Laboratory for Atmospheric Research Package

Version 0.3b

This file orients the user to available features of larpkg and provides guidance on using such features. Complete reference material for functions in this package, their syntax and directions for use is available in the lower-half of this file under this topic: LAR Package Reference

Overview

larpkg is a collection of Igor Pro functions focusing on time series, atmospheric fluxes, meterological parameters, and handling of a few less common scientific data formats. While some functions are decidedly narrow in focus, