User's Guide for wrscilib V1.00

Idea, concept and implementation in Scilab by Werner Roethlisberger Thailand

Genereously supported by Mr. Thanisorn Chivatee Radej (Samran Tienna) CEO Thai Master Inter Law and Accounting (Thailand).

To contact me and say thank you as well as for your complains you may send mails to wrscilib@yahoo.com. Make sure that there is always the subject **"wrscilib"** added into the subject line of your mails. Any other mails without this subject will just be deleted! And please understand and do not ask for support!

This document was written with:
***Open Office** the original is @ http://www.openoffice.org/*
*One of a derivate I also often use **Go Open Office** can be found @ http://go-oo.org/*
*Another excellent one is **Oxygen Open Office** can be found @ http://sourceforge.net/projects/ooop/files/*

General Information

This library makes some calculations hopefully little easier or even possible and for that it provides few functions which some of us may missing in Scilab.

There are not any kind of validating applied out of this library. Thus made it possible to calculate even Gregorian Dates direct and correct e.g.

```
-->jdn2gd(gd2jdn([1960 2 26+4])) > 1960.    3.    1.
```

This implies that there are no additional error messages out of this library at all. If in case a calculation is not possible from the mathematical point of view, such cases will be handled and prompted by Scilab solely.

Trap warning 1 (missing right parenthesis)

```
-->cities=locofcity('usa ');val2(cities(2,2)
                                                    !--error 3
Waiting for right parenthesis.
```

Trap warning 2 (nested function calls without parentheses)
You may use the function gdnow (without parentheses) but NOT in nested cases:

```
-->gd2jdn(gdnow)
 !--error 246
Function not defined for given argument type(s),

  check arguments or define function %mc_size for overloading.

at line       2 of function gd2jdn called by :
-->gd2jdn(gdnow)
```

Correct use with empty parantheses so-called voids:

```
-->gd2jdn(gdnow())
 ans  =

    2455111.5
```

Trap warning 3 (list vs. matrix)

```
-->gd2jdn(2010, 1, 1)
                   !--error 58
Wrong number of input arguments:

Arguments are :

 _gymd
```

Correct use with a matrix:
Most, but unfortunately not all, functions in this library uses parameters in form of a matrix. In case of Scilab it doesn't matter whether or not the matrix is comma separated.

```
-->gd2jdn([2010, 1, 1]) or gd2jdn([2010 1 1])
 ans  =

    2455198.
```

Therefore it's highly recommended to use empty parentheses all the time to avoid stepping into such traps!

```
-->tnow()
 ans  =

    10.    26.    43.
```

Trap warning 4 (omitting parameters)
In most cases when parameters are optional, such constructions as
`gd2aged([2504 01 26],,['b'])`
will be formed up very often. As you can see under **gd2aged(gregorian date [[+|-]year1,month1,day1, [[+|-]year2,month2,day2], ['g'|'b'])** takes the second parameter as optional, which means instead to type the actual date you just omit this one as a “empty” parameter.

Many algorithms from this library provides a unlimited validity but they are limited to the implemented system boundaries of course. Using them up to the limit of the system boundaries may produce some unexpected side effects which may influencing the results. Please read Scilab’s handbook or the help for further details about this matter.

Few of the algorithms doesn't provide any information of its validity but whenever possible you'll find a statement of validity. However, in most cases of the navigation calculations algorithms may be valid in the range between 1900 and 2100. Furthermore some algorithms doesn't provide information about its author. **All algorithms are carefully tested, however, nor the author of this library as well as not any other individuals or any company will under no circumstances take any responsibilities for any kind of mistakes, errors or damages.**

All functions can be nested as usual in Scilab, but not forged to use empty brackets whenever no arguments will be given.

Many functions provides possibilities to give matrices for calculations even for date and time calculations. Unfortunately there are still some functions which does NOT yet allow using matrices.

For preventing problems with user variables, all internal library variables uses a leading underscore “_x”.

Check your wrscilib version:

```
-->libver
File: wrscilib.sci:
- Collection of 'Scilab' functions v1.00
- Copyright (c) Sep/2K5 by Werner Roethlisberger (Thailand)
- All rights and copyrights (c) are reserved Sept/2K5 by Thai Master
Watches(TM)
- All rights reserved under the copyright laws of Kingdom of Thailand
- Any commercial use without permission is strictly prohibited!
- Any permissions from other authors have no effect to our copyrights,
algorithms, ideas and concept
- mail: wrscilib@yahoo.com for some response!
- The author and the company Thai Master will not take any
responsibility for any kind of errors, problems and damages!
```

Installing the Library

Since there is not yet a installation finished out of this library. Just copy the contents of the wrlibsci2010.7z to any location on your harddisk.

After that write the few lines below into scilab.start rather somewhere to end of it.
Well, this not yet a sophistic installation procedure I know. Maybe I will change this sometime.

```
// Loading WRLIBSCI01
//
xlib=lib('d:\datas\scilab5\wrlib01'+'/')
chdir('d:\datas\scilab5\wrlib01\')
//
//
```

Be aware! This is not yet a sophisticated way to get access to the library. Because in some cases when the path is misspelled or otherwise wrong, Scilab may hang (stop working)!

A newest version of this handbook should always be included in the latest package. In case is not let me know.

Time Calculations

hms2jtn([[+|-]h m s;…;m])
Convert a consistent matrix of [+|-]hms to Julian Time Number ([+|-]jtn) sometimes also called serial time.
Example: Convert time 3:24:5 hms2jtn([3 24 5]] = 0.1417245370370370 (jtn).

jtn2hms([[+|-]jtn;…;m])
Convert a consistent matrix of [+|-]jtn Julian Time Number to time [+|-]hms.
Example: Convert JTN jtn2hms(0.1417245370370370) = 3. 24. 5. (hms).

hms2min([[+|-]h m s;...;m])
Covert a consistent matrix of [+|-]hms to [+|-]minutes
Example: Convert time hms2min([3 24 5]) = 204.08333333333334 (minutes).

min2hms([[+|-]minutes;…;m])
Convert a consistent matrix of [+|-]minutes to [+|-]hms.
Example: Convert minutes min2hms(204.08333333333334) = 3. 24. 5. (hms).

hms2dec([[+|-]h m s;...;m])
Convert consistent matrix of [+|-]hms to decimal time [+|-]dd.mmss
Example: Convert 10° 27’ 36 degrees hms2dec([10 27 36]) = 10.459999999999999 (dd.mmss) decimal degrees.

dec2hms([[+|-]dd.mmss;…;m])
Convert consistent matrix of decimal time [+|-]dd.mmss to [+|-]hms.
Example: Convert decimal time dec2hms(10.459999999999999) = 10. 27. 36. (hms).

dec2uhm([[+|-]dd.mmss;…;m])
Convert consistent matrix of decimal time [+|-]dd.mmss up rounded to next minute of time (hm).
Example: Convert decimal time dec2uhm(10.459999999999999) = 10. 28. 0. (hm)

tstr2jtn([“[+|-]h m s”])
Convert time string “HMS” to Julian Time Number (jtn).
Example: Convert time string tstr2jtn("230901") = 0.9645949074074074 (jtn).

jtn2tstr([[+|-]jtn])
Convert Julian Time Number (jtn) to time as a string (“hms”).
Example: Convert Julian Time Number jtn2tstr(0.9645949074074074) = “230901” (“hms”)

tstr2dec([“[+|-] h m s”])
Convert time string “HMS” to decimal time dd.mmss.
Example: Convert time string tstr2dec(“230901”) = 23.150277777777777 (dd.mmss)

dec2tstr([+|-]dec)
Convert decimal time dd.mmss to time string (“hms”).
Example: Convert decimal time dec2tstr(23.150277777777777) = “230901” (“hms”).

uxt2jdn(ut)
Convert UNIX date and time (ut) to Julian Date and Time Number (jdn).
Example: Convert UNIX date and time uxt2jdn(1181811539.0000091) = 2454265.8742939816 (jdn).

jdn2uxt(jdn)
Convert Julian Date and Time Number to UNIX date and time (ut).
Example: Convert jdn jdn2uxt(2454265.8742939816) = 1181811539.0000091 (ut).

gdhr2uxt(optional [gymd,hms])
Convert Gregorian Date (year month day) and Time (hms) to UNIX Date and Time (ut).
Example: Convert ymd, hms gdhr2uxt([2007 6 14, 9 1 2]) = 1181812208.9999959 (ut).
Both parameters date and time are optional. However, any not given parameter will be replaced by systems date or time or both. Be aware of systems date and time since it could differ between systems or they often just wrong set.

uxt2gdhr(optional [ut])
Convert UNIX Date and Time (ut) to Gregorian Date and Time (gdhr).
Example: Convert UNIX Date and Time to Gregorian Date (YMD) and Time (HMS).
uxt2gdhr(1181811539.0000091) = 2007. 6. 14. 8. 58. 59. (YMD HMS).
The only one parameter for the UNIX date and Time is optional. In this case systems date and time will be converted. Be aware of systems date and time since it could differ between systems or they often just wrong set.

hmsdiff([[+|-]hms1 [+|-]hms2;…;m])
Calculate the difference for a consistent matrix of time (HMS) between two times (hms1 hms2).
Example: Calculate the difference hmsdiff([10 9 23 8 5 1]) = 2. 4. 22. (hms). In Scilab it's possible giving matrices with or without separated by comma means both ways are identical hmsdiff([10 9 23, 8 5 1]) or hmsdiff([10 9 23 8 5 1]).

addhms([[+|-]hms [+|-]hms;...;m])
Adds two times of a consistent matrix of n x two times (hms1 hms2).
Examples: Add two times (hms1 hms2) addhms([10 9 23,8 5 1]) = 18. 14. 24. (hms).

sec2day([[+|-]sec;...;n])
Convert consistent matrix of second(s) (ss.ssss) to Gregorian Day(s) (dd.dddd).
Example: Convert second(s) sec2day(86400) = 1. (dd.dddd).

sec2hr([[+|-]sec;...;n])
Convert consistent matrix of second(s) to hour(s).
Example: Convert seconds sec2hr(86400) = 24. (hr).

sec2mins([[+|-]sec;...;n])
Convert consistent matrix of second(s) to minute(s).
Example: Convert seconds sec2mins(3600) = 60. (minutes).

extsec([[+|-]sec;...;n])
Extract second(s) from a consistent matrix of seconds (ss.ssss) to time second(s).
Example: Extract seconds extsec(1) = 84600.

sec2dhms([[+|-]sec;...;n])
Convert second(s) to Gregorian Day(s) and Time in ([dd h m s]).
Example: Convert seconds sec2dhms(84600) = 1. 0. 0. 0.

day2sec([[+|-]days;...;n])
Convert consistent matrix of [+|-]day(s) to second(s)

Example:
```
-->day2sec(1) =  86400.
```

hr2sec([[+|-]hour;...;n])
Convert consistent matrix of time in [+|-]hour(s) to second(s)

Example:
```
-->hr2sec(1) = 3600.
```

mins2seconds([[+|-]mins...;n])
Consistent matrix of time in [+|-]minute(s) to second(s)
Example: Convert mins2seconds(1) = 60.

dhms2sec([days,hour,min,sec;...;m])
Convert consistent matrix of [+|-]day(s) and [+|-]time (hms) to second(s)
Example: Convert dhms2sec([1 0 0 0]) = 86460.

jdn2hr([+|-]jdn,[+|-]tz)
Convert the time part from an Gregorian Julian Day Number [optional time zone (hms)] into time (hms)

Example:
```
-->jdn2hr(2437325.5) = 0.    0.    0.
```

Why is this result zero? Since this library is based on the calculation method of “Joseph Justus Scaliger” JD 0 designates the 24 hours from non to noon on 1st January 4713 BC to noon UTC on 2nd January 4713 BC. The result above means therefore exactly noontime.

tnow([n])
Get time from system clock.

