Chronological Objects in S

David A. James
Daryl Pregibon
AT&T Bell Laboratories, Murray Hill, NJ 07974

October 2, 1992

Abstract

S [Becker et al., 1988] is a powerful language and system for data analysis used extensively in academic and research institutions. With the exception of time-series, the **S** language treats time variables as ordinary numerical data, thus making ancillary information such as day-of-week, month, holidays, etc., difficult to retrieve. We describe a class of chronological objects that provide a natural and seamless means of integrating time variables into data analysis, and in particular, into the statistical modeling software [Chambers and Hastie, 1992]. This is attained using the object-oriented aspects of the **S** language whereby *methods* for date arithmetic, common date manipulations, as well as printing and plotting are defined. Several motivating applications are provided and detailed documentation is given in an Appendix.

1 Chronological Variables 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. Furthermore styles for clock times and dates vary from one application to another (sometimes even within an application!), making the task more burdensome. 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, ancillary information such as day-of-week, shift, holidays etc., is difficult to retrieve.

We describe an implementation of chronological objects in the **S** language and system for data analysis [Becker et al., 1988]. With the exception of time-series, nominally defined as a set of equally spaced measurements on a variable, **S** functions treat time variables as ordinary numerical data. As argued above, this is inconvenient for effective data analysis. Our implementation allows the analyst to work with time variables in a natural and intuitive way. We exploit the object-oriented paradigm of **S** [Chambers and Hastie, 1992, Appendix A] to seamlessly integrate time variables into data analysis and define arithmetic, summaries, printing, and graphics in a consistent manner.

Our approach to calendar data is similar to that of Therneau (Statlib[1] 1992 posting). He provides a class of *date objects* and methods (i.e. S functions) for date arithmetic, printing, and plotting. Our work, which was independently developed, supplies additional functionality and integration with the S language. In particular, *chronological objects* were designed to fit comfortably with the new modeling software[Chambers and Hastie, 1992]. We also developed associated data structures and methods for clock times, which together with calendar variables, provide a unified yet flexible approach to time-oriented data.

Section 2 introduces chronological objects and the operations defined on them. Section 3 provides a more detailed exposition using three applications that motivated our approach. Section 4 briefly describes the internal representation of chronological objects for readers wanting to extend or modify the current implementation. Section 5 states our conclusions. Appendix A contain comments on date formats; Appendix B includes detailed documentation.

2 Chronological Objects

There are three classes of chronological objects: `times`, `dates` and `chron`. A `times` object represents elapsed time (in days) while `dates` and `chron` objects represent dates, that is, time from a specified origin. 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`.

All chronological objects have a `format` attribute that stores the output formatting style for dates and times of day. The variety of styles and conventions are illustrated by our examples. Tailoring format styles is discussed in Appendix A.

2.1 Creating Chronological Objects

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

```
> dts.str
[1] "02/27/92" "02/27/92" "01/14/92" "02/28/92" "02/01/92"
[6] "01/30/92" "01/08/92" "01/10/92" "01/19/92" "02/23/92"

> dts <- dates(dts.str)
> dts
```

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

```
[1] 02/27/92 02/27/92 01/14/92 02/28/92 02/01/92 01/30/92
[7] 01/08/92 01/10/92 01/19/92 02/23/92
```

Although they look similar, the internal representations of `dts.str` and `dts` 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.str
[1] "23:03:20" "22:29:56" "01:03:30" "18:21:03" "16:56:26"
[6] "12:05:06" "22:17:15" "20:09:59" "13:07:49" "11:42:26"

> tms <- times(tms.str)
> tms
[1] 23:03:20 22:29:56 01:03:30 18:21:03 16:56:26 12:05:06
[7] 22:17:15 20:09:59 13:07:49 11:42:26
```

The function `chron()` creates objects with either `dates` or `times` components or both. In the latter case, the functions `dates()` and `times()` can be used to extract the dates and times components from a `chron` object:

```
> dts.tms <- chron(dts.str, tms.str)
> dts.tms
 [1] (02/27/92 23:03:20) (02/27/92 22:29:56) (01/14/92 01:03:30)
 [4] (02/28/92 18:21:03) (02/01/92 16:56:26) (01/30/92 12:05:06)
 [7] (01/08/92 22:17:15) (01/10/92 20:09:59) (01/19/92 13:07:49)
[10] (02/23/92 11:42:26)

> dates(dts.tms)
[1] 02/27/92 02/27/92 01/14/92 02/28/92 02/01/92 01/30/92
[7] 01/08/92 01/10/92 01/19/92 02/23/92

