Dealing with time in IDL using the JHU/APL/S1R library

The intrinsic IDL tools for dealing with time have been greatly extended by additional routines available in the JHU/APL/S1R IDL library. This document describes these extended capabilities. The library is always evolving, this description is for June 1994.

Obtaining the JHU/APL/S1R IDL library

The JHU/APL/S1R IDL library is available by anonymous ftp as follows.

```
ftp fermi.jhuapl.edu
   login: anonymous
   password: enter your email address
cd pub/idl
get readme.txt
bye
```

Following the instructions in the `readme.txt` file to get the actual library.

Categories of time support software

Routines that deal with time fall into several categories:

- Finding the current time
- Timing program execution
- Working with dates and times
- Working with time series data

These categories are described in following sections.

Finding the current time

The current time is useful to know for a number of purposes. The routines described here are:

`systime()`	Built-in routine to return a string with the current time and date.
`gmt_offsec()`	Return the number of seconds between local and Universal Time.
`lmst()`	Return the Local Mean Sidereal Time.
`sunclock`	Show time and map of earth showing daylight, twilights, and night.

Current time

IDL has a built-in routine to return the current time:

```
print,systime()
   Thu May 19 10:55:34 1994
```

The returned result is a string containing the current date and time. It is useful for documentation, but not very useful for any kind of computation in this form. Tools for dealing with date/time strings are described later.

Universal Time

Universal time differs from local time by an amount depending on the local time zone and whether or not daylight savings is in effect. The JHU/APL/S1R library routine `gmt_offsec` returns the time difference in seconds:

```
print,gmt_offsec()
     14400.0
```

The built-in help for this routine is:

```
x=gmt_offsec(/help)
 Return offset in seconds from local time to GMT (UT).
 off = gmt_offsec()
   off = offset in seconds from local time to GMT (UT).
 Note: Times are rounded to the nearest 1/4 hour.
   This should handle any time zone, even fractional ones.
```

Routines are described in a later section that make it easy to convert between date/time strings and numeric values, so the returned offset can be used to obtain the actual date and time for the current UT.

Local Mean Sidereal Time (LMST)

The local mean sidereal time is the Right Ascension of the local meridian in the sky and is of importance in astronomy. The JHU/APL/S1R library function `lmst` gives this value:

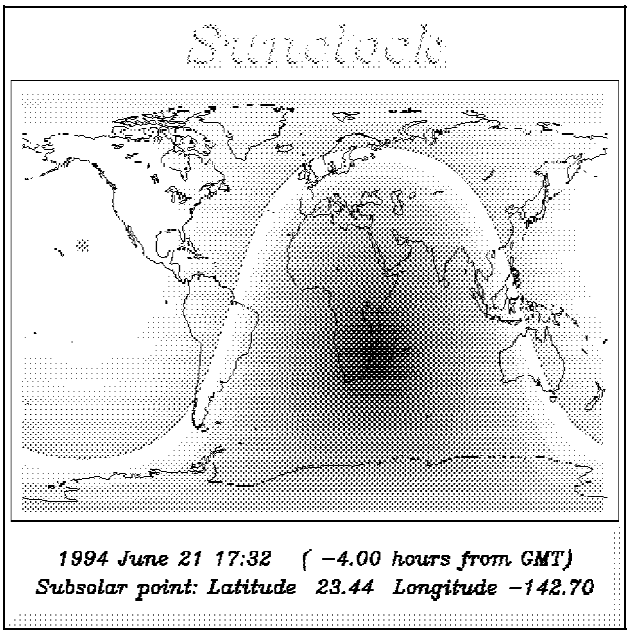
```
x=lmst(/help)
 Give local mean sidereal time.
 st = lmst( jd, ut, long)
   jd = Julian Day (starting at noon).          in
   ut = Universal time as fraction of day.      in
   long = observer longitude (deg, East is +). in
   st = sidereal time as fraction of day.       out
```

The Julian Day may be obtained from the year, month, day or a date/time string and will be discussed in a later section.

Map

The routine `sunclock` may be used to draw a map of the earth showing the point at which the sun is directly overhead, and daylight, twilight, and nighttime areas:

```
sunclock,/help
 Show world map with day, twilight, and night.
 sunclock, plat, plong
   plat = map projection central latitude (deg).
   plong = map projection central longitude (deg).
     West longitude < 0: 90 W = -90.
 Keywords:  map_set keywords may be given to control the map
   display.
   TIME=dt_tm  Specify a date and time.
     Format is flexible but requires month name (3 char min).
     Default is current system time. Ex: "1994 jun 22 13:45"
   ZONE=n  Specify a time zone as hours ahead of GMT.
     Default is local time zone.  Ex: zone=-4 for EDT.
     Only needed if in another time zone.
   /USA plots U.S. states.
 Notes: The meaning of the colors is as follows:
   Subsolar point: Red.
   Daylight: Yellow-Orange.
   Sun rise/set line: Red.
   Twilights: Pink, Blue, and Darker Blue bands.
     Civil:        Sun is   0 to  -6 degrees below horizon.
     Nautical:     Sun is  -6 to -12 degrees below horizon.
     Astronomical: Sun is -12 to -18 degrees below horizon.
```



Various map projections may be used. The example shown is the default. The original image had smooth color shading. In this example the red sun symbol is plotted just east of Hawaii. The color is bright yellow near the sun, smoothly grading to orange at the sun rise/set curve (which is red). The next three bands are Civil (pink), Nautical (blue), and Astronomical (darker blue) twilights.

Another interesting map projection is orthographic. The central latitude and longitude may be given to view the globe from any position.

Timing program execution

Timing program execution is useful for comparing various techniques to see which is faster. For such purposes times are needed at a fairly good resolution. Two routines are of interest:

`systime()`	Built-in systime routine.
`timer`	Timer utility.

systime
The built-in function `systime` with an optional non-zero argument gives such times:

```
print,systime(1)
   7.6936117e+08
```

The returned value is double precision and is the number of seconds since January 1, 1970. Timing a piece of code is easy:

```
t = systime(1)
... code to time ...
print,systime(1)-t
```

timer
This is a utility based on `systime`. It allows multiple timers to be in operation.

```
timer,/help
 Measure elapsed time between calls.
 timer, [dt]
   dt = optionally returned elapsed time in seconds.    out
 Keywords:
   /START  starts timer.
   /STOP   stops timer (actually updates elapsed time).
   /PRINT  prints timer report.
   NUMBER = n. Select timer number to use (default = 0).
      Timer numbers 0 through 9 may be used.
   COMMENT = cmt_text. Causes /PRINT to print:
     cmt_text elapsed time: hh:mm:ss (nnn sec)
 Notes:
  Examples:
  timer, /start  use this call to start timer.
  timer, /stop, /print, dt   use this call to stop timer
    and print start, stop, elapsed time.  This example also
    returns elapsed time in seconds.
  Timer must be started before any elapsed time is available.
  Timer may be stopped any number of times after starting once, and
  the elapsed time is the time since the last timer start.
  timer, /start, number=5   starts timer number 5.
  timer, /stop, /print, number=5   stops timer number 5
  and prints result.
```

Working with dates and times

A rather large number of routines fall into this category. Both dates and times are commonly specified using a mix of units instead of a single unit. Dates use years, months, and days, with years and months each containing a variable number of days. Times use hours, minutes, and seconds, and are simpler since the size of the units do not vary. Dates and times may be mixed as, for example, in the output of `systime()`.

Dealing with dates is further complicated because several systems are in use. The most human readable uses year, month, and day in some order. Other systems are day number of the year (dn), and Julian Day (jd) (these are not the same). Software should be flexible enough to allow dates using year, month, and day to have any order for these components. One valid constraint is to require that the month be specified by name, using at least three letters. Routines described below that deal with date as a string make this requirement. Other routines allow the year, month, and day to be specified as separate numeric values. Routines are provided to work with weekdays.

Some notes on routine naming conventions might be useful. These routines were not designed from scratch as a set. That would certainly have been desirable, but they were written as needed, and the style hopefully improved over time. Still, certain patterns have emerged. For example, a number of routines convert from one date system to another. The names often reflect this by being in the form: input2output. Often this type of routine comes in pairs where one routine is the inverse of the other; sometimes there are one or more closely relate variants. An example is the routine `date2ymd` which converts a date from something like '8 Oct, 1994' to the numeric values of 1994, 10, 8. The inverse routine is `ymd2date`.

In the following routines, the term *date* refers to a form that gives only the day but not a time on that day. Dates may be given in several forms such as year, month, day in numeric values; year and day of year; Julian Day as a long integer; or a date string. The format of a date string is flexible except that the month must be month name. Dashes, commas, periods, or slashes are allowed. Some examples:

23 sep, 1985
sep 23 1985
1985 Sep 23
23/SEP/85
23-SEP-1985
85-SEP-23
23 September, 1985.

The term *date/time* string refers to a string containing both a date and time. For example:

Fri May 27 17:11:00 1994

is a date/time string. The components may be in any order. The day of the week may be included but is not used (it is redundant anyway). Date/time strings are handled correctly as dates by the routines that deal only with date strings. If time is not given it is assumed to be 0:00:00. In most cases the full 4-digit year is required (not just the 2-digit year).

The term *time only* refers to just hours, minutes, and seconds. This may be in one of several forms such as a string hh:mm:ss; the numeric values h, m, s; the number of seconds after midnight as a number; or in general, the number of seconds after some known starting point. Time only, as in the form hh:mm:ss, does not completely specify by itself an actual point in time, but a time interval. In practice the starting point is often known.

The term *complete time* refers to a time system that specifies a single definite point in time. Several example forms are date/time strings; Julian Days which include the fractional part of the day; and *Julian Seconds*, a new form discussed below. Julian Days as used in this library are long integers so are dates and not complete times.

A number of routines are related to calendar type functions such as finding the number of days in a certain month, or finding the day of the week.

As a guide to the routines in this section the one line descriptions are given below. An indication is given on what type of routine each is: (f) means function, and (p) means procedure. Each routine has built-in help. It may be displayed using the following IDL commands (xxx is routine name):

For procedures: `xxx, /help`
For functions: `help,xxx(/help)`

Date only
ymd2jd (f) = From Year, Month, and Day compute Julian Day number.
jd2ymd (p) = Find year, month, day from julian day number.
date2ymd (p) = Date text string to the numbers year, month, day.
ymd2date (f) = Convert from year, month, day numbers to date string.
jd2date (f) = Convert a Julian Day number to a date string.
jd2str (p) = Find year, month, day strings from julian day number.
ydn2date (f) = Convert year and day of the year to a date string.
ydn2md (p) = Convert from year and day number of year to month and day of month.
ymd2dn (f) = Convert from year, month, day to day number of year.
xdate (p) = Widget based date selection tool.