Date Calculation

Notice for implementation of the function below to other environments

[x] = the greatest integer that does not exceed x. e.g.[-1.5]= -2
This is sometimes called the floor function (in C/C++ and in some PPC's)

INT(x) = [x] NOTE: some computer languages have a different definition
E.g. some defines int(-3.25/2)= -1, while MS defines int(-3.25/2)= -2 which is more often called floor.

Depending of expected results of an algorithm, rather use floor(-1.5)= -2 or fix(-1.5)= -1 instead

FIX(x) = the number x without its fraction e.g. Fix(-1.5)= -1
x\y = FIX(x/y) so-called integer division

MOD = NOTE: Some computer languages have different definition see below

NOTICE: This library always tries to provide the expected consistent use of all functions to prevent confusion.

WE DEFINE HERE:
int = int(-1.5)= -1 identical to fix (as usually expected). However, in most MS environments the definition of int is different

fix = fix(-1.5)= -1 identical to int (as usually expected). However, even MS agrees here, which isn't really important.

[x] = floor(-1.5)= -2 (as usually expected). However, in most MS environments the definition is different.

modulo = Scilab provides modulo for both negative modulo and pmodulo for positive or zero modulo.

Mod = To be platform independent the library provides emulations of both Scilab's modulo (mod) and pmodulo (mod2) as well.

UNLIMITED:
The term unlimited means limited by the word size of the implementation environment. This doesn't interfere to algorithms with limited validity e.g. "gd2seclip" (which method is good to 0.01 degrees over the period 1900 to 2100).

JULIAN DAY NUMBER:
The French scholar Joseph Justus Scaliger (1540-1609) was interested in assigning a positive number to every year without having to worry about BC/AD (for BC/AC see below). He invented what is today known as the "Julian Period".
Astronomers have used the julian period to assign a unique number to every day since 1st January 4713 BC. This is the so-called Julian Day (JD), in this library rather called Julian Day Number (JDN) to prevent confusion.

Trap warning: None of the algorithms checks for the Gregorian calendar reform at 15th October 1582

JD 0 designates the 24 hours from noon UTC on 1st January 4713 BC to noon UTC on 2nd January 4713 BC. The Julian period (and the Julian Day Number) must not be confused with the Julian Calendars.

Basic conversion algorithm by Joseph Justus Scaliger enhanced and optimized by Peter Baum "url: vsg.cape.com/~pbaum" some small improvement added, adapted and implemented in Scilab by Werner Roethlisberger clever_comment@yahoo.com.

gd2jdn(gregorian date [[+|-]year;...;n,month,day])
Convert consistent matrix of full qualified Gregorian date(s) (y m d) with [+|-]year unlimited to Gregorian Julian Day Number (jdn). The year 50 will be used as a full qualified year and not as 1950. Because this is not the wrong and limited table calculation. That's why years must always be full qualified. For years before zero a "-" must be added as a prefix e.g. [-1961 1 26]. This indicates a date before the year zero!

Example:
```
-->gd2jdn([1961 1 26]) = 2437325.5
-->gd2jdn([-1961 1 26]) = 1004844.5
```

Be aware of Scilab's default numbering format. Use Scilab's format to change.

jdn2gd(gregorian date [[+|-]jdn;...;n])
Convert consistent matrix of (Gregorian) [+|-]Julian Day Number(s) to Gregorian Date (y m d).
Example: Convert jdn2gd(2437325.5) = 1961. 1. 26. jdn2gd(1004844.5) -1961. 1. 26.

jdn2year(gregorian date [+|-]jdn)
Convert Gregorian Year from Gregorian Julian Day Number(s).

jdn2month(gregorian date [+|-]jdn)
Convert Gregorian Month from Gregorian Julian Day Number(s).

jdn2day(gregorian date [+|-]jdn)
Convert Gregorian Day from Gregorian Julian Day Number(s).

gdnow(gregorian date [n])
Get Gregorian Day from system clock.

jdnow(gregorian date [n])
Get Gregorian Julian Day Number from system clock.

jdnnow(gregorian date [n])
Get Julian Day Number from system clock.

tbnow(thai buddhist date [n])
Get Thai Buddhist date from system clock.

gd2mjdn([[+|-]year;...;n,month,day])
This algorithm does the same as **gd2jdn** does. But it claims to be the original Meeus algorithm from Astronomical Algorithms, 1991, page 63. Jean Meeus gives this algorithm. Adapted for Scilab by Werner Roethlisberger.

mjdn2gd(gregorian date [[+|-]jdn;...;n])
This algorithm does the same as **gd2jdn** does, but it claims to be the original Meeus algorithm from Astronomical Algorithms, 1991, page 63. Jean Meeus gives this algorithm. Adapted for Scilab by Werner Roethlisberger.

gd2cjdn(gregorian date [[+]year,month,day;...;n])
Returns the so-called Modified Julian Date Number also known as the Chronological Julian Day Number since 1858 Nov at 17 00:00h. This algorithm is valid for any Gregorian Date Calendar after 1582 Oct 15. Algorithm by Peter Baum adapted for Scilab by Werner Roethlisberger.

gd2cjdn2(gregorian date [[+]year,month,day;...;n])
Returns the so-called Modified Julian Date Number of days since 1858 Nov at 17 00:00h also known as the Chronological Julian Day Number. The validity is unknown. The method used here is adapted from Montenbruck and Pfleger's Astronomy on the Personal Computer, 3rd Ed, section 3.8. It takes the year, month and day of a Gregorian Calendar Date and returns the Modified Julian Day Number. Consistent matrix (n x 3) of Gregorian Date(s) with [+]year unlimited.

jdn2jdc2k(gregorian date [+|-]jdn)
Returns Centuries since year 2000 from a given Gregorian Julian Day Number.

gd2jdc2k(gregorian date [[+]year,month,day;...;n], [[+|-]h m s],_[[+|-]h m s])
Returns Centuries since year 2000 from Gregorian Date, Time and Time Zone. All parameters are optional for actual system date and time at time zone 0.

gdhr2jdt(gregorian date [[+]year,month,day;...;n], [[+|-]h m s],_[[+|-]h m s])
Calculates Julian Day and Time Number from Gregorian Date, Time and Time Zone.
Notice: The time must explicitly given even for midnight to get correct results since default (optional date and time) results in current date and current time.
For calculation at noon time 12 O'clock must be given in the form
([2007 11 28],[12 0 0],[0]) or ([2007 11 28],[12])

Be aware that the timezone becomes a part of the whole time calculation by using the reverse function jdt2gdhr below.

Examples:
```
-->gdhr2jdt([2007 11 28],[0 0 0],[0]) > 2454432.5
-->gdhr2jdt([1987 8 12],[13 51 0],[-2]) > 2447019.99375
```

jdt2gdhr([+|-]jdt)
Calculates the time in hms from given Julian Time Number. Be aware of rounding problems in all cases, when the Julian Time Number have large leading days. If anyone would like to provide a much more reliable algorithm please let me know.
Notice: The time must explicitly passed even for midnight to get correct results since default (optional date and time) results in current date and current time.
Example: jdt2gdhr(2447019.99375) > 1987. 8. 12. 11. 51. 0.

gd2wdna(gregorian date [[+]year,month,day;...;n])
Calculates the Cardinal Weekday Number from Gregorian Date whereby the corresponding cardinal numbers are 0=Su, 1=Mo, 2=Tu, 3=We, 4=Th, 5=Fr, 6=Sa. Which means a week starts on Sunday.

gd2wds(gregorian date [[+]year,month,day;...;n])
Returns Weekday string from Gregorian Date corresponding to gd2wdna.

gd2iwdna(gregorian date [[+]year,month,day;...;n])
Returns ISO 8601 Weekday Number from Gregorian Date. Corresponding to ISO 8601 the day of week numbers begins with 1=Mo, 2=Tu, 3=We, 4=Th, 5=Fr, 6=Sa, 7=Su. A ISO week starts with Monday.

gd2iwds(gregorian date [[+]year,month,day;...;n])
Returns Weekday string from Gregorian Date corresponding to gd2iwdna.

gd2boy(gregorian date [[+|-]year,month,day;...;n])
Days from the beginning of a Gregorian Year to a given Gregorian Date.

gd2eoy(gregorian date [[+|-]year,month,day;...;n])
Days to the End of a Gregorian Year from a given Gregorian Year.

gdisiwn(gregorian date [[+|-]year,month,day,isoweekday;...;n])
Test if Gregorian Date is ISO8601 weekday. The numerical result is true = 1 if the weekday is ISO 8601 conform otherwise false = 0.

gdissasu([[+|-]year,month,day;...;n])
Test if Gregorian Date is weekend Saturday or Sunday. The numerical result is true = 1 if the weekday is Sa or Su otherwise false = 0.

gdismofr(gregorian date [[+|-]year,month,day;...;n])
Test if Gregorian Date is weekday Monday to Friday. The numerical result is true = 1 if the weekday is Mo to Fr otherwise false = 0.

gdislyr(gregorian date [[+|-]year,month,day;...;n])
Test if Gregorian Date is a leap year. The numeric result is true = 1 if the year is a leap year otherwise false = 0 if the year is non leap year.

gddiff(gregorian date [[+|-]year,month,day;...;n])
Difference in days between two Gregorian Dates.

adday2gd(gregorian date [[+|-]year,month,day,n...;n])
Add or subtract days to a Gregorian Date.
Example: `-->adday2gd([2007 11 24 -1]) > 2007.    11.    23.`

gd2dpm(gregorian date [[+|-]year,month,day...;n])
Days per month of a Gregorian Date.