> times(dts.tms)
[1] 23:03:20 22:29:56 01:03:30 18:21:03 16:56:26 12:05:06
[7] 22:17:15 20:09:59 13:07:49 11:42:26
```

Chronological objects can be created from integer vectors by interpreting them as Julian dates. The following example creates a vector of random 1992 dates:

```
> rdts <- sample(0:365, size=10)       # 1992 is a leap year
> rdts <- chron(rdts, origin = c(1, 1, 1992) )
> rdts
[1] 08/31/92 11/28/92 01/20/92 03/18/92 07/18/92 01/15/92
[7] 05/24/92 01/24/92 02/09/92 10/26/92
```

2.2 chron Arithmetic

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:

```
> dts + 10
[1] 03/08/92 03/08/92 01/24/92 03/09/92 02/11/92 02/09/92
[7] 01/18/92 01/20/92 01/29/92 03/04/92

> dts - 1
[1] 02/26/92 02/26/92 01/13/92 02/27/92 01/31/92 01/29/92
[7] 01/07/92 01/09/92 01/18/92 02/22/92

> 3 * dts
Error in Ops.dates(3, dts): * not defined for chron objects
Dumped
```

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

```
> dts[1] + dts[2]
Error in Ops.dates(dts[1], dts[2]): chron objects may not
be added together
Dumped

> dts[1] - dts[3]
[1] 44
```

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

```
> dts > "01/25/92"
 [1] T T F T T T F F F T

> dts[ dts > "01/25/92"]
[1] 02/27/92 02/27/92 02/28/92 02/01/92 01/30/92 02/23/92

> dts > rev(dts)
 [1] T T T T T F F F F F
```

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 **S** functions that use these basic arithmetic and logical operators is predictable:

```
> range(dts)      # min(dts) and max(dts) work as well
[1] 01/08/92 02/28/92

> diff(range(dts.tms))
[1] (50 days 20:03:47)

> sort(dts)
 [1] 01/08/92 01/10/92 01/14/92 01/19/92 01/30/92 02/01/92
 [7] 02/23/92 02/27/92 02/27/92 02/28/92
```

2.3 Integration with Other Methods including Graphics

Arithmetic and logical operations are formally handled in **S** 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, `"["()`, and printing methods, `print()`.

We expect that time variables co-exist with other character and numeric data in a given application. This implies that chronological objects should naturally co-exist with **S** objects that represent such data, for example **`factors`**, or any other user-defined class of objects. Data frames are used in S as a repository for application-specific data; they also play an important role in the tools for statistical modeling. A `dataframe` is a generalization of a matrix whereby columns need not be of the same mode or class. Our implementation of chronological objects anticipated such needs by ensuring that methods for browsing, printing, plotting, and summarizing individual objects in a dataframe behave appropriately. Section 3 provides several examples of this behavior.

Graphics are important in data analysis — they allow 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()` and `identify()`.

Table 1 lists some additional methods for chronological objects. These refer to existing S functions that we converted to `generic` functions with the previous definition serving as the default. For details on default methods and inheritance see [Chambers and Hastie, 1992].

seq()	cut()	format()
diff()	hist()	

Table 1: *New generic functions for which chronological methods exist.*

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. The functions `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. These derived variables can be used to carry out the partitioning prior to aggregation (e.g., as the second argument to the S function `split()`):

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

Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

The function `cut.dates()` can be used to create an ordered factor from a `dates` object with an arbitrary set of break points through the `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.dates()` uses all components of the dates vector in defining the factor levels:

```
> DTS <- c(dts, dts + 366)
> months(DTS)
 [1] Feb Feb Jan Feb Feb Jan Jan Jan Jan Feb Feb Feb Jan Feb Feb
[16] Jan Jan Jan Jan Feb
Levels:
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct"
[11] "Nov" "Dec"

Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec

> cut.dates(DTS, "months")
 [1] Feb 92 Feb 92 Jan 92 Feb 92 Jan 92 Jan 92 Jan 92 Jan 92 Jan 92
[10] Feb 92 Feb 93 Feb 93 Jan 93 Feb 93 Jan 93 Jan 93 Jan 93 Jan 93
[19] Jan 93 Feb 93
Levels:
 [1] "Jan 92" "Feb 92" "Mar 92" "Apr 92" "May 92" "Jun 92" "Jul 92"
```

```
[10] "Aug 92" "Sep 92" "Oct 92" "Nov 92" "Dec 92" "Jan 93" "Feb 93"

 Jan 92 < Feb 92 < Mar 92 < Apr 92 < May 92 < Jun 92 < Jul 92
 < Aug 92 < Sep 92 < Oct 92 < Nov 92 < Dec 92 < Jan 93 < Feb 93
```

The functions `is.weekend()` and `is.holiday()` return a logical vector indicating whether the corresponding element of the dates vector is a weekend or a holiday. The function `is.holiday()` matches supplied dates to the `chron` object `.Holidays` (by default `NULL`). In either case, these logical variables provide yet another way to subscript `chron` objects:

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

> dts[is.weekend(dts)]
 [1] 02/01/92 01/19/92 02/23/92

> .Holidays <- chron(c("01/01/92","05/25/92","07/04/92",
                       "09/07/92","11/26/92","12/25/92"))
> is.holiday(dts)
[1] F F F F F F F F F F
```

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 the *Change Management Tracking System*, a project management database that includes information on MRs[2]. The first 10 lines of the (ASCII) file are given by:

[2] Of course, outstanding MRs are thereby censored.

```
              opened type sev state  entered
mr91-00301 01/03/91 enh  3   c     03/11/91
mr91-01007 01/10/91 hw   3   c     10/09/91
mr91-01402 01/14/91 doc  3   c     07/11/91
mr91-01403 01/14/91 doc  3   c     07/23/91
mr91-01804 01/18/91 sw   3   ui    01/18/91
mr91-01802 01/18/91 doc  2   c     07/23/91
mr91-01803 01/18/91 hw   3   c     01/08/92
mr91-03503 02/04/91 hw   2   c     07/01/91
mr91-04302 02/12/91 hw   2   c     07/12/91
```

These data are read into S by:

```
> mrs <- read.table(file = "mr.data", header = T,
                    as.is = c(F, T, F, F, F, T) )
> mrs$opened <- chron(dates = mrs$opened)
> mrs$entered <- chron(dates = mrs$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:

```
> summary(mrs)
      opened          type          sev    state       entered
 Min.   :01/03/91   doc:32   low     :13   bf:13   Min.   :01/18/91
 1st Qu.:03/03/91   enh: 5   med     :43   c :47   1st Qu.:07/22/91
 Median :07/03/91   hw :12   high    :22   ui:24   Median :09/30/91
 Mean   :07/01/91   sw :35   critical: 6           Mean   :09/21/91
 3rd Qu.:10/05/91                                  3rd Qu.:12/03/91
 Max.   :12/19/91                                  Max.   :02/14/92
```

Plotting these data is natural and direct:

```
> attach(mrs)
> plot(opened, xlab = "MR Opening Date",
       ylab = "Cumulative Number of MRs") # see Figure 1
```

We can easily count events for specific time periods:

```
 # simple count of MRs by month
> table(months(opened))
 Jan 91 Feb 91 Mar 91 Apr 91 May 91 Jun 91 Jul 91 Aug 91
      7     14      7      3      3      7      9      3
```

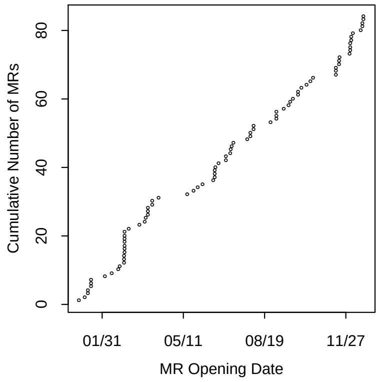

Figure 1: `dates` *plotting method. The plot shows cumulative number of MRs against opening dates.*

```
 Sep 91 Oct 91 Nov 91 Dec 91
      9      4      7     11
# Count number of outstanding MRs by month
> period <- seq(from = min(opened)-30, to = max(entered),
                by = "months")
> labels <- paste(as.character(months(period)),
                  as.numeric(years(period)) - 1900)[-1]
> opn <- cut(opened, breaks = period, labels = labels)
>  cls <- cut(entered[state=="c"], breaks = period,
                   labels = labels)
> cumsum(table(opn) - table(cls))
 Jan 91 Feb 91 Mar 91 Apr 91 May 91 Jun 91 Jul 91 Aug 91
      1      8     21     29     30     34     38     27

 Sep 91 Oct 91 Nov 91 Dec 91 Jan 92 Feb 92
     30     29     28     40     46     37
```

We compute time-to-close for each MR, taking into consideration that some MRs have not yet been closed:

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

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]:

```
> for(i in levels(severity))  # see Figure 2
  event.chart(mr.time[severity==i], opened[severity==i])
```

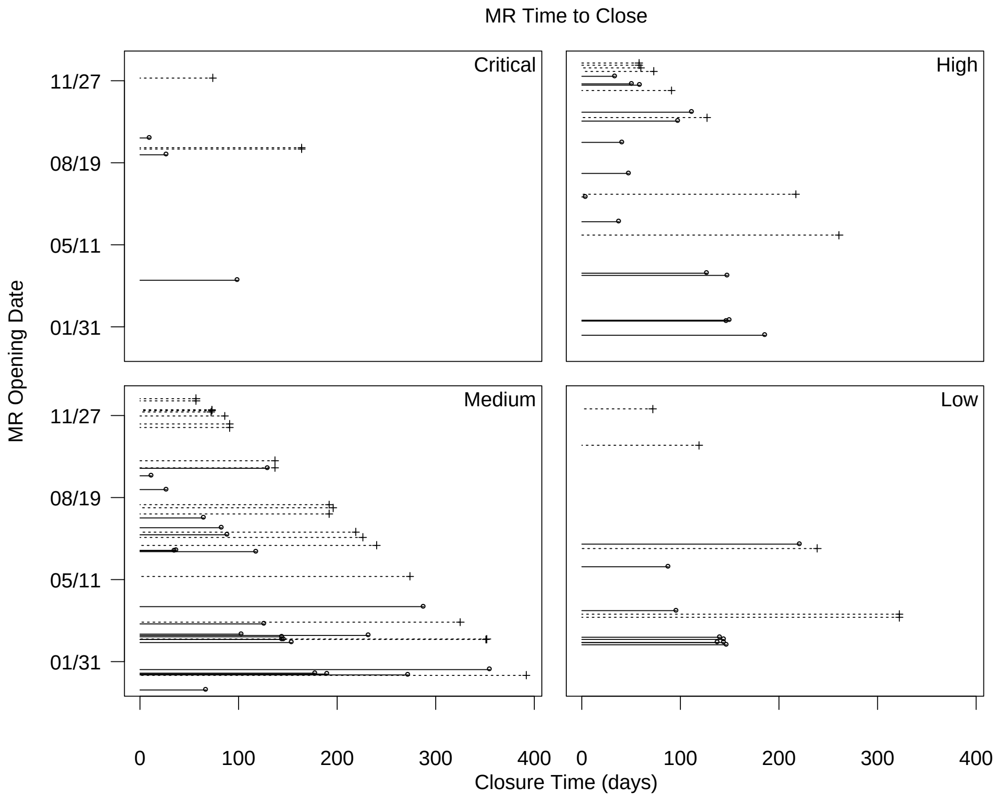

Figure 2: *Event chart for MR time-to-close by severity. The solid lines depict time-to-close, and the dotted lines depict censored observations.*

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 *Product Tracking System*, a database running on a commercial relational database package.

Our second data set consists of four measurements derived from the refractive index profile of preforms manufactured over a two month period. Data was extracted into an ASCII file that was read into **S**; the first 10 lines of the file are:

```
        lathe   date  delta  pcore  ovality  ccratio
H62647    125 920102  0.378 12.187     3.38    5.570
H62628    125 920103  0.377 11.955     0.74    5.613
H75459    125 920104  0.371 12.011     0.22    5.619
H75631    125 920104  0.379 12.083     1.34    5.588
H64206    125 920105  0.376 12.168     0.94    5.561
H75537    125 920105  0.378 12.052     1.76    5.565
H75786    125 920106  0.370 12.051     0.98    5.617
H64695    125 920107  0.377 12.067     0.00    5.582
H75867    125 920107  0.374 12.010     1.63    5.607
```

These data were read into the data frame `Scan` as follows:

```
> Scan <- read.table("Scan", head = T)
> Scan$lathe <- factor(Scan$lathe)
> Scan$date <- chron(as.character(Scan$date), format = "ymd")
```

Note that the date formatting style is `"ymd"`. A summary of the 7,984 observations on the 6 variables contained in the data.frame `Scan` is easily obtained:

```
> summary(Scan)
      lathe           date              deltas
 129    : 117   Min.   :920102   Min.   :0.3310
 184    : 116   1st Qu.:920120   1st Qu.:0.3760
 179    : 114   Median :920203   Median :0.3830
 161    : 114   Mean   :920202   Mean   :0.3821
 180    : 113   3rd Qu.:920216   3rd Qu.:0.3890
 231    : 112   Max.   :920229   Max.   :0.7790
 (Other):7298

      pcore          ovality          ccratio
 Min.   : 8.488   Min.   :0.000   Min.   :5.022
 1st Qu.:11.980   1st Qu.:0.500   1st Qu.:5.574
 Median :12.060   Median :1.020   Median :5.623
 Mean   :12.070   Mean   :1.193   Mean   :5.623
 3rd Qu.:12.160   3rd Qu.:1.700   3rd Qu.:5.674
 Max.   :12.740   Max.   :8.960   Max.   :7.888
```

As part of exploratory data analysis we produced a plot of the within-lathe standard deviation versus the within-lathe mean for each of the four preform characteristics. Figure 3 (top panel) displays this plot for the variable `ccratio`. 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.

```
> plot(weeks(date[lathe=="129"]), ccratio[lathe=="129"],
          main = "129") # see Figure 3
```

The above expression, which creates a boxplot, illustrates the integration of new objects with existing ones: The function `weeks()` 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 automatically recorded on a personal computer attached to an optical time-domain reflectometer. The following data shows the first 10 measurements of power loss deviations at wavelengths of 1310nm and 1550nm for a specific fiber drawn at 2pm on 5/22/92:

```
May 22 14:41 1992  0.0000  0.0000
May 22 15:13 1992 -0.0011 -0.0020
May 22 15:46 1992 -0.0019 -0.0033
May 22 16:18 1992 -0.0024 -0.0041
May 22 16:50 1992 -0.0028 -0.0049
May 22 17:22 1992 -0.0032 -0.0054
May 22 17:54 1992 -0.0036 -0.0059
May 22 18:26 1992 -0.0038 -0.0063
May 22 18:58 1992 -0.0040 -0.0067
May 22 19:31 1992 -0.0042 -0.0070
```

The following **S** expressions create the data frame `relax`:

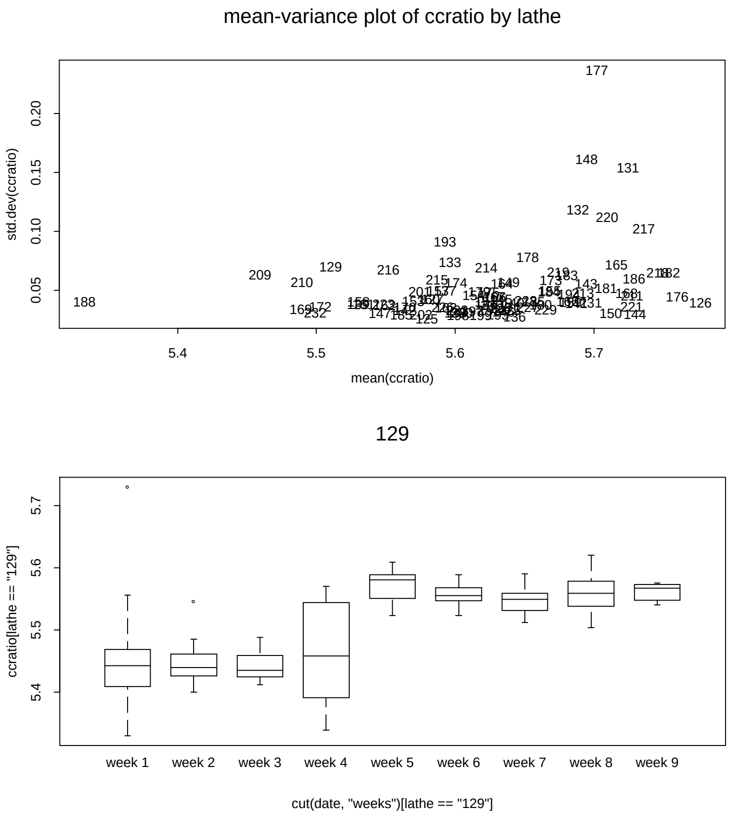

Figure 3: *Mean-variance plot (top panel) of tt ccratio by lathe. Boxplot (bottom panel) of* `ccratio` *by weeks for lathe 129.*

```
> relax <- scan(file = "relax.data", what = list(month ="", day =0,
             time = "", year = 0, wave1310 = 0, wave1550 = 0) )
> # paste together dates and times (append seconds)
> dts.str <- paste(relax$month, relax$day, relax$year)
> tms.str <- paste(relax$time, "00", sep = ":")

> relax$time  <- chron(dates = dts.str, times = tms.str,
                       format = c("month day year", "h:m:s"))
> relax$month <- relax$day <- relax$year <- NULL
```

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. This first involves computing the time elapsed between draw and each of the measurements:

```
> attach(relax)
> elapsed <- time - chron("05/22/92", "14:00:00")
> plot(elapsed, wave1550, ylab="change in loss", font=-1)
> points(elapsed, wave1310)   # see Figure 4 (left panel)
```

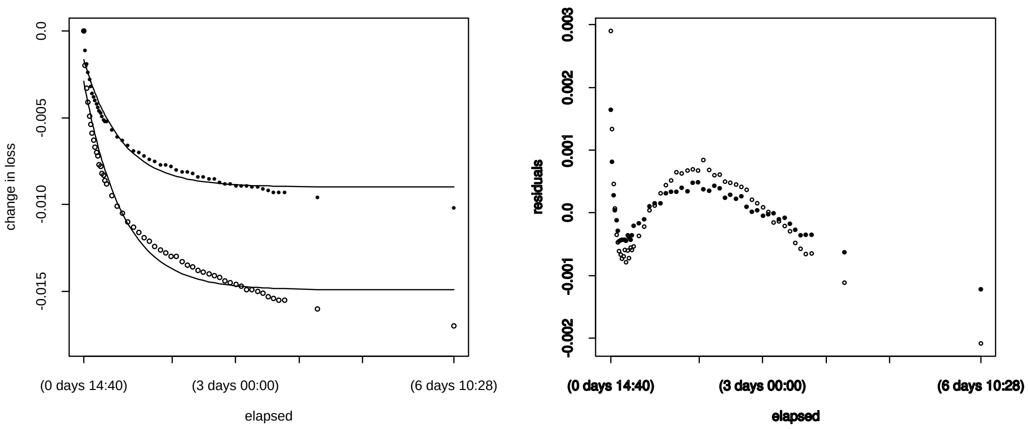

Figure 4: *Change in loss versus time since draw (left panel) at 1310nm and 1550nm. The solid lines are obtained by nonlinear least squares. Residuals from the nonlinear model are displayed in the right panel.*

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:

```
> f1310 <- formula(wave1310 ~ alpha + beta * exp(-theta*elapsed))
> s1310 <- list(alpha = wave1310[1], beta = diff(range(wave1310)),
                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:

```
> # add fitted curves to Figure 4
> lines(elapsed, fitted(m1310) )
> lines(elapsed, fitted(m1550) )

> # plot residuals plot in right panel of Figure 4
> plot(elapsed, residuals(m1550), ylab="residuals", font=-1)
> points(elapsed, 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.

3.4 Chronological Objects in Statistical Models

Our implementation of `chron` objects integrates smoothly into the statistical modeling software distributed in the current release of **S**. As another example, we describe how to mimic the behavior of the function `stl()` which decomposes a times series object into seasonal, trend, and remainder series [Cleveland et al., 1990]. There are several reasons why one might want to mimic `stl()` behavior:

- `stl()` does not handle NAs;
- `stl()` does not handle irregularly spaced time series,
- `stl()` cannot decompose data with several seasonal components.

There are compelling reasons why `stl()` should still be the method of choice in well behaved cases, and we list them at the end of the section; our purpose here however is simply to further illustrate the integration of `chron` objects into the modeling language and the use of the `months()` function to pull out a seasonal effect.

We consider the example in the `stl()` documentation which uses carbon monoxide data from 1959 to 1990 in the **S** dataset `co2`. The expression

```
> d <- stl(co2, ss.window=17, fc.window=c(101,25),
                fc.degree=c(1,2))
> plot(d) # see Figure 5
```

produces an `stl` object which plots as displayed in Figure 5. The figure displays the data in the top panel and in the remaining panels, its decomposition into long-term trend, short-term trend, seasonal variation, and the remainder.

We create a `chron` object to correspond to the `tsp` attribute of `co2`:

```
> Date <- paste(rep(1:12,32), 1, rep(59:90, rep(12,32)), sep = "/")
> Date <- chron(Date)
```

A decomposition of co2 over time can be represented as a generalized additive model [Hastie and Tibshirani, 1990]. Fitting this model is accomplished using the gam() function provided by the statistical modeling software [Chambers and Hastie, 1992]. The following expressions fit a gam model and plot the terms in the fitted model in "stl()" style:

```
> D <- gam(formula = co2 ~ s(Date, 5) + s(Date, 40) + months(Date))
> par(mfrow = c(5,1), mar = c(0, 4.1, 0, 4.1))
> plot(Date, co2, type = "l") # see Figure 6
> plot(D, rug = F)
> plot(Date, residuals(D), type= "h")
```

The formula uses the default scatterplot smoother s() with different bandwidths to remove long-term and short-term trend, and then the months() function to fit a (constant) seasonal component. The remainder term is then plotted using available modeling functions, i.e., residuals().

The remarkable similarity of Figures 5 and 6 might suggest deprecating stl() since its behavior can be reproduced by the statistical modeling software. But the decomposition of time series is a well studied problem and stl(), which embodies much experience in accomplishing this decomposition, should be the method of choice. A particular aspect that is critical is the order in which components are fitted. It is also critical that the short-term trend be free from any long-term trend. The stl()function forces both of these to occur as required, but gam() does not. The appropriate ordering of terms in a gam() formula is long-term-trend + short-term-trend + seasonal. Different orderings produce different decompositions which are equivalent in the sense that trend = long-term + short-term, i.e., that the sum of the two components is well-determined but not the components themselves. Given this ordering, long-term trend can be implicitly forced out of the short-term trend by insuring that the long-term trend is rich enough. This is accomplished by increasing the degrees of freedom of the scatterplot smoother used to model this component. For both gam() and stl(), trial and error is required to effectively set the required smoothing parameters. Finally, stl() uses smoothing on the seasonal component as well, where in our example, we used constants for each subseries. We can move toward stl() by replacing the last term in the formula by months(dat)*dat. This models the seasonal subseries linearly in time. In principle, smoothing the subseries is possible by replacing this last term by months(dat) * s(dat, 3), but crossing a factor with a smooth is not yet implemented. Other attempts, say using b-splines, bs(), or natural splines, ns(), fail because these parametric components are fitted first in the backfitting algorithm of gam() which disrupts the ordering we require for the decomposition.

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] 8092 8092 8048 8093 8066 8064 8042 8044 8053 8088

> as.character(dts)
[1] "02/27/92" "02/27/92" "01/14/92" "02/28/92" "02/01/92"
[6] "01/30/92" "01/08/92" "01/10/92" "01/19/92" "02/23/92"
```

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] 8092.961 8092.937 8048.044 8093.765 8066.706 8064.504 8042.929
[8] 8044.840 8053.547 8088.488
```

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 chron() to change the output date style of a chronological object:

```
> dts2 <- chron(dts, out.format="m-d-y")
> dts2
[1] 01-01-92 02-27-92 01-14-92 02-28-92 02-01-92 01-30-92
[7] 01-08-92 01-10-92 01-19-92 02-23-92
```

Time formats are specified similarly to date formats; any permutation of the letters h, m, and s separated by a printing character are valid. Thus "h.m.s" produces "08.34.23".

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] (01/01 23:03) (02/27 22:29) (01/14 01:03) (02/28 18:21)
[5] (02/01 16:56) (01/30 12:05) (01/08 22:17) (01/10 20:09)
[9] (01/19 13:07) (02/23 11:42)
> print(dts, simplify=F)
[1] 01/01/92 02/27/92 01/14/92 02/28/92 02/01/92 01/30/92
[7] 01/08/92 01/10/92 01/19/92 02/23/92
```

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 ubiquitous 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. Any implementation of time variables in statistical computing environments should facilitate computations of time variables in order to exploit the temporal structure in data.

Our implementation of chronological objects provides a natural and seamless means of integrating time variables into data analysis in the S language. We use the object-oriented features of the language to provide methods for date arithmetic, common date manipulations, printing, graphics, and summaries. Chronological objects were designed to integrate smoothly into statistical modeling software. Some functions that are arguably natural for chronological objects such as `cut.dates()`, require the `dates` suffix because the current S language does not define `cut()` as a generic. As syntactic sugar, we elevated such functions to generic methods and redefined the current S function as the default.

We expect that our implementation could be mimicked by other systems. We believe however that any such implementation requires a *class* of chronological *objects* in order for operations on time variables such as arithmetic, printing, and plotting to be intuitive and consistent with behavior of other (ordinary) variables. This implies that the target system needs to provide an object-oriented programming language or similar features.

6 Acknowledgments

We want to express our appreciation for the comments, suggestions, and bug reports we received from Linda A. Clark, Lorraine Denby, Brad Kummer, and Scott Vander Wiel. Thanks also to Allan W. Laird and Andy Ingles for providing us with the software and manufacturing data, respectively. Some of our ideas on chronological objects were influenced by the language FIT©developed at AT&T Bell Laboratories by Lloyd H. Nakatani and Laurence W. Ruedisueli.