The most used routines here are probably `ymd2jd` and `jd2ymd`. Some examples:

```
print, ymd2jd(1994,6,22) gives 2449526.
jd2ymd, 2449526,y,m,d
print,y,m,d gives 1994   6   22
```

Complete time
dt_tm_tojs (f) = Convert a date/time string to "Julian Seconds".
dt_tm_fromjs (f) = Convert "Julian Seconds" to a date/time string.
ymds2js (f) = Convert to year, month, day, second to "Julian Seconds".
js2ymds (p) = Convert from "Julian Second" to year, month, day, second.
dt_tm_mak (f) = Set up a time and date string from reference JD and offset.
dt_tm_brk (p) = Break a date and time string into separate date and time.
dt_tm_chk (f) = Check date & time string that both parts exist & are valid.
dt_tm_dif (f) = Find time interval in seconds between two date/time strings.
dt_tm_inc (p) = Increment a date/time by a given number of seconds.

The most used routines here are probably `dt_tm_tojs` and `dt_tm_fromjs`. Some

simple examples:

```
js = dt_tm_tojs('8:39 22 Jun, 1994')
print,js  gives -1.7441046e+08
print,dt_tm_fromjs(js) gives 1994 Jun 22 08:39:00 Wed
```

The date returned by `dt_tm_fromjs` in the example is in the default format. Extensive time formatting is supported and will be discussed in a later section.

Time only
secstr (f) = Convert a time string to seconds after midnight.
strsec (f) = Convert seconds after midnight to a time string.
sechms (p) = Seconds after midnight to h, m, s, numbers and strings.
gmt_offsec (f) = Return offset in seconds from local time to GMT (UT).
lmst (f) = Give local mean sidereal time.

Calendar related routines
monthdays (f) = Given a year and month returns number of days in that month.
monthnames (f) = Returns a string array of month names.
nthweekday (f) = Julian Day of N'th weekday of month (like 2nd tues).
weekday (f) = Compute weekday given year, month, day.

Time series support
jsplot (p) = Make a time series plot with time in Julian Seconds.
izoom (p) = zoom an image and display with labeled axes (use /js keyword).
cyph (p) = Construct a cycle/phase array from time series data.
veri (p) = Interactive vertical line on screen or plot (use /js keyword).
crossi (p) = Interactive cross-hair cursor on screen or plot (use /js keyword).
timeaxis (p) = Plot a time axis.

Time series support is discussed in greater detail in a later section.

It would be valuable to look at the built-in help and try some of these routines so you will be familiar with them when a need arises.

Time series data

The problem with time

Data sets which cover just a short span of time are easy to deal with by giving the time in seconds or hours as appropriate. Longer data sets may cover days or weeks and cross month or year boundaries. Day of the year is a commonly used time standard, but doesn't handle year crossings. A somewhat related time standard is Julian Day, which is the number of days since 12 noon January 1, 4713 B.C. This is a well known standard and the JHU/APL/S1R IDL library contains a number of routines for dealing with Julian Day number. Julian Days are very commonly used in astronomy. Julian Days give a time reference that makes it easy to compute time intervals and to handle month or year crossings.

But for common scientific use Julian Days have at least two problems. First, they start at 12 noon. This can easily lead to confusion since most other time units start at midnight. Second, since the zero point is so far in the past (this was purposely done) Julian Day numbers have a lot of significant figures used up just to get to our century. For January 1, 1900 the Julian Day is 2415021. So all recent Julian Day numbers carry a lot of dead weight, and it gets worse (but only very slowly) as we move away from the zero point.

A proposed new standard time system

Julian Days work as a time standard, even the number of digits used is not a real problem for computers. But the 12 hour shift and the use of a day as the time unit are not especially desirable features. The 12 hour shift may be ignored but the result is not truly Julian Days which can lead to problems and confusion.

A new time system has been in use in our group since mid-1992 and found to work well. It has been named Julian Seconds and is the number of seconds *after* 0:00 Jan 1, 2000. Currently Julian Seconds (js) are negative values. The number of significant digits needed will decrease until the new century starts. Julian Seconds are double precision and typically have a resolution of about a millisecond over the time interval +/-500 years from the zero point (depending on the computer).

A number of JHU/APL/S1R IDL library routines work with Julian Seconds. They are easily converted to and from more human readable formats. The two most important conversion routines are:

dt_tm_tojs (f) = Convert a date/time string to "Julian Seconds".
dt_tm_fromjs (f) = Convert "Julian Seconds" to a date/time string.

The first, `dt_tm_tojs`, is the simplest to use. It converts an input date/time string to Julian Seconds. For example:

```
js1 = dt_tm_tojs('1994 jan 1 0:00')
js2 = dt_tm_tojs('1995 jan 1 0:00')
dt = js2 - js1
print,dt,dt/86400. gives  31536000.  365.00000
```

The second routine, `dt_tm_fromjs`, is also simple to use, especially with the default format. For example to find 100 days after the first date in the last example note that each day has 86400 seconds. The variable `js1` contains the first date in Julian Seconds. The desired date is:

```
print,dt_tm_fromjs(js1+100*86400)  gives
   1994 Apr 11 00:00:00 Mon
```

Date/time formatting is discussed in a section below.

Other routines that deal with Julian Seconds are:

jsplot (p) = Make a time series plot with time in Julian Seconds.
izoom (p) = zoom an image and display with labeled axes (use /js keyword).

veri (p) = Interactive vertical line on screen or plot (use /js keyword).
crossi (p) = Interactive cross-hair cursor on screen or plot (use /js keyword).
ymds2js (f) = Convert to year, month, day, second to "Julian Second".
js2ymds (p) = Convert from "Julian Second" to year, month, day, second.

The routines `izoom`, `veri`, and `crossi` are general purpose routines not restricted to working only with Julian Seconds. They each allow the keyword `/js` to indicate that the x values are to be considered Julian Seconds. `izoom` is similar to `jsplot` but places a given image within the plot window, the `/js` keyword indicates that one dimension of the image is time. The first four routines above will be described more later.

Formatting date/time strings

The default date/time format used by `dt_tm_fromjs` is an ISO standard and has the units of time in order of decreasing significance. This format is usually helpful but not always exactly what may be desired. A string specifying how to format the result may be given to `dt_tm_fromjs`. The actual formatting is done by the routine `dt_tm_mak` and the format codes are described in its built-in help text. However, since `dt_tm_fromjs` was written `dt_tm_mak` is seldom used directly.

To format a date/time string the keyword `FORMAT` is used to give a format string with format codes representing time or calendar units which will replace these format codes. The format codes represent such items as year, hour, or weekday name. Several special characters are also allowed. Any other characters in the format string are passed directly to the resulting output string. The following substitutions take place in the format string:

```
Y$ = 4 digit year.
y$ = 2 digit year.
N$ = full month name.
n$ = 3 letter month name.
d$ = day of month number.
W$ = full weekday name.
w$ = 3 letter week day name.
h$ = hour.
m$ = minute.
s$ = second.
f$ = fraction of second.
I$ = time interval in days to 2 decimal places.
H$ = time interval in integer hours.
@  = Carriage Return.
!  = Line feed.
```

Only the items desired need be specified in a format string. For example, to format a time:

```
print,dt_tm_fromjs(js1+10000,format='The time is h$:m$:s$')
```

gives: `The time is 02:46:40`

The contents of the format string was passed to the output after substituting the correct values for the format codes. Note the colons in the time were just passed from the format string. An alternate format string:

```
format='The time is h$ hours, m$ minutes, and s$ seconds') gives
The time is 02 hours, 46 minutes, and 40 seconds
```

An example that displays a time interval is:

```
t=dt_tm_fromjs(js1+8000,form='The range is n$ d$, Y$ h$:m$')+$
  dt_tm_fromjs(js1+10000,form=' to h$:m$')  which gives
The range is Jan 1, 1994 02:13 to 02:46
```

Carriage Returns and Line Feeds may be inserted into the string for devices or routines that understand them.

Formatting fractions of a second

Higher time resolution may be obtained using the fraction of a second format code, f$. By default f$ gives 3 decimal places and includes the decimal point. It is normally placed just after the seconds format code, sf, but need not be. No leading 0 is included, if a standalone fraction is desired with a leading 0 simply add it in the format string: 's$ and 0f$'. The number of decimal places may be set using the keyword `DECIMAL=dp` where `dp` is the number of decimal places desired.

Sometimes it makes sense to display the fraction of a second as a real fraction. For example, in video work each field is 1/60 second. The keyword `DENOMINATOR=d`, if specified, directs that the fraction of a second be displayed as a fraction using the given denominator. If `den=60` then f$ will be replaced by a 5 character string, like `07/60`. No leading spaces are added to this fraction so sf is not a good idea in this case; either add a space or a colon (s$ f$ or s$:f$) or use some other layout.

If both `DECIMAL` and `DENOMINATOR` are given `DECIMAL` is ignored. Sometimes a fraction of a second is needed but a custom format doesn't matter. A shortcut keyword, `/FRACTION`, will add the fraction with 3 decimal places to the default format. One of the keywords `DECIMAL` or `DENOMINATOR` may be used with `/FRACTION` to control the format of the fraction, f$.

Some example format strings and their results:

```
form='h$:m$:s$'            02:46:40
form='h$:m$:s$f$'          02:46:40.123
form='h$:m$:s$ f$'         02:46:40 .123
form='h$:m$:s$ and 0f$'    02:46:40 and 0.123
form='h$:m$:s$f$'          02:46:40.1234        with dec=4
form='h$:m$:s$ f$'         02:46:40 123/1000    with denom=1000
```

Plotting time series data

Time series data plots show how values vary through time over some interval. Often these values are scalars, like temperature, or the components of a vector, like wind speed and direction. In some cases the actual time is unimportant, for example, what is the velocity of a cannon ball dropped from a tower? But for other cases actual time is very important, such as

what was the wind speed throughout the day on 21 July, 1992 at a certain place? The JHU/APL/S1R library routine `jsplot` may be used to plot time series data when a time axis is desired. Ordinary plot axes are surprisingly complex, time axes are even more complex for several reasons. The labeled tick marks must be selected carefully to be useful. This is complicated because of the mixture of units: seconds, minutes, hours, days, months, and years. The lengths of months vary from each other and leap years add to the interest. Evenly spaced tick marks are not the best if more than one month is involved, it's better to make their positions depend on the month boundaries. The routine `jsplot` handles the time axes and also the problem of missing data: there may be times for which no data is available between times with data. All the normal plot options, such as color, character size, linestyle, plot symbols, and so on are also handled by `jsplot` (requires a version of IDL supporting the special keyword `_EXTRA`). An added problem that `jsplot` must deal with is that IDL does not handle double precision data in its plots, it is internally converted to floating point. To see this try the following:

```
x = findgen(1000)    ; Make an x array.
y = (x/1000)^2       ; Make a y array.
plot,x,y             ; Plot.  No problem.
plot,x+1d8,y         ; Add an offset.  Plot is corrupted.
plot,x+1d9,y         ; Even worse.
```

`jsplot` takes as the x array an array of Julian Seconds which are double precision values and typically not close to 0. Trying to directly plot a value against time in Julian Seconds gives a corrupted plot as explained. The solution is simple, offset the times to start at 0:

```
plot,js-js(0),y
```

which is what `jsplot` does internally while keeping track of the offset. This internal offset may be returned through the keyword `OFFSET` and used to do additional overplots:

```
jsplot, js, y, offset=off   ; Make a time series plot.
ver, js(9)-off              ; Plot vertical line for js(9).
```

The built-in help for jsplot is:

```
Make a time series plot with time in Julian Seconds.
jsplot, js, y
  js = Time array in Julian Seconds.   in
  y  = Data to be plotted.             in
Keywords:
  GAP=gs  Gap in seconds.  This or greater time difference
    will cause a break in the curve.  If a plot symbol
    is desired use a negative psym number to connect pts.
  CCOLOR=cc     Curve and symbol color (def=same as axes).
  XTITLE=txt    Time axis title text.
  XTICKLEN=xtl  Time axis tick length (fraction of plot).
  XTICKS=n      Suggested number of time axis ticks.
    The actual number of tick marks may be quite different.
  FORMAT=fmt    Set date/time format (see timeaxis,/help).
  LABELOFF=off  Adjust label position (see timeaxis,/help).
  MAJOR=mjr     Major grid linestyle.
  MINOR=mnr     Minor grid linestyle.
  TRANGE=[js1,js2] Specified time range in JS.
  OFF=off       Returned JS of plot min time. Use to plot
```

```
    times in JS: VER, js(0)-off.
  /OVER         Make overplot.  oplot,js-off,y also works
    but will not handle any gaps.
  Any plot keywords will be passed on to the plot call.
Notes:  Julian seconds are seconds after 0:00 1 Jan 2000.
  See also dt_tm_tojs(), dt_tm_fromjs(), ymds2js(), js2ymds.
```

A few notes may be useful. The keyword XTICKS may be used to control how many tick marks appear on the time axes. The way this keyword works is not the same as for ordinary IDL plots where it specifies exactly how many ticks to use. In jsplot XTICKS is a suggestion, not an exact number. Since tick mark spacing may not even be constant on a time axis an exact number of tick marks is not as reasonable. jsplot trys to pick good tick marks that fit in with the time period covered by the axis and takes the desired number of tick marks into account.

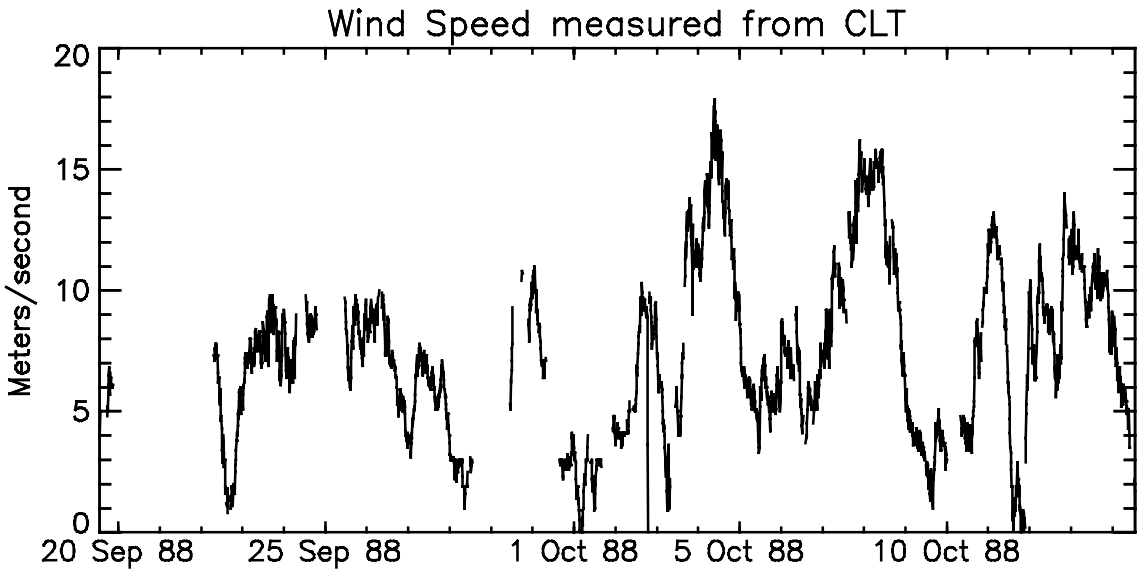
The time axis tick mark labels may be formatted as described in the section **formatting date/time strings** above. Time axis tick labels may have more than one line. This is done by adding a carriage return character between lines. For example, try the following:

```
js1=dt_tm_tojs('1992 july 28')        ; Make a time range.
js2=dt_tm_tojs('1992 aug 3')
js=maken(js1,js2,200)
y=makey(200)                          ; Make up some data.
jsplot,js,y                           ; Default time plot.
jsplot,js,y,form='n$ d$'              ; Add better labels.
jsplot,js,y,chars=2,form='n$ d$'      ; Bigger labels.
jsplot,js,y,chars=2,form='n$@d$'      ; Two line labels.
```

As mentioned in the help text, using jsplot with /OVER to overplot another curve will allow gaps to be handled using the GAP keyword. The GAP keyword may be used to specify that any gaps longer than that length will give a break in the plotted curve. A minimum gap size is specified in seconds.

An example time series plot was made using the following IDL statement:

```
jsplot,js,ws,max=20,gap=1800,yran=[0,20],pos=[.15,.15,.9,.5],$
  ytitle='Meters/sec',titl='Wind Speed measured from CLT',$
```



Displaying time series image data

For scalar data an ordinary time series plot works ok. For vector data, such as wind speed and direction, two time series plots can be used to show the data. An alternate way to display a vector time series is to plot a series of line segments representing the vector but displaced along a time axis. This has problems if the vector is oriented along the time axis. For higher order data, such as a spectrum it makes sense to display the data as an image. For a spectrum the independent variable is frequency and the dependent might be something like power. To display a time series of spectra let time vary along the x axis, frequency along the y and let power be indicated by the image value (color). Such a time series image may be displayed using the routine `izoom` which is described next.

The JHU/APL/S1R library routine `izoom` displays images with labeled x and y axes. This routine displays a 2-d array z which has been scaled for the display. Let array z be nx × ny. Also required is a 1-d array x which has nx values which are the x coordinates of every column in z, and a 1-d array y which has ny values which are the y coordinates of every row in z. One special case takes the x array as Julian Seconds and displays a time axis. The built-in help for `izoom` is:

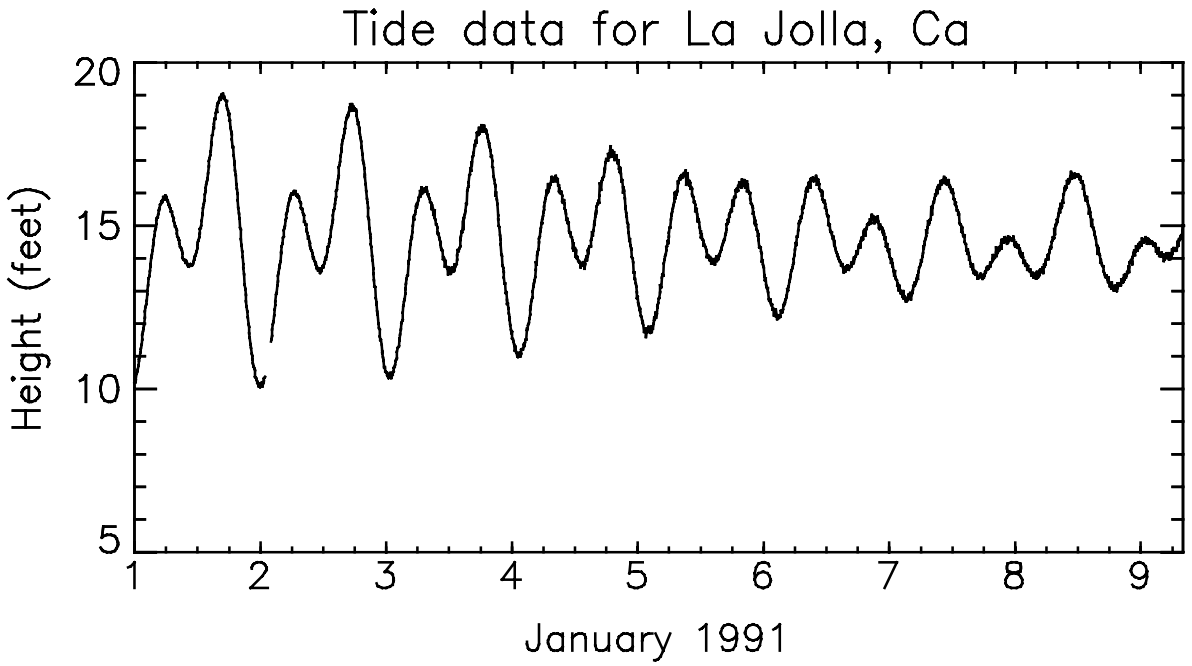
```
Zoom an image and display with labeled axes.
izoom, x, y, z
  x = 1-d array of x coordinates for every column in z.  in
  y = 1-d array of y coordinates for every row in z.     in
  z = 2-d byte scaled image array.                       in
Keywords:
  XRANGE=xran  X range to zoom. (def=all).
  YRANGE=yran  Y range to zoom. (def=all).
    Out of range values are clipped to the valid range.
  /INTERP means do bilinear interpolation (else pixel rep).
```

```
    /JS means x is time in Julian Seconds (seconds after
      2000 Jan 1 0:00).  Gives a date/time axis.
    OFFSET=off  returned time offset when used with /JS.
      Same as for jsplot.
    Any other keywords are passed on to the plot call used
    to display the axes (like TITLE, CHARSIZE, POSITION, ...).
  Notes: By default entire image is displayed.  XRANGE and
    YRANGE display a subset of the image.
    An example use might be: x=array of longitudes,
    y=array of latitudes, z=array of scaled elevations.
    May use movbox to return the data ranges of a selected
    area, then call izoom to zoom it:
      movbox,ix,iy,dx,dy,xran=xran,yran=yran,/noerase
      izoom,x,y,z,xran=xran,yran=yran
```

