1, day=1, year=1970))
leap.year(y)
day.of.week(month, day, year)
```

ARGUMENTS

`m`: vector of month numbers.

`d`: vector of day numbers.

`y`: vector of years.

`jul`: vector of Julian Dates, i.e., number of days since `origin`.

`origin`: vector specifying the origin as month, day, and year. The default is `c(month=1, day=1, year=1970)`.

VALUE

A vector of Julian dates (number of days since `origin`) when `julian()` is called, or a list with members `month`, `day`, `year` corresponding to the input Julian dates if `month.day.year()` is called. `leap.year()` returns a logical vector indicating whether the corresponding year is a leap year. `day.of.week()` returns a number between 0 and 6 to specify day of the week–0 refers to Sunday.

These functions were taken from Becker, Chambers, and Wilks (1988), and were slightly modified to take `chron` and `dates` objects; some also take the extra argument `origin`. The original functions are stored in the `library(example)`.

SEE ALSO

chron

EXAMPLE

```
n.days <- julian(mon, day, 1990+yr, origin=c(1,1,1990))
```

.Holidays	Variables used by the chron library	.Holidays

```
.Holidays
day.name
day.abb
month.name
month.abb
```

VALUE

The first object is a `dates` vector of known holidays, and as such it should be initialized according to the time period and location of the data. See the function `is.holiday()`.

The others are character vectors storing the English names and abbreviations of the days of the week and the months, respectively.

days — Return Various Periods from a Chron or Dates Object — days

```
days(x)
weekdays(x, abb=T)
months(x, abb=T)
quarters(x, abb=T)
years(x)
```

ARGUMENTS

`x`: a dates object.

`abb`: should abbreviated names be returned? Default is `TRUE`.

VALUE

an ordered factor corresponding to days, weekdays, months, quarters, or years of `x` for the respective function.

SEE ALSO

is.weekend, is.holiday

EXAMPLE

```
# produce boxplots by day of the week and months
plot( weekdays(dts), x)
plot( months(dts), y)
```

seq.dates — Generate Chron or Dates Sequences — seq.dates

```
seq.dates(from, to, by="days", length.)
```

ARGUMENTS

`from`: starting date; it can be a chron or dates objet, a character string, e.g., "05/23/91", or a Julian date.

`to`: ending date, like `from`.

`by`: either a numeric value or one of the valid strings `"days"`, `"weeks"`, `"months"`, or `"years"`.

`length.`: optional number of elements in the sequence.

VALUE

a sequence with values (`from`, `from + by`, `from + 2*by`, `...`, `to`) of class `class(from)` and origin `origin(from)`. Note that `from` must be less than or equal to the argument `to`.

SEE ALSO

cut.dates

EXAMPLE

```
> seq.dates("01/01/92", "12/31/92", by="months")
 [1] 01/01/92 02/01/92 03/01/92 04/01/92 05/01/92 06/01/92
 [7] 07/01/92 08/01/92 09/01/92 10/01/92 11/01/92 12/01/92
> end.of.the.month <- seq.dates("02/29/92", by="month", length=15)
> end.of.the.month
 [1] 02/29/92 03/31/92 04/30/92 05/31/92 06/30/92 07/31/92
 [7] 08/31/92 09/30/92 10/31/92 11/30/92 12/31/92 01/31/93
[13] 02/28/93 03/31/93 04/30/93
```