gd2fwm(gregorian date [[+|-]year,month,day...;n])
First ISO 8601 weekday of a month from a Gregorian Date (a week starts with Monday).

gd2lwm(gregorian date [[+|-]year,month,day...;n])
Last ISO 8601 weekday of a month from a Gregorian Date (a week starts with Monday).

d2lpm(gregorian date [[+|-]year,month,day...;n])
The last day from previous month of a Gregorian Date.

gd2dpy(gregorian date [[+|-]year,month,day...;n])
Days per year of a Gregorian Date.

gd2aged(gregorian date [[+|-]year1,month1,day1, [[+|-]year2,month2,day2], ['g'|'b'])
Age in days from a given Gregorian Date. The second parameter is optional which is the calculation date (date2). If not given the date from the system clock will be used for calculations. The last argument the interval is optional needed to indicate Thai Buddhist Dates 'b' or Gregorian Dates 'g' explicitly. Default here is Greorian Date of course.
Gregorian Date Example: `-->gd2aged([1961 01 26]) > 17099.`
Thai Buddhist Example: `-->gd2aged([2504 01 26],,['b']) > 17099.`

gd2agey(gregorian date [[+|-]year1,month1,day1, [[+|-]year2,month2,day2], ['g'|'b'])
Same as above for gd2aged, but the results will be years instead.

gd2nbd(gregorian date [[+|-]year,month,day, ['g'|'b'])
The next day for a birthday will be calculated by just passing the birthday either as a Gregorian Date or Thai Buddhist Date. The last argument is optional and only needed indicate either, for Thai Buddhist Dates 'b' or Gregorian Dates 'g' explicitly. The result will be: Days until next birthday, the Age, the Weekday, Gregorian Date (y m d) as a list or matrix.

Example: Next birthday (Thai Buddhist date):
```
-->gd2nbd([2504 01 26],['b'])  >  67.    47.    6.    2008.    1.
26
```

The result is: 67 days until next birthday. The age will then be 47. The cardinal weekday is the number 6, which means Sunday (see at gd2wdna) following by the full Gregorian date.

The algorithm applies weekdays in cardinals means;
0=Mo, 1=Tu, 2=We, 3=Th, 4=Fr, 5=Sa, 6=Su

tb2aged(gregorian date [[+|-]tyear1,month1,day1, [[+|-]tyear2,month2,day2], ['g'|'b'])
Here the meaning are same as for gd2aged but the defaults changed to Thai Buddhist Date instead of Gregorian Dates.

tb2agey(gregorian date [[+|-]tyear1,month1,day1, [[+|-]tyear2,month2,day2], ['g'|'b'])
Here the meaning are same as for gd2agey but the defaults changed to Thai Buddhist Date instead of Gregorian Dates.

tb2nbd(gregorian date [[+|-]year,month,day, ['g'|'b'])
Same as for gd2nbd mentioned except that the defaults are for Thai Buddhist Dates of course.

gd2qy(gregorian date [[+|-]year,month,day;...;n])
Calculates the part of quarter of a year from a Gregorian Date. The corresponding result are:

- 1 for 1st Quarter till 31st March
- 2 for 2nd Quarter till 30th June
- 3 for 3rd Quarter till 30th September
- 4.for 4th Quarter till 31th December

gd2qy2(gregorian date [[+|-]year,month,day;...;n])
Calculates same as gd2qy but with a totally different algorithm which may use some more CPU than the other one.

gd2kw(gregorian date [[+|-]year,month,day;...;n])
Calculates the week number based on DIN 1355 for a Gregorian Date.

gd2dkw2(gregorian date [[+|-]year,month,day;...;n])
Calculates the week number based on DIN 1355 for a Gregorian Date. Algorithm by Hans-G. Mekelburg "url: home.nordwest.net/hgm/kalender" adapted and implemented in Scilab by WR.

gd2esu(gregorian date [[+|-]year,month,day;...;n])
Calculates the day for Easter Sunday of a Gregorian Year.
Algorithm by J.M. Oudin from the year 1940.
From Étude sur la date de Pâques. in: Bulletin Astronomique, Bd. XII, S. 391-410 Last publication in 'Explanatory Supplement to the Astronomical Almanac (c) by P. K. Seidelmann (1992)'. Seen at http://www.ortelius.de/kalender/calc_de.php by Holger Oertel 2000, 2001, 2002, 2003 and can also be found at http://www.computus.de/menton/index.html by Herbert Metz 29.4.2000-11.4.2002. Unknown validity of algorithm. Slightly enhanced and implemented in Scilab by Werner Roethlisberger.

gd2esu2(gregorian date [[+|-]year,month,day;...;n])
It calculates the day for Easter Sunday of a Gregorian Year based on a different algorithm from above. This algorithm was first appeared 1876 in Butcher's Ecclesiastical Handbook. Adapted and implemented in Scilab by WR. The algorithm is valid for all positive years in the Gregorian calendar.

gd2esu3(gregorian date [[+|-]year,month,day;...;n])
This also calculates the day for Easter Sunday of a Gregorian Year based on a further different algorithm which was improved by Dr.Heiner Lichtenberg. Original By Jean Meeus. Found at:

- http:www.iivs.de/buchbach/buerger/tommyhof/kalender
- Published 1997 in the periodica 'Historia Mathematica 24' http://www.ptb.de/en/org/4/44/441/oste_e.htm

Adapted and implemented in Scilab by WR

gy2eudh(gregorian date [hd(0-7), [[+|-]year,month,day])
Calculates 8 different EU holidays or let say important dates of a full qualified Gregorian Year. The days with its corresponding interval for calculation:

- 0 = 1st Advent
- 1 = 2nd Advent
- 2 = 3rd Advent
- 3 = 4th Advent
- 4 = Mother's Day every 2nd Sunday in May
- 5 = Middle European Summertime (Begin daylight of saving time last Sunday of March)
- 6 = Middle European Wintertime (End daylight of saving time last Sunday of October)
- 7 = Day of Prayer and Repentance 2nd last Wednesday of November

The result will be a Gregorian Date in form [y m d wd] "wd" represents the ISO 8601 weekday which means the week starts with Monday.
Example for 1st Advent:

```
-->gy2euhd([0 2007 11 22]) >  2007.    12.    2.    7.
```

The algorithm is valid for all Gregorian Dates.

gy2ushd(gregorian date [hd(0-7), [[+|-]year,month,day])
Calculates 8 different US holidays of a full qualified Gregorian Year. The holidays with its corresponding interval for calculation:

- 0 = Mother's Day May
- 1 = Father's Day
- 2 = Election Day
- 3 = Thanksgiving
- 4 = Washington's Birthday
- 5 = Labor Day
- 6 = Columbus Day
- 7 = Veteran's Day

The original algorithm by W.M. O'Neil Time and the Calendars, Sydney University Press Australia, 1975. Adapted and implemented in Scilab by WR. Unspecified validity!

gd2jdnv(gregorian date mm.ddyyyy)
This algorithm calculates the number of weekdays of a Gregorian Dates.
Be aware of the different notation of the arguments and the limited validity because of the input format mm.ddyyyy.
The arguments must always include a leading zero for all month and days <10. This program calculates the current “W” as a variant of jdn in this lbrary. To calculate the weekdays (Mo-Fr) between two dates, you may use gd2nwd (below) as interface for passing the correct and consistent use of the arguments.

Original algorithm by W.M. O'Neil Time and the Calendars, Sydney University Press Australia, 1975 Adapted and implemented in Scilab with a small extension added by WR.
The original algorithm was from noon-to-noon, so the difference between a weekday and non-weekday was a half day. The small extension supplements this drawback, but makes the program somehow ugly. Sorry Mr. O'Neil!

Pope Gregory XIII decreed that the day after October 4, 1582 would be October 15, 1582, the Catholic countries of France, Spain, Portugal, and Italy complied. Various Catholic German countries (Germany was not yet unified), Belgium, the Netherlands, and Switzerland followed suit within a year or two, and Hungary followed in 1587. The algorithm is valid from beginning of Gregorian Calendar 1582, up to year 9999 positive years solely.

gd2nwd(gregorian date [[+|-]year1,month1,day1 [[+|-]year2,month2,day2])
This interface prepares and passes the argument to gd2jdnv for the algorithms N-weekdays from above and allows a user a more comfortable and consistent use of arguments.
Examples: gd2nwd([2007 1 1;2007 12 31]) > 261.

gd2jdnx(gregorian date mm.ddyyyy,wdn)
This program calculates the occurrence of an given weekday between two dates.
The week-day-numbers are defined by: Su=0, Mo=1, Tu=2, We=3, Th=4, Fri=5, Sa=6
Please notice the different notation of arguments and the limited validity e.g. mm.ddyyyy.
To calculate the occurrence of a given weekday between two dates you may use gd2mds (below) as the interface to passing the correct use of arguments.

Algorithm by W.M. O'Neil Time and the Calendars, Sydney University Press Australia, 1975
Adapted and implemented in Scilab by WR.

gd2mds(gregorian date [y m d;y m d] wdn 0-6)
This interface prepares and passes the arguments to gd2jdnx for the algorithms N-days from above and allows a user a comfortable and consistent use of arguments.
Example:

```
-->gd2mds([2007 1 1;2007 12 31],1)  >  52.
```

iwn2gd([[+]year,week number,[[+]iso8601weekday number];...;n])
This algorithm calculates the Monday of a Gregorian Date by passing a Gregorian year and a optional Gregorian ISO Week Number.

- The algorithm was tested for positive years solely
- ISO 8601 weekday numbers are 1=Mo, 2=Tu, 3=We, 4=Th, 5=Fr, 6=Sa, 7=Su
- The third parameter for passing the weekday number is optional whereby not limited to the weekday numbers 1 to 7 only.

add2gd(interval=["s"|"n"|"h"|"d"|"m"|"yy"|"bb"|"b"|"q"|"y"|"w"],[+|-]number [+|-], [Gregorian Date [y m d] or Gregorian Julian Day Number [gdjn]])

The meaning of intervals which will be added or subtracted:

- s=seconds
- n=minutes
- h=hours
- d=days
- m=month
- yy=years only
- bb=buddha years
- b=buddha year to date
- q=quarters
- y=full date
- w=weeks

Given Gregorian Date[[+|-]y m d] or Gregorian Date as Julian Day Number jdn([+|-]number)
The result will be Gregorian Date if Gregorian date was given or Gregorian Julian Day Number if gjdn was given. In case of any errors the given date will be leaved unchanged e.g. mis placed intervals etc.

Example: add two quarter of days (2*92) to a date

```
-->add2gd("q",2,[2006 3 10]) >  2006.    9.    10.
```

gd2jdns(gregorian date [[+|-]year,month,day;...;n], [season([1,2,3,4])
Calculates the the astronomical beginning for the four seasons (Spring, Summer, Fall and Winter). The result is Gregorian Julian Date and Time Number (jdn) and Julian Time Number (jtn) or better [jdnts] of a given Gregorian Date.