If x is time in Julian Seconds use the `/JS` keyword in `izoom`. A subsection of the image is easily zoomed up to fill the plot using the keywords `XRANGE` and `YRANGE`. An example result from izoom is shown in the next section.

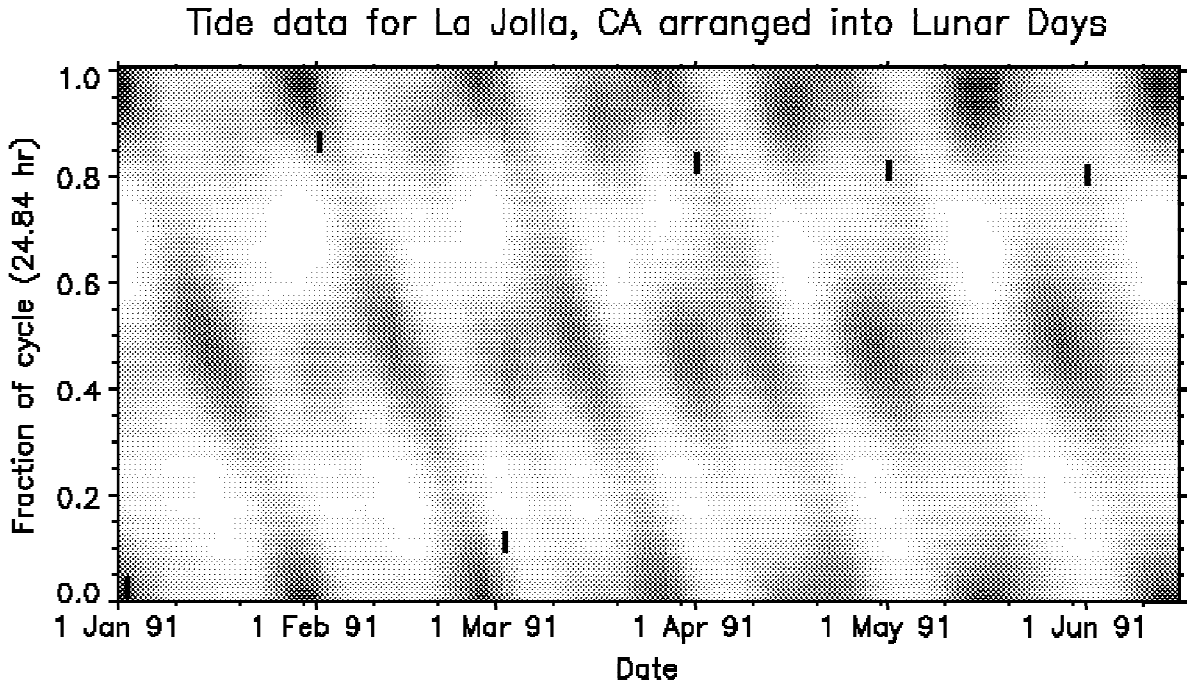
Dealing with periodic data

Some time series data is periodic in nature or has components of one or more periods. Sometimes such data may be usefully displayed compactly as an image if it is arranged properly. An clean example of such data is tide height data for La Jolla, CA. An ordinary time series plot of a small part of the data in hand is shown in the next plot.



For this plot the month and year were factored out of the time labels and placed in the x axis title. The format used for the time labels was simply d$, the month day.

If the tide height data are rearranged into a 2-d array where each column of the array covers one period of some cycle it may show the data in a new light. There are a number of periods that occur in tide data, the two dominant ones are the solar period of 12.00 hours, and even more important, the lunar period of 12.42 hours. The following image shows the data arranged into lunar days of 24.84 hours each.



In the above image brightness represents height of the water. The dominant feature of the image is the modulated horizontal banding which are the lunar tides. For this location two high tides occur every lunar day of 24.84 hours. A vertical cut through the image shows that often the heights of the two high tides differ. Also very noticeable are the diagonal bands. Note that they diagonally cross the image with a horizontal period close to one month. These are due to the solar tides which have a period of 12.00 hours. The image was arranged for a period of 24.84 hours, any other period will appear as diagonal features in the image. Because the moon's orbital period around the earth is about one month the solar tides and lunar tides come in and out of phase with this same period. The small black marks are missing data due to tape recorder tape changes. Such missing data have no effect on building the cycle/phase image and, unless they are severe, they have little effect on interpreting the resulting image.