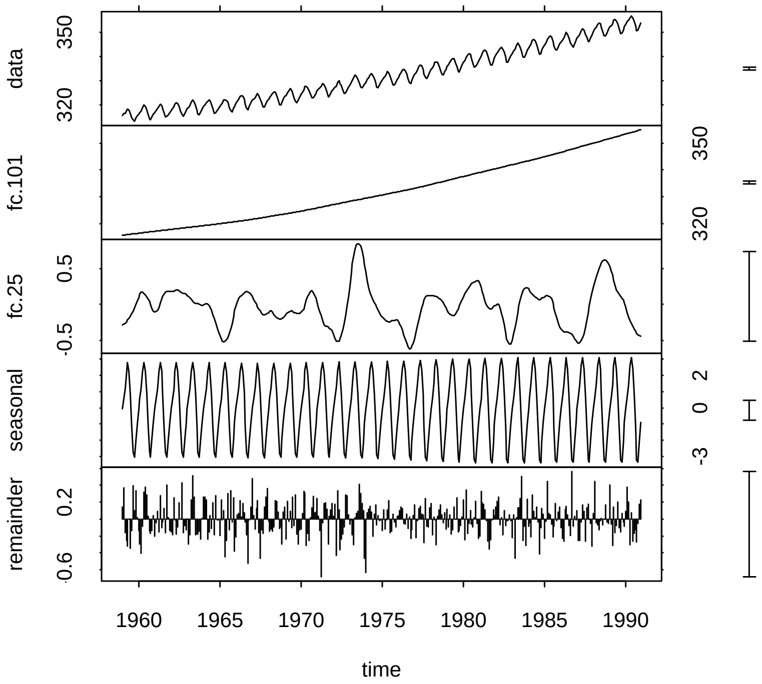

Figure 5: *STL Decomposition of* `co2` *data into frequency components*

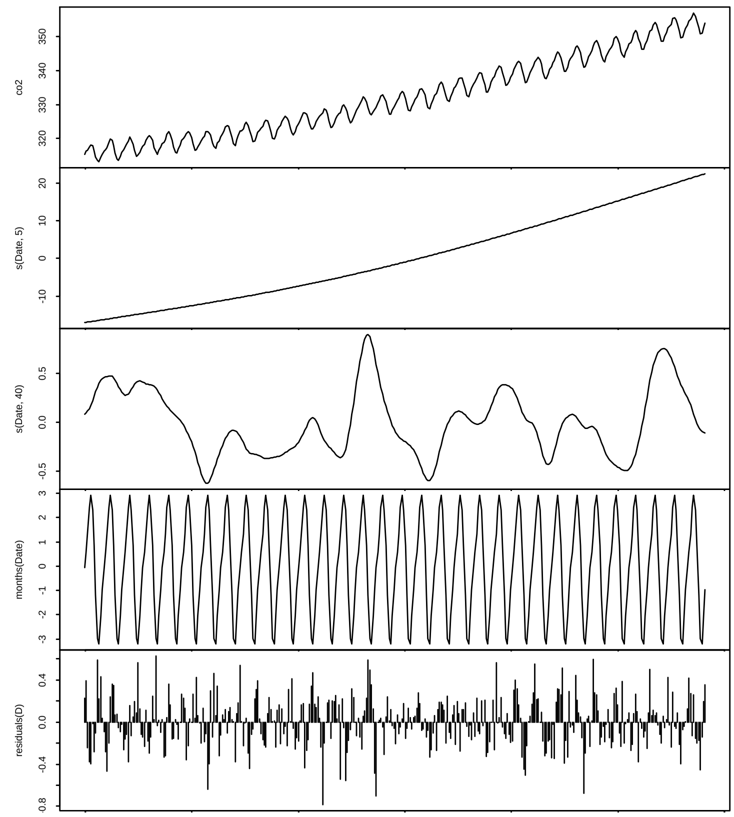

Figure 6: *Generalized additive model decomposition of* `co2`

References

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

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

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

[Cleveland et al., 1990] 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, 1992] Goldman, A. I. (1992). Eventcharts: Visualizing survival and other timed-events data. *The American Statistician*, 46(1):13–18.

[Hastie and Tibshirani, 1990] Hastie, T. J. and Tibshirani, R. J. (1990). *Generalized Additive Models*. Chapman and Hall, London.

A Specifying Functions for Date Formatting

The simple mechanism describe in Section 2 to specify formatting styles is adequate for many situations, but there will be occasions that will not suffice. Formatting is relevant both for reading and printing. 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 <- c("jan", "feb", "mar", "apr", "may", "jun",
               "jul", "aug", "sep", "oct", "nov", "dec")
      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)
      d <- as.character(out$day)
      m <- out$month
      y <- as.character(out$year - 1900)
      mon <- c("jan", "feb", "mar", "apr", "may", "jun",
               "jul", "aug", "sep", "oct", "nov", "dec")
      mon <- mon[m]
      paste(d, mon, y, 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))
```

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
```