This algorithm should be valid for all positive Gregorian Dates between 1900 -2100

Original algorithm by Keith Burnett at http://www.xylem.f2s.com/kepler/
Implemented, heavily shortened and optimized for Scilab by Werner Roethlisberger

Meaning for the arguments are:

- 1 = Spring
- 2 = Summer
- 3 = Fall
- 4 = Winter

Empty season input (no argument) will list all four seasons.

Extensively tested with huge and various data from:
Earth's Seasons Equinoxes, Solstices, Perihelion, and Aphelion 1992-2020
http://aa.usno.navy.mil/data/docs/EarthSeasons.html

Example: calculate all four seasons by using the date from the system clock.

```
-->gd2jdns()

    2454180.5
    2454273.3
    2454366.9
    2454456.8
```

In this example the result is for the location of Greenwich by default!
Furthermore be aware of Scilabs default numbering format! You may change it by using format(n).

Find the beginning of summer for the Gregorian year 1950 for the location of Greenwich.

```
-->gd2jdns([1950 1 1],2) > 2433454.5
```

Using the more comfortable interface for gd2jdns. Actually there are two different interfaces available:

- gd2gdts([gregorian date = [+|-]year month day], [season = (1 to 4)], [tz = h [m] [s]])
- gdc2gdts([gregorian date = [+|-]year month day], [season = (1 to 4)], [nameofcity = “green”])

All interfaces accept all arguments as optional. By default the result will always be the location of “Greenwich” at system clocks date! Then the user may change the arguments at his needs and convenience. To use gdc2gdts becomes very comfortable since it takes all information form the internal latitude, longitude and time zone table with city names from locofcity (see description locofcity).

Examples:
calculate all four seasons for time zone -1 Rome, Italy at system clocks date (2007)

```
-->gd2gdts(,,[-1])

   2007.    3.     20.    23.    7.     38.
   2007.    6.     21.    17.    6.     21.
   2007.    9.     23.    8.     51.    12.
   2007.    12.    22.    5.     7.     57.
```

calculate the beginning of the winter season for the city of Rome, Italy for the year 1961, by using internal location table.

```
-->gdc2gdts([1961 1 1],4,"Rom")

       ans(1)

 ROME, ITALY

       ans(2)

    1961.    12.    23.    20.    7.    6.
```

gd2rdd(gregorian date [[+|-]year,month,day;...;n])
Gregorian Date to Rata Die Number. The Rata Die is a count of the number of days since a base date of December 31 of the year zero in the proleptic Gregorian calendar. Thus Rata Die day one occurs on January 1 of the year 1 which begins at Julian Day Number 1721425.5. Further information about Rata Die Number could be found on Peter Baum's websites.

rdd2gd(rata die number)
Rata Die Number to Gregorian Date.
Further information about Rata Die Number could be found on Peter Baum's websites.

jd2be(julian date[[+|-]year,month,day;...;n])
A Besselian epoch, named after the German mathematician and astronomer Friedrich Bessel (1784 – 1846) is an epoch that is based on a Besselian year of 365.242198781 days which is a tropical year measured at the point where the Sun's longitude is exactly 280°.

Besselian epochs are calculated according to:
B = 1900.0 + (Julian Date 2415020.31352) / 365.242198781
The standard epoch that was in use before the current standard epoch (J2000.0) was B1950.0, a Besselian epoch.
Since the right ascension and declination of stars are constantly changing due to precession, astronomers always specify these with reference to a particular epoch. Historically used Besselian epochs include B1875.0, B1900.0 and B1950.0. The official constellation boundaries were defined in 1930 using B1875.0.

be2jd(julian date jdn(be))
This is just the Besslian vice-versa function from above.

jd2jdn(julian date [[+|-]year,month,day;...;n])
Julian Date to Julian Day Number. The Julian Calendar was introduced by Julius Caesar in 45 BC. It was in common use until the late 1500s, when countries started changing to the Gregorian Calendar.
Original algorithm by Peter Baum "url: vsg.cape.com/~pbaum" implemented in Scilab by WR

jdn2jd(julian date [+|-]jdn)
Calculates Julian Day Number to Julian Date.
Original algorithm by Peter Baum "url: vsg.cape.com/~pbaum" implemented in Scilab by WR

jd2mjdn(julian date [[+|-]year,month,day;...;n])
Julian Date to Meeus Julian Day Number. This algorithm does in fact same as jd2jdn but it claims to be the original from Astronomical Algorithms, 1991, page 63, Jean Meeus.

mjdn2jd(julian date [+|-]mjdn)
Meeus Julian Day Number to Julian Date. This algorithm does same as jd2jdn, but it claims to be the original one from Astronomical Algorithms, 1991, page 63, Jean Meeus.

jd2wdn(julian date [[+]year,month,day;...;n])
Julian Date to cardinal week day number whereby the corresponding cardinal numbers are:
0=Su, 1=Mo, 2=Tu, 3=We, 4=Th, 5=Fr, 6=Sa

jd2wds(julian date [[+]year,month,day;...;n])
Julian Date to week day string corresponding to the cardinal numbers from above.

jd2iwdn(julian date [[+]year,month,day;...;n])
Julian Date to ISO 8601 week day numbers. ISO does of course not consider Julian dates. This algorithm was just made for compatibility reasons with the Gregorian programs of this library.

Following to the ISO 8601 the day of week numbers beginning with
1=Mo, 2=Tu, 3=We, 4=Th, 5=Fr, 6=Sa, 7=Su

jddiff(julian date [[+|-]y1 m1 d1;[+|-]y2 m2 d2;...,...;n])
Difference of two Julian Dates.

adday2jd(julian date [[+|-]year month day jday...;...,...;n])
Add/Subtract days from Julian Date.
Example: add 3 days to a Julian Date adday2jd([-1233 4 1,3]) > - 1233. 4. 4

jd2dpm(julian date [[+|-]year month day jday...;...,...;n])
Days Per Month of a Julian Date.

jd2dpy(julian date [[+|-]year month day jday...;...,...;n])
Days Per Year of a Julian Date.

jd2boy(julian date [[+|-]year month day jday...;...,...;n])
Days from the beginning of a Julian Year.

jd2eoy(julian date [[+|-]year month day jday...;...,...;n])
Days to the end of a Julian Year.

jdissasu(julian date [[+|-]year month day jday...;...,...;n])
Test if Julian Date is Weekend (Saturday or Sunday). The numeric result will be true = 1 if the date is a weekend otherwise false = 0.

jdismofr(julian date [[+|-]year month day jday...;...,...;n])
Test if Julian Date is Weekday Monday to Friday. The numeric result will be true = 1 if the date is a weekday (Mo to Fri) otherwise false = 0.

jdislyr(julian date [[+|-]year,month,day;...;n])
Test if Julian Date is Leap Year. The numeric result will be true = 1 for any leap year or false = 0 if not a leap year.

jd2esu(julian date [[+|-]year,month,day;...;n])
Calculates the julian date of easter sunday at a given julian year for uknown range of year. Easter Sunday is the first Sunday after the first full moon after vernal equinox. Original algorithm by C.F. Gauß from the year 1800.

Found at http://www.computus.de/menton/index.html by Herbert Metz implemented in Scilab by Werner Roethlisberger.

add2jd(interval=["s"|"n"|"h"|"d"|"m"|"yy"|"q"|"y"|"w"],[+|-]number [+|-],[JULIAN DATE OR JULIAN DAY NUMBER]) or Julian Day Number [gdjn]])

Meaning of intervals which will be added or subtracted:

- s=seconds
- n=minutes
- h=hours
- d=days
- m=month
- yy=years only
- bb=buddha years
- b=buddha year to date
- q=quarters
- y=full date
- w=weeks

Given Julian Date[[+|-]y m d] or Julian Date as Julian Day Number jdn([+|-]number) the result will be Julian Date if Julian Date was given or julianish Julian Day Number if jdn was given. In case of any errors the given date will be leaved unchanged e.g. mis placed intervals etc.

jdnepoch([“epoch”]|[epoch sequence n])
This function searches case insensitive for any search string and checks for first match e.g. "ed-din" would find the epoch of Iranian Dschelal ed-Din at JD 1079-03-15 etc. and returns the value as “jdn” of the corresponding epoch. In other hand it is also possible to give a corresponding sequence number to get a epoch e.g.
jdnepoch(18) is equivalent to jdnepoch(“ed-din").

If no argument is given a full list of all available epochs will be generated as a matrix.
Type jdnepoch to get the full list.

Examples:

```
-->jdnepoch("ed-Din") > 2115235.5
-->jdnepoch(18) > 2115235.5
-->jdnepoch()
```

Be aware of Scilab's default number format

addby(gregorian date [[+|-]year month day])
Add Buddhist Year Constant (543) to a Gregorian Year.

subby(thai buddhist date [[+|-]year month day])
Subtract Buddhist Year Constant (543) from a Buddhist Year.

igcall(gregorian date [y,m,interval ['g'|'b']])
Creates a ISO 8601 Gregorian Calendar as a list either from a Gregorian or a Thai Buddhist Date which is indicated by the interval “b” while Gregorian Date is “g” by default. All arguments are optional to create a calendar from the system clock.

Example:

```
   ans(1)

        ISO 8601 Gregorian Calendar Nov 2007

   ans(2)

                  (Leap Year 'NO')

   ans(3)

Mon    Tue    Wed    Thu    Fri    Sat    Sun    W

   ans(4)

0.     0.     0.     1.     2.     3.     4.     44.
5.     6.     7.     8.     9.     10.    11.    45.
12.    13.    14.    15.    16.    17.    18.    46.
19.    20.    21.    22.    23.    24.    25.    47.
26.    27.    28.    29.    30.    0.     0.     48.
```

igcalf(gregorian date [y,m,interval ['g'|'b']])
Creates a ISO 8601 Gregorian Calendar but as a formated table either from a Gregorian or from a Thai Buddhist Date which is indicated by the interval “b” while for Gregorian Date is “g” by default. All arguments are optional to create a calendar from the system clock.

Example:

```
-->igcalf(1961,1)

     ISO 8601 Gregorian Calendar Jan 1961
               (Leap Year 'NO')
    -------------------------------------
    Mon  Tue  Wed  Thu  Fri  Sat  Sun   CW
    -------------------------------------
                                  1   52
      2    3    4    5    6    7    8    1
      9   10   11   12   13   14   15    2
     16   17   18   19   20   21   22    3
     23   24   25   26   27   28   29    4
     30   31                             5
```

If you want to give a Thailand's Buddhist Date instead of a Gregorian, this must be indicated by using the interval “b” for the Buddha year. Otherwise the calculated calendar will NOT be wrong, but just misinterpreted.

```
-->igcalf(2504,1,"b")

     ISO 8601 Gregorian Calendar Jan 1961
               (Leap Year 'NO')
    ------------------------------------
    Mon  Tue  Wed  Thu  Fri  Sat  Sun   CW
    ------------------------------------
                                   1   52
      2    3    4    5    6    7    8    1
      9   10   11   12   13   14   15    2
     16   17   18   19   20   21   22    3
     23   24   25   26   27   28   29    4
     30   31                             5
```

ijcall(julian date [y,m,interval [“b”|”j”])
Creates a ISO 8601 Julian Calendar as a list from a Julian Date. All arguments are optional to create a calendar from th system clock. ISO does NOT consider Julian Dates of course! You may use Thailand's Buddhist year which is indicated by the interval “b” while for Julian Dates is “j” by default.

Example:

```
-->ijcall(1145,2)

       ans(1)

          ISO 8601 Julian Calendar Feb 1145

       ans(2)

                   (Leap Year 'NO')

       ans(3)

    Mon    Tue    Wed    Thu    Fri    Sat    Sun

       ans(4)

    0.     0.     0.     1.     2.     3.     4.
    5.     6.     7.     8.     9.     10.    11.
    12.    13.    14.    15.    16.    17.    18.
    19.    20.    21.    22.    23.    24.    25.
    26.    27.    28.    0.     0.     0.     0.
```

ijcalf(julian date [y,m,interval [“b”|”j”])
Creates a ISO 8601 Julian Calendar as a formated table from a Julian Date. All arguments are optional to create a calendar from current system clock. Again, ISO does NOT consider Julian Date. You may use Thailand's Buddhist year which is indicated by the interval “b” while for Julian Date is “j” by default.

Example:

```
-->ijcalf(1148,6)

    ISO 8601 Julian Calendar Jun 1148
            (Leap Year 'YES')
    ---------------------------------
    Mon  Tue  Wed  Thu  Fri  Sat  Sun
    ---------------------------------
           1    2    3    4    5    6
      7    8    9   10   11   12   13
     14   15   16   17   18   19   20
     21   22   23   24   25   26   27
     28   29   30
```

tbcall(thai buddhist date [y,m,interval])
Creates a Thailand's Buddhist Calendar as a list either from a Thai Buddhist Year or from a Gregorian Date indicated by the interval “b” which is default, while for Gregorian Date is “g”. All arguments are optional to create a calendar from the current system clock.

The official year in Thailand is reckoned from 543 BC, the beginning of the Buddhist Era. Means the Gregorian year 1999 is equal to Buddhist year 2542 in Thailand.

Example: Celebration of His Majesty King Bhumibol Adulyadej The King Of Thailand's 80th birthday at Thailand's Buddhist Date 5th December 2550.

```
-->tbcall(2550,12)

       ans(1)

       Thailand Buddhism Calendar Dec 2550

       ans(2)

                  (Leap Year 'NO')

       ans(3)

    Sun    Mon    Tue    Wed    Thu    Fri    Sat

       ans(4)

    0.     0.     0.     0.     0.     0.     1.
    2.     3.     4.     5.     6.     7.     8.
    9.     10.    11.    12.    13.    14.    15.
    16.    17.    18.    19.    20.    21.    22.
    23.    24.    25.    26.    27.    28.    29.
    30.    31.    0.     0.     0.     0.     0.
```

Same as from above but as now in a formated way!

tbcalf([tbyear|gregorian year,month,interval[“b”|”g”])

```
-->tbcalf(2550,12)

   Thailand Buddhism Calendar Dec 2550
            (Leap Year 'NO')
    --------------------------------
    Sun  Mon  Tue  Wed  Thu  Fri  Sat
    --------------------------------
                                   1
      2    3    4    5    6    7    8
      9   10   11   12   13   14   15
     16   17   18   19   20   21   22
     23   24   25   26   27   28   29
     30   31
```

If you want to give a Gregorian year instead of a Thai Buddhist year, just pass the interval “g” otherwise the calendar will NOT be wrong, but misinterpreted!

```
-->tbcalf(2007,12,"g")

   Thailand Buddhism Calendar Dec 2550
            (Leap Year 'NO')
    ---------------------------------
    Sun  Mon  Tue  Wed  Thu  Fri  Sat
    ---------------------------------
                                    1
      2    3    4    5    6    7    8
      9   10   11   12   13   14   15
     16   17   18   19   20   21   22
     23   24   25   26   27   28   29
     30   31
```

Additional Date Calculations

Weekday calculation by Julius Christian Johannes Zeller (24 June 1822 Mühlhausen am Neckar, Germany - 31 May 1899 Cannstatt)

Published by the Royal Office for Statistics and Topography. Annual volume 5, 1882. Stuttgart. W. Kohlhammer 1882.

Further information can be found @ http://en.wikipedia.org/wiki/Christian_Zeller
This unique algorithm has been adapted and implemented for Scilab by Werner Roethlisberger

Based on the algorithm the validity is limited for positive years within 4 digits (1000 – 9999).
Trap warning: None of the algorithms checks for the Gregorian calendar reform at
15th October 1582.

Be informed that those algorithms below are still in beta state. If you find any errors feel free to contact me.

gd2zwdg([year month day]) > (1=Sun, 2=Mon, 3=Tue, 4=Wed, 5=Thu, 6=Fri, 0=Sat)
This algorithm calculates the weekday number of a giving Gregorian date. It calculates directly out from the Gregorian date without any prior conversion. However, be aware of the different meaning of results from this algorithm: 1=Sun, 2=Mon, 3=Tue, 4=Wed, 5=Thu, 6=Fri, 0=Sat

Example: 1882, 11th Sept
gd2zwdg([1882 9 11]) > 2

The 11th of September 1882 was on the second day of the week or Monday.

jd2zwdj([year month day]) > (1=Sun, 2=Mon, 3=Tue, 4=Wed, 5=Thu, 6=Fri, 0=Sat)
This algorithm calculates the weekday number of a giving Julian Date. It calculates directly out from the Julian Date without any prior conversion. However, be aware of the different meaning of results from this algorithm: 1=Sun, 2=Mon, 3=Tue, 4=Wed, 5=Thu, 6=Fri, 0=Sat

Example: 1492, 12th Oct. Discovery of the New World.
Jd2zwdj([1492 10 12]) > 6

The 12th of October 1492 was on the sixth day of the week or Friday.

d2zwd([year month day])
This little program is just an interface for both algorithms from above, but it checks for the Gregorian calendar reform first and calls after the corresponding algorithms. The input is therefore either a Gregorian or a Julian date and the trap warning can be revoked here.

Easter calculation by Julius Christian Johannes Zeller (24 June 1822 Mühlhausen am Neckar, Germany - 31 May 1899 Cannstatt)

Based on the algorithm the validity is limited for positive years within 4 digits (1000 – 9999).

gd2zeg(Gregorian year)
The algorithm calculates days after 21th March from a Gregorian year.

Example: Easter 1886.
gd2zeg(1886) > 35

Easter is 35 days after the 21st of March = 56th of March = 25th of April.

jd2zej(Julian year)
The algorithm calculates days after 21th March from a Julian year.

Example: Easter 1355. Coronation of Charles IV in Rome.
Jd2zej(1355) > 15

Easter is 15 days after the 21st of March = 36th of March = 5th of April.

First Pasha day by Karl Friedrich Gauß 1802 by considering the Gregorian calendar reform. The validity of this algorithm is unknown. I assume that it will work fine for positive years in certain range of years.

gd2fpd([year month day])
This algorithm is actually in beta state and NOT yet finally tested. If you find something wrong please let me know.

Math Calculation

scale([min max scale value])
This algorithm is to scale any kind of values makes scaling now very easy.
Let say one needs to scale 360 degrees in 16 compass directions and want to find out any direction in degrees:

Type in:
```
-->a = 128 (while we looking for the compass direction “SE = 128 degrees”)
-->cv=scale([0 360 16 a]);cv=scale([0.0 16.0 3600.0 cv])/10 > 135.
```

val2mx(value)
Converts reel values to 3 x 1 matrix. Since many algorithm in this library takes its arguments as a 3 x 1 matrix it becomes helpful having this little matrix converter.

mx2val([matrix])
Converts a 3 x 1 matrix to a real value.

alog1(x)
Antilogarithmus.

xlogn(x, base)
Logarithmus.

xpower(base, n)
Power of base n.
NOTE: Some implementation have different definition!

ex(x)
Natural Exponential.

ln(x)
Natural Logarithmus

nsqrtx(n, x)
N Square Root from x.

floor2(x)
Rounding down.
NOTE: Some implementation have different definition!

round2(x, comma)
Rounding up reel values.
NOTE: Some implementation have different definition!

trunc(x, comma)
Truncates reel values to the digits of (commas).

ip(x)
Integer Part
NOTE: Some implementation have different definition!
In case your calculation needs do get INT(-7.3)=-8 then use Scilab's floor instead.

fp(x)
Floating Part.

mod(n,m)
This emulates Scilab's modulo defintion:
mod(-5,21) = 16, mod(-5,-21) = -5, mod(5,21) = 5
NOTE: Some implementation have different definition.

mod2(n,m)
This emulates Scilab's pmodulo definiton:
(mod2(-5,21) = 16, mod2(-5,-21) = -5, mod2(5,21) = 5
NOTE: Some implementation have different definition.

amod2(n,m)
Returns numerator if modulus is zero.

crossum(x)
Calculates the cross sum of a value.

Example:
```
-->crossum(123) > 6
```

sgn(x)
Signum of a value.
x<0 result = -1 | x > 0 result = 1 | x = 0 result = 0
NOTE: Some implementation have different definition!

isint(x)
Test is value is integer.
NOTE: Some implementation have different definition!

cib2b(in, ibase, obase)
Convert the original integer number from any base between desired output base between 2-36. This algorithm is taken from BASIC Programs edited by Lon Poole and published by Osborne/McGraw-Hill, copyright 1980 adapted and implemented in Scilab by Werner Roethlisberger.

Example:
```
-->cib2b("FFFF",16,8) > 177777
```

Values can bee given as string as well as integer values. Please consider that values with a higher base than 10 must always given as a string.

a2r(value)
Converts (Arabic integer) value to roman string.
Example:

Convert year 2010 into roman.

```
->a2r(2010)
 ans  =

 MMX
```

r2a(romanstr)
Converts a roman string to (Arabic) integer value.

Convert roman to arabic value:

```
-->r2a("MMX")
 ans  =

    2010.
```

Compatibility Emulations

Please notice that this emulation functions are not designed for daily use, they are ment for make it a bit easier to convert algorithms from other environments. After a successful converting of your algorithm to Scilab, always apply the corresponding functions from this library or from Scilab to fully unchain your algorithms from other system dependencies.

asin2(x)
Emulation of "Application.Asin2"

acos2(x)
Emulation of "Application.Acos2"

atan2(x,y)
Emulation of "Application.Atan2"
NOTE: Some implementation have different definition!

atan3(y,x)
Emulation of a variant of atan which result in -pi< Atn2 <= pi

arcsin2(x)
Emulation of arcsin2.

arctan2(x,y)
Emulation of arctan2.

val2(str)
Emulation of value which converts a string to a reel value.
NOTE: Some implementation have different definition!

dateserial2(y, m, d)
Unlimited emulation of dateserial2.

timeserial2(h, m, s)
Emulation of timeserial2.
Note that the integer part of the result (1.0) points on to day.

weekday2(y, m, d)
Limited emulation of weekday2.
Note that this algorithm solely considering "Sunday" within the 1900 date system.

The corresponding weekday numbers are:
1=Su, 2=Mo, 3=Tu, 4=We, 5=Th, 6=Fr, 7=Sa

year2(dateserial2)
Emulation of year2.

month2(dateserial2)
Emulation of month2.

day2(dateserial2)
Emulation of day2.

Angle Conversion

pi(void)
Generates an accurate phi. It is a bit time consuming but not need atan or phi as a constant value. It needs the sqrt function instead, which is usually available.

rad2deg(radians)
Converts Radians to Degrees.
Note that this function checks the existence of the variable “radindeg”. If it exits the calculation would be somehow faster because of the unnecessary divide in phi. Further information can be found for the constant library.
Example: convert 4 radians to decimal degrees rad2deg(4)=229.18312

rad2grad(radians)
Converts Radians to Grads.

deg2rad(degrees)
Converts Degrees to Radians.

Examples:
```
-->find cosine of 35 degrees: cos(deg2rad(35))=.8191520 (degrees)
-->find sine of 5 degrees: sin(deg2rad(5))=0.0871557 (degrees)
-->convert 45 degrees to radians: deg2rad(45)=.7853982 (radians)
```

deg2grad(degrees)
Converts Degrees to Grads.

grad2rad(grads)
Converts Grads to Radians.
Example: find tangents of 43.66 grad tan(grad2rad(43.66))=.8183157 (grad)

grad2deg(grads)
Converts Grads to Degrees.

deg2bm(degrees)
Converts Degrees to Bogenmass.

bm2deg(bogenmass)
Converts Bogenmass to Degrees.

grad2bm(grads)
Converts Grads to Bogenmass.

bm2grad(bogenmass)
Converts Bogenmass to Grads.

range360(x)
Limits the angle to 360 Grads.

rangerad(x)
Limits an angle in Radians.

xy2pr(x, y)
Convert rectangular to polar coordinates with angle in radians.

Example: convert rectangular coordinates (4,3) to polar with the angle in radians
`-->xy2pr(4,3)=5. + 0.6435011i` (magnitude r + angle in radians i)

xy2pg(x, y)
Convert rectangular to polar coordinates with angle in grads.

Example: find th length of the diagonal and the angle in degrees of a square with 4 meter length each side

```
-->xy2pd(4,4) = 5.6568542 + 45.i
```

xy2pd(x, y)
Convert rectangular to polar coordinates with angle in degrees.

pr2xy(cxra)
Convert complex polar-coordinates to rectangular with angle in radians.

pg2xy(cxra)
Convert complex polar-coordinates to rectangular with angle in grads.

Example: convert polar coordinates (8,120 grads) to rectangular coordinates
`-->pg2xy(8+120*%i) = -2.472136    7.6084521` (x-coordinate y-coordinate)

pd2xy(cxra)
Convert complex polar-coordinates to rectangular with angle in radians.

argr(cxy)
Convert complex xy-coordinates to polar with angle in radians.

argg(cxy)
Convert complex xy-coordinates to polar with angle in grads.

Example: convert complex xy coordinates in its angle (real) in grad
`-->argg(-7.3+3*%i) = 175.17706` angle (real) in grad

argd(cxy)
Convert complex xy-coordinates to polar with angle in degrees.

cxy2rad(cxy)
Convert complex xy-coordinates to angle in radians.

cxy2deg(cxy)
Convert complex xy-coordinates to angle in degrees.

cxy2grad(cxy)
Convert complex xy-coordinates to angle in grads.

ang2time(a)
Convert angle 360 grads to angle in time.

a2t(a)
Convert angle time to angle 360 in grad.

hrd2cxy(hr)
Convert hour in degrees to complex xy-coordinates.

hmd2cxy(hm)
Convert hours and minutes in degrees to complex xy-coordinates.

mind2cxy(min)
Convert minutes in degrees to complex xy-coordinates.

secd2cxy(sec)
Convert seconds in degrees to complex xy-coordinates.

hr2deg(hr)
Convert hours to degrees.

in2deg(min)
Convert minutes to degrees.

sec2deg(sec)
Convert seconds to degrees.

deg2compassv(degrees)
Convert degrees to compass direction as a value. There are 16 compass directions available:

0 / 22.5 / 45 / 67.5 / 90 / 112.5 / 135 / 157.5 / 180 / 202.5 / 225 / 247.5 / 270 / 292.5 / 315 / 337.5

Example:
what will be the nearest compass direction for 169 degrees?

```
-->deg2compassv(169)
 ans  =

    180.
```

deg2compasst(degrees)
Convert degrees to compass direction as string (text). There are 16 compass direction as text available:

"N", "NNE", "NE", "ENE", "E", "ESE", "SE", "SSE", "S", “SSW", "SW", "WSW", "W", "WNW", "NW", "NNW"

Example:

How much in degrees will be the compass direction for “SSE”?

```
-->compasst2deg("sse")
 ans  =

    157.5
```

compasst2deg(compasstxt)
Convert compass direction in text to values in degrees. There are 16 directions as text available:

"N", "NNE", "NE", "ENE", "E", "ESE", "SE", "SSE", "S", “SSW", "SW", "WSW", "W", "WNW", "NW", "NNW"

Example: Same from above but the result will now be in text.

```
-->deg2compasst(169)
 ans  =

 S
```

Business Calculation

cround(n, comma)
Round currency to next x.x5.

Example:
```
-->cround(23.73,2)
 ans  =

    23.75
```

percent(n1, n2)
Calculate percent.

dpercent(n1, n2)
Delta percent.

tpercent(n1, n2)
t percent.

addperc(n, p)
Add percent p to n.

Navigation Calculation

esrss(mm, dd, glat, glon, riseset, tz)
Estimated time of sunrise and sunset from Applications Library ©1976, Texas Instruments Incorporated adapted, implemented in Scilab and some little improvements added by Werner Roethlisberger.

Original text from the book:

- Atmospheric refraction is not considered
- Errors increases with Latitude <65°
- Errors of approx 0.1 hours may be possible with this approximations
- Negative GMT means preceding day
- GMT >24 means succeeding day

Be aware that the arguments of this algorithm differs now slightly from its original one. See below:

esrss([date mm, dd],glatitude,glongitude,riseset,tz)

- cdate is the optional Gregorian date for calculation
- latitude in degrees (for +N/-S)
- longitude in degrees (for +E/-W)
- rise set value for rise[+1] | set[-1]
- time zone [+|-]

The original algorithm would need longitude entries in -E/+W which differs from UTC time zones and locations. That's why a line have been added for making the algorithm fits to UTC time zones as well as the locations are in standard -W/+E notations in this library.

Furthermore it needs the month (mm), day (dd), latitude in degrees, longitude in degrees and a value for indicating rise -1 and +1 for sunset and optional a time zone.

Examples:

```
esrss(10, 24, 33.54, -94.31, -1, 0)  >  12.    31.    55 GMT for Sunrise
esrss(10, 24, 33.54, -94.31, 1, 0)  >   23.    30.    1. GMT for Sunset
```

etsrss([gregorian date [y m d]], [name of city])
This is just a interface for esrss to make the use somehow more comfortable. All arguments are optional here for the actual system date at “Greenwich” of course. For daily hobby users this small algorithm is accurate enough. If more high precision data needed, please see at **srss** algorithm later.

Example: `-->etsrss(,"bangk")`

```
!Estimated                                                    !
!                                                             !
!Sunrise and        Sunset for                                !
!                                                             !
!BANGKOK            THAILAND                                  !
!                                                             !
!Lat:               13.45       Lon:          100.31        !
!                                                             !
!Timezone:          7           At date:      2007/11/22    !
!                                                             !
!Sunrise at:        061925      Sunset at:    175145        !
!                                                             !
!Daylength (hms):   113220      Noon (hms):   120535        !
!                                                             !
!                                                             !
```

locofcity(['nameofcity'])
This function generates a list of cities in several different ways:

- no arguments generates a full list of all available cities
- the internal table holds the country name which can be searched
- the internal table holds the names of cities which can be searched
- full name as well as just a part of the name can be searched too

However, the search function is case insensitive!

The result will always be a list with:

- the name of the city and country
- the coordinate latitude in degrees
- the coordinate longitude in degrees
- the time zone

Access to data elements data are very easy, since the data is represented as a simple 4 x n matrix. Just count the column and row and you'll get access to the elements.

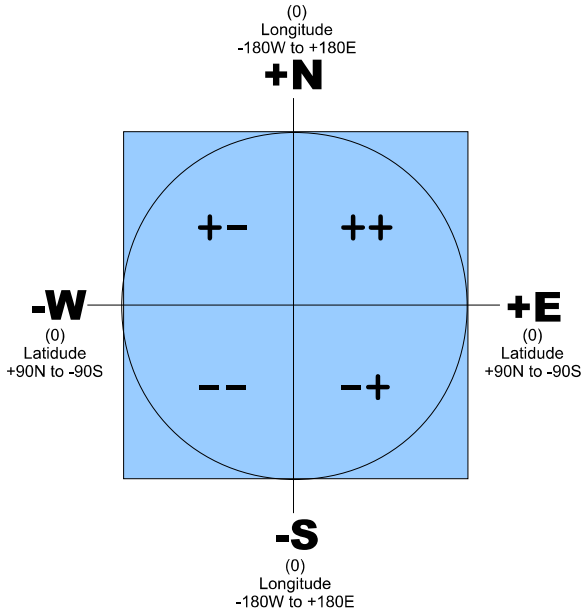
Example: To get access to latitude of New York which is at column 2 and row 2 you just may type in: cities=locofcity('usa ');cities(2,2) > “40.44”

Well, of course, the result is a string! But yes it can be made a floating point value in several easy ways:

- cities=locofcity('usa ');val2(cities(2,2)) > 40.44
- cities=locofcity('usa ');msscanf(cities(2,2),"%lg") > 40.44

Declaration notice:

- **Latitude** (+90N to -90S) is in degrees **positive for north** and **negative for south.**
- **Longitude** (-180W to +180E) is in degrees **negative for west** and **positive for east.**
- **Time Zone** in UTC ([-|+]hh:mm) is **negative for west** and **positive for east** (-W/+E).