The cycle/phase image was constructed using the JHU/APL/S1R library routine `cyph`. The built-in help for `cyph` is:

```
Construct a cycle/phase array from time series data.
cyph, data, time
  data = time series data.              in
  time = time tag of each data point.   in
Keywords:
```

```
  CPOUT=z  Returned reformatted data image.
  TOUT=t   Returned time for bottom line image.
  OFFSET=frac  Given starting offset into data as fraction
    of time slice size (def=0).
  SLICE=dx  Given slice size in samples (may be fractional).
```

Time does not have to be in Julian Seconds and may be nothing more than an index number. The desired period is entered using the keyword `SLICE` and may be fractional.

Interactive measurement of time series plots on the screen

There are two JHU/APL/S1R library routines of interest in measuring positions on screen time series plots, `veri` and `crossi`. `veri` is an interactive vertical line that gives a continuous readout of its position. `crossi` is an interactive cross-hair cursor that also gives a continuous readout of its position. Both routines have the keyword `/JS` which displays the x coordinate as a date/time string if `jsplot` or `izoom` with `/js` were used to make the plot. The keyword `/MAG` for `crossi` gives a magnified view of the area around the cursor and makes it easy to position the cursor to a single pixel on the screen. The built-in help for these two routines is:

```
Interactive vertical line on screen or plot.
veri, [x]
  x = X coordinate of line.      in, out
Keywords:
  /DATA   Causes data coordinates to be used (default).
  /DEVICE Causes window device coordinates to be used.
  /NORMAL Causes normalized coordinates to be used.
  COLOR=c Set color of line (ignored for /XOR).
  LINESTYLE=s Line style.
  XFORMAT=xfn  These keywords are given names of functions
  YFORMAT=yfn  that accept the numeric value of x or y
    and return a corresponding string which is displayed
    in place of the actual value.  For example, Julian
    days could be displayed as a date with jd2date.
  XSIZE=xs, YSIZE=ys  Coordinate display widths.
  /JS  Means X axis is time in Julian seconds.
  /NOSTATUS   Inhibits status display widget.
  SETSTAT=st  May use the same status display widget on
    each call to veri (stays in same position).
    On first call: the status widget structure is returned.
    Following calls: send st.  Must use with /KEEP.
    To delete status display widget after last box1 call:
      widget_control,st.top,/dest (or drop /KEEP)
  /KEEP   Do not delete status widget on exit.
  /XMODE  Means use XOR plot mode instead of tvrd mode.
  INSTRUCTIONS=t  String array with exit instructions.
    Default: Press any button to exit.
  /DIALOG Means give an exit dialog box.
  MENU=m  A string array with exit dialog box options.
    Def=Exit. An option labeled Continue is always added.
  EXITCODE=x Returns exit code.  Always 0 unless a dialog
    box is requested, then is selected exit option number.
  BUTTON=b   Returned button code: 1=left, 2=middle, 4=right.
```

```
Note: data coordinates are default.
  X may be set to starting position in entry.


Interactive cross-hair cursor on screen or plot.
crossi, [x, y]
  x = X coordinate of line.      in, out
  y = Y coordinate of line.      in, out
Keywords:
  /DATA   Causes data coordinates to be used (default).
  /DEVICE Causes window device coordinates to be used.
  /NORMAL Causes normalized coordinates to be used.
  COLOR=c Set color of line (ignored for /XOR).
  LINESTYLE=s Line style.
  MAG=m   Magnification for an optional magnified window.
    Setting MAG turns window on. /MAG gives magnification 10.
  SIZE=sz Mag window approx. size in pixels (def=200).
  XFORMAT=xfn  These keywords are given names of functions
  YFORMAT=yfn  that accept the numeric value of x or y
    and return a corresponding string which is displayed
    in place of the actual value.  For example, Julian
    days could be displayed as a date with jd2date.
  XSIZE=xs, YSIZE=ys  Coordinate display widths.
  /JS  Means X axis is time in Julian seconds. Example:
    x=maken(-2e8,-1.9e8,200) & y=maken(20,30,200)
    z=bytscl(makez(200,200))
    izoom,x,y,z,/js
    crossi,/js
  /NOSTATUS   Inhibits status display widget.
  SETSTAT=st  May use the same status display widget on
    each call to crossi (stays in same position).
    On first call: the status widget structure is returned.
    Following calls: send st.  Must use with /KEEP.
    To delete status display widget after last box1 call:
      widget_control,st.top,/dest (or drop /KEEP)
  /KEEP   Do not delete status widget on exit.
  /XMODE  Means use XOR plot mode instead of tvrd mode.
  INSTRUCTIONS=t  String array with exit instructions.
    Default: Press any button to exit.
  /DIALOG Means give an exit dialog box.
  MENU=m  A string array with exit dialog box options.
    Def=Exit. An option labeled Continue is always added.
  EXITCODE=x Returns exit code.  Always 0 unless a dialog
    box is requested, then is selected exit option number.
  BUTTON=b   Returned button code: 1=left, 2=middle, 4=right.
Note: data coordinates are default.
  X and Y may be set to starting position in entry.
```

Ray Sterner
The Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20723-6099

sterner@tesla.jhuapl.edu
North Latitude: 39.16
West Longitude: 76.90