Other examples: `-->locofcity('usa')`

```
!Jerusalem, Israel     31.7833  35.2333   2   !
!                                           !
!New York, USA         40.44    -73.55    -5  !
!                                           !
!Chicago, USA          41.51    -87.41    -6  !
!                                           !
!Detroit MI, USA       42.23    -83.05    -6  !
!                                           !
!Houston, USA          29.45    -95.23    -6  !
!                                           !
!Washington DC, USA    38.53    -77.02    -5  !
!                                           !
!Miami, USA            25.47    -80.13    -5  !
!                                           !
!W. Lafayette, USA     40.2624  -86.5436  -6  !
!                                           !
!Denver, USA           39.43    -104.59   -7  !
!                                           !
!Atlanta Georgia, USA  33.46    -84.25    -5  !
```

Notice: The search phrase 'usa' can be a part of the city too. To prevent unwanted results, just be a bit more precise e.g. 'usa ' (e.g. with a blank or so). It would then return a slightly different list:

Examples: `-->locofcity('usa ')`

```
!New York, USA         40.44    -73.55    -5  !
!                                             !
!Chicago, USA          41.51    -87.41    -6  !
!                                             !
!Detroit MI, USA       42.23    -83.05    -6  !
!                                             !
!Houston, USA          29.45    -95.23    -6  !
!                                             !
!Washington DC, USA    38.53    -77.02    -5  !
!                                             !
!Miami, USA            25.47    -80.13    -5  !
!                                             !
!W. Lafayette, USA     40.2624  -86.5436  -6  !
!                                             !
!Denver, USA           39.43    -104.59   -7  !
!                                             !
!Atlanta Georgia, USA  33.46    -84.25    -5  !
```

srss(gdate, glat, glong, riseset, altitude, tz)
Calculates a much more accurate time for sunrise and sunset for any place in the world.
The method is from 9.311 and 9.33 of the Explanatory Suppliment to the Astronomical Almanac (Seidelmann, 1992) implemented in Scilab by Werner Roethlisberger.
The algorithm takes the Julian Day Number since the year 2000 at 0h UT on a given Gregorian Date. The optional arguments [altitude] can be used to give an arbitrary value for the altitude of the Sun. The default value corresponds to the top limb of the sun touching the mathematical horizon, allowing for refraction at sea level.
This algorithm may not give sensible values above 60N or below -60N, and is only accurate to a few minutes either way. Sun rise/set times dependents heavily on local horizons anyway.

The arguments are:
srss([cdate],glatitude,glongitude,riseset,refraction,tz)

- cdate is the optional Gregorian Date for calculation
- latitude in degree (for +N/-S)
- longitude in degrees (for +E/-W)
- rise set value for rise[+1] | set[-1]
- refraction in degree (see below for altitudes)
- time zone [+|-]

Meaning of altitudes in degrees:
- 18.0 result at astronomical time
- 12.0 result at nautical time
- 6.0 result at civil time
- 0.833 result at sun (default) time

srssc([gregorian date [y m d]], [name of city])
Interface for srss to make the use somehow comfortable. All arguments are optional for the actual date at “Greenwich”.

Example: Sunrise and Sunset at current date in Buenos Aires, Argentina at system clock date (2007).

```
-->srssc(,"buen")

!Seidelmann's                                          !
!                                                      !
!Sunrise and        Sunset for:                        !
!                                                      !
!BUENOS AIRES       ARGENTINA                          !
!                                                      !
!Lat:               -34.2        Lon:          -58.3   !
!                                                      !
!Timezone:          -3           At date:      2007/11/22  !
!                                                      !
!Sunrise at:        053734       Sunset at:    194314  !
!                                                      !
!Daylength (hms):   140540       Noon (hms):   124024  !
!                                                      !
```

gd2mrs(year, month, day, lat, long, tz)

This Program computes the UTC time of moonrise and moonset anywhere in the world. Over the years, a series of articles discussing the computation of times of rise and set was published in Sky & Telescope magazine.

All of the articles were written by Roger W. Sinnott:

- Sky & Telescope, July 1989, pp. 78-80, "Ups and Downs of the Moon".
- Sky & Telescope, August 1994, pp. 84-85, "Sunrise and Sunset: A Challenge".
- Sky & Telescope, March 1995, pp. 84-86, "Sunrise/Sunset Challenge: The Winners".

Original text by Sky & Telescope magazine:
“BASIC is quite readable, thus facilitating the conversion of the code to other computer languages. Better still, the BASIC source code given in the first two articles is available for free download from Sky & Telescope's Web site.
The programs are SUNUP.BAS and MOONUP.BAS code by Roger W. Sinnott was based on algorithms by Jean Meeus in Astronomical Formulae for Calculators (Willmann-Bell, 1982).”

This unique algorithms which is based on the Century of the Gregorian year 2000-01-01 is here adapted, slightly enhanced and implemented in Scilab by Werner Roethlisberger.

It have been added TDT Terrestial Dynamical Time correction. Used as a time-scale of ephemerides from the Earth's surface.

TDT = TAI + 32.184 seconds formerly called ET, Ephemeris Time.

Furthermore It has been added a reverse of the time zone (1-*tz) for make it more consistent use with other programs in this library.

The arguments are now looking like this:

- Gregorian Date comma separated year, month, day
- latitude (DD.MMSS) (+N|-S)
- longitude (DD.MMSS) (-W|+E)
- time zone [+|-] UTC

mrsc(gregorian date [y m d], [name of city])
Interface for gd2mrs to make the use somehow more comfortable. All arguments are optional for the system date at “Greenwich”.

```
-->mrsc(,"johann")

!Roger          W. Sinnott's                        !
!                                                   !
!Moonrise and   Moonset for:                        !
!                                                   !
!JOHANNESBURG   SOUTH AFRICA                        !
!                                                   !
!Lat:           -26.08        Lon:       27.54      !
!                                                   !
!Timezone:      2             At date:   2007/11/22 !
!                                                   !
!Moonrise at:   162844        Azimuth:   72.129377  !
!                                                   !
!Moonset at:    030043        Azimuth:   284.12674  !
!                                                   !
```

gd2mp(gymd, hms, tz)
Moon phase for a Gregorian date at a time in hms.
No information available about validity of this algorithm. But based on the way of the calculation it will be valid between 1900 – 2100.

Since the phase of moon appears same all over the world there is no need to calculate the phases for a specific location on the world. An exception from this is the perspective of the light and shadow which changes depending from observers location. E.g. ascending on the northern hemisphere appears as “)” while on southern hemisphere the same phase appears as “(“ etc.

Interpretation of results:

- The results doesn't show the illuminated nor the non-illuminated part of the moon!
- The moon phases results are values in degrees where 360 to 180 means descending phases while 180 to 0 means ascending phases.
- Full moon is always near at 0 or 360 degrees while newmoon is always near 180 degrees.
- Negative values indicates descending phases fo making results more human readable.
- Never expect exact 180 for new moon or 0/360 for full moon without any given time data.
- The accuracy is around 30 minutes or less for several tested years.

Meaning of results are:

- around 0 or 360 = Full Moon

Descending:

- -315 = Three Quarter (descending)
- -270 = Last Half (descending)
- -225 = a Quarter (descending)

New Moon:

- 180 = New Moon (no illumination)

Ascending:

- 135 = a Quarter (ascending)
- 90 = First Half (ascending)
- 45 = Three Quarter (ascending)

- 0 or 360 = Full Moon

Example:

```
-->gd2mp([2007 11 22],,) > 29.770617
```

This means that the phase of moon recently was started to ascending at this date for the location of Greenwich, England. Because there was no time zone given.

Same example but for the time zone (TZ in UTC) +7 in Southeast Asia:

```
-->gd2mp([2007 11 22],,7)
 ans  =

    25.68799077635776
```

Basic Statistics

linfit(predict_y, x_data, y_data)
Linear fitting

Example:
```
-->predict_y=5,x=[1 2 3 4 5],y=[11 16 20 23 26];
```

logfit(predict_y, x_data, y_data)
Logarithmic fitting

expfit(predict_y, x_data, y_data)
Exponential fitting

pwrfit(predict_y, x_data, y_data)
Power fitting

mvafit(predict_y, x_data, y_data)
Moving average fitting

Ascii Graphics

Please notice in case you'll like to tryout this ascii graphics functions, that they would appear best with courier fonts!

abar([min, max, barsize, barchar, value])

Example:
Draw a bar for a given data point (-5) which is expected in the range between -10 and 10. In this example the length of the bar we want limited to 30 characters and we choose the hash as draw characters. The result must be interpreted as a ratio between a data point (-5), the range (min -10) and (max 10) and resolution (30). The range is not limited to a symetric range of course.

The result below shows the null-line (zero) as “|” and the bar is therefore on the negative side.

```
-->abar([-10, 10, 30, str2code("#"), -5])
 ans  =

        ########|
```

mabar([min, max, barsize, barchar], [valuelist as matrix of n*m element(s)])

Same as from above, but with multiple data points and the data points must be given as matrix or let say a list. In this example we been giving 5 data points [-5, 6, .1, -10, 15]. The range we can leave same as from above.

The zero line here is same “|” but what means the “@” here? As you can see, the 3rd data point is NOT zero, but would not be drawn because of its very small value. To prevent disappearing of data points which are <> zero, for all such data points a “@” will be placed instead of the zero line “|”.

It is suggest to play a around a bit with all this functions. It can be fascinating to let ASCII's rock.

```
-->mabar([-10, 10, 30, 37],[ -5, 6, .1, -10, 15])
 ans  =

!        ########|                  !
!                                   !
!                 |#########        !
!                                   !
!                 @                 !
!                                   !
!################|                  !
!                                   !
!                 |###############  !
```

aplot([min, max, plotsize, plotchar, value])

We plotting now, while before we have been drawn bars. But now you'll understand already how it works what all this means about. Just recall your input from above and type aplot. Change the data point, let say to a 6 and then press enter. Don't forget we plotting now, that's why it looks somehow different from above.

```
-->aplot([-10, 10, 30, 37, 6])
 ans  =

                 |        #
```

maplot([min, max, plotsize, plotchar], [valuelist as matrix of n*m element(s)])

Again just recall the example from above or copy it from here to your Scilab console. However, we want to use maplot (multiple ascii plot) and then press the enter key. I have drawn a line over the data points to make it a bit more human readable. Well yes, I have been jitter a bit, but it's hand made!

```
-->maplot([-10, 10, 30, 37],[ -5, 6, .1, -10, 15])
 ans  =

!        #        |                  !
!                                    !
!                 |        #         !
!                                    !
!                 @                  !
!                                    !
!#                |                  !
!                                    !
!                 |                  #  !
```

eaplot([min, max, plotsize], [plotchar number], [valuelist])

Again, just recall the examples from above or copy the line from here to your Scilab console. However, this looks very surprising doesn't it? But it is easy to understand just look again at the parameter meanings. Now, here every data point can have it's own character. As many data points you have a same amount of characters you should give. In Scilab ascii(10) means "a" ascii(11) means "b" and so on. Take a look at Scilab's help for more information about ascii. Now may be able to associate the letters with the corresponding data points.

```
-->eaplot([-10, 10, 30],[10, 11, 12, 13, 14],[ -5, 6, .1, -10, 15])
 ans  =

 d      a        @         b      e
```

meaplot([min, max, plotsize], [valuelist as matrix of n*m element(s)])

And again, just recall the examples from above or copy the line from here to Scilab. Then take out the ascii list "`[10, 11, 12, 13, 14],`" (incl. the trailing comma) and then press enter. And now you may missing the characters to give? Not need here, because this function applies automatically a letter per data point, starting with capital letters. By comparing this result with the one from above, it is easy to see that is same but with much less parameters.

```
-->meaplot([-10, 10, 30],[ -5, 6, .1, -10, 15])
 ans  =

 D      A       @        B     E
```

brlst([birthday], [optional calcdate], [optional plotwidth], [optional calc-range])

Now let's play with birthdays and biorythms. The next two functions doing same, just calculates the biorythm and plot it with ascii chars. The list is a standard Scilab list, while the second function plots biorythm formated. However, except the birthday all further parameters are optional for your convenience.

Notice that biorythm not only needs the birthday but also a calculation date. This can be any date as well as the date when you're born or somewhere in the future or so. But whenever you don't give a calculation date, the system clocks date will be used!

History of Biorhythms

The classical theory originated at the turn of the 19th century, between 1897 and 1902, from observational research.

Hermann Swoboda, professor of psychology at the University of Vienna, who was researching periodic variations in fevers, looked into the possibility of a rhythmic change in mood and health. He collected data on reaction to pain, outbreak of fevers, illnesses, asthma, heart attacks, and recurrent dreams. He concluded that there was a 23-day physical cycle and a 28-day emotional cycle.[citation needed]

Wilhelm Fliess, a nose and throat specialist and reportedly a numerologist, was independently researching the occurrences of fevers, recurrent illnesses and deaths in his patients. He too came to the conclusion that there was a 23 and a 28-day rhythm. Fliess' theories were of great interest and importance to Sigmund Freud during his early work in developing his psychoanalytic concepts.

Alfred Teltscher, professor of engineering at the University of Innsbruck, observed that his students' good days and bad days followed a rhythmic pattern of 33 days. Teltscher found that the brain's ability to absorb, mental ability, and alertness ran in 33 day cycles. In the 1920s, Dr. Rexford Hersey (psychologist; Pennsylvania, America) also reportedly made contributions to the classical theory.

These three biorhythms compose the classical theory. The classical theory has been studied, especially in Germany, Japan, and the United States, with conflicting results. Various modern derivatives exist of the classical theory.

Interpretation of the cycles:

- **Physical cycle (P)**
  - 23 days; Circavigintan
  - coordination
  - strength
  - well-being

- **Emotional cycle (E)**
  - 28 days; Circatrigintan
  - creativity
  - sensitivity
  - mood
  - perception
  - awareness

- **Intellectual cycle (I)**
  - 33 days; Circatrigintan
  - alertness
  - analytical functioning
  - logical analysis
  - memory or recall
  - communication

The following description of interpretation comes from:
http://www.halloran.com/whatarebiorhythms.htm

Originally, there were only three known biorhythms - Physical, Emotional and Intellectual. The Intuitional rhythm was discovered in recent years and is considered to be less important. It's cycle is 38 days long.
Intuitional cycle influences unconscious perception, hunches, instincts and 'sixth sense'. On low and especially critical days it may be difficult to do work related with art or other tasks that require lot of creativity and intuition.

Reading a Biorhythms Study

Three separate curves, marked **(P)**, for Physical rhythm, **(E)** for Emotional rhythm and **(I)**, for Intellectual rhythm. Depending upon the day of the month, the curves are either above or below the horizontal axis, which represents the mean level. The positive phase of each curve is above the axis and the recovery phase is below the axis. Biorhythms may therefore be used to forecast whether one will be in good or mediocre form, on a given date.

Consistency of Two Persons

By comparison of Biorhythms of two persons you can determine degree of their consistency. If all three curves are in the same phase, that is for both persons they are rising and falling more or less simultaneously, then there is a big chance that these people are compatible and can be a good partners. If phases of curves are almost opposite, then there is a big possibility that their relations could be in a disharmony.

For more thorough Interpretation

Advocates of Biorhythm theory do not only establish calendars for the high and low periods of our vitality, they also claim to be able to forecast critical dates upon which the individual, in a state of least resistance, will be exposed to more risks than normally. These dates do not correspond to the lowest points on the curves, as would seem logical, but to the points at which the curves intersect the axis, i.e. the phase changes. Obviously, when only one curve intersects the axis, which occurs regularly and frequently, then the risk on that day is not very great. However, if two curves change phase simultaneously, then particular care should be taken. This is, briefly, the theory of those who believe in Biorhythms, backed up by statistics.

However, the author does not take any responsibility of your calculated biorhythms and the way of its interpretation. Even not in any cases of bad days and not for good days too. Use this program at your totally own risk!

The list can growth very large, depending of given parameters! By default a biorhythms for 25 days will always be listed.

Example:

```
-->brlst([1961 1 26])
 ans  =


       ans(1)

 Neg                      0                      Pos   Date

       ans(2)

!                         |       E      P    I   | 2009-9-16  !
!                         |                       |            !
!                         |  E     P       I      | 2009-9-17  !
!                         |                       |            !
!                       E |  P          I         | 2009-9-18  !
!                         |                       |            !
!                   E   P |          I            | 2009-9-19  !
!                         |                       |            !
!             E  P        |     I                 | 2009-9-20  !
!                         |                       |            !
!         EP              | I                     | 2009-9-21  !
!                         |                       |            !
[Continue display? N (no) to stop, any other key to continue]
-->n
```

brfmt([birthday], [optional calcdate], [optional plotwidth], [optional calc-range])

By recalling the last example and then just change from “lst” to “fmt” press enter. The result now is somehow more readable.

```
-->brfmt([1961 1 26])
Neg                      0                       Pos   Date
|                        |                 P I   E|2009-09-10
|                        |                    PI E|2009-09-11
|                        |                      +P|2009-09-12
|                        |                    E  +|2009-09-13
|                        |                  E    PI|2009-09-14
|                        |             E     P  I |2009-09-15
|                        |        E      P    I   |2009-09-16
|                        |  E     P       I       |2009-09-17
|                      E |  P          I          |2009-09-18
|                E   P   |          I             |2009-09-19
|            E  P         |     I                  |2009-09-20
|        EP               | I                      |2009-09-21
|    +                 I  |                        |2009-09-22
| PE               I      |                        |2009-09-23
|+           I            |                        |2009-09-24
|+         I              |                        |2009-09-25
| E P  I                  |                        |2009-09-26
|   +  P                  |                        |2009-09-27
| I    E    P             |                        |2009-09-28
|I         E      P       |                        |2009-09-29
|I               E        @                        |2009-09-30
| I                    E  |     P                  |2009-10-01
|  I                      | E          P           |2009-10-02
|    I                    |      E          P      |2009-10-03
|        I                |           E       P    |2009-10-04
```

Available Constants

```
NA    = 6.0221367e23;...      // 1/gmol       = Avogadro's number
k     = 1.380658e-23;...      // J/K          = Boltzmann
Vm    = 22.4141;...           // gmol         = molar volume
R     = 8.31451;...           // J/(gmol*K)   = universal gas
StdT  = 273.15;...            // K            = std temperature
StdP  = 101.325;...           // kPa          = std pressure
rho   = 5.67051e-8;...        // W/(m^2*K^4)  = Stefan-Boltzmann
c     = 299792458;...         // m/s          = speed of light
e0    = 8.85418781761E-12;... // Fu0          = permittivity
u0    = 1.25663706144e-6;...  // H/m          = permeability
g     = 9.80665;...           // m/s^2        = accelration of gravity
G     = 6.67259E-11;...       // m^3/(s^2*kg) = gravitation
h     = 6.6260755E-34;...     // J*s          = Planck's constant
hbar  = 1.05457266E-34;...    // J*s          = Dirac's constant
q     = 1.60217733E-19;...    // C            = electronic charge
me    = 9.1093897E-31;...     // kg           = electron mass
qme   = 175881962000;...      // C/kg         = q/me ratio
mp    = 1.6726231E-27;...     // kg           = proton mass
mpme  = 1836.1527001;...      //              = mp/me ratio
a     = .00729735308;...      //              = fine structure
o     = 2.06783461E-15;...    // Wb           = mag flux quantum
F     = 96485.309;...         // C/gmol       = Faraday
R8    = 10973731.531;...      // 1/m          = Rydberg
a0    = .0529177249;...       // nm           = Bohr radius
uB    = 9.2740154E-24;...     // J/T          = Bohr magneton
uN    = 5.0507866E-27;...     // J/T          = nuclear magneton
l0    = 1239.8425;...         // nm           = Photon wavelength
f0    = 2.4179883E14;...      // Hz           = photon frequency
lc    = .00242631058;...      // nm           = Compton wavelength
pi2   = 6.28318530718;...     // radiands     = 2*phi radiands
c3    = .002897756;...        // m*K          = Wien's
kq    = .00008617386;...      // J/(K*C)      = k/q
esi   = 11.9;...              //              = dielectric constant
eox   = 3.9;...               //              = Si02 dielectric
constant
IO    = .000000000001;....    // W/m^2        = ref intentsity
JC    = 36525;...             //              = Julian Century
JY    = 365.25;...            //              = Julian Year
DY    = 365.242191;...        //              = Dropical Year
(equinoxes)
SIY   = 365.256363;...        //              = Sidereal year (360°)
AY    = 365.259635;...        //              = Anomalistic year
(nodes)
EY    = 346.620072;...        //              = Eclipse Year
SD    = [23 56 04 1];...      //              = Sideral Day
SDU   = 0.99726968;...        //              = Sideral Day in Solar
Units
SDI   = 1.0027377795;...      //              = Solar Day in Sidereal
Day Units
LYM   = 29.530588;...         //              = Lunar Synodic Month
LIM   = 27.321660;...         //              = Lunar Siderial Month
```

```
GPC   = 5029.0966')           //              = General Precession
(arc seconds per century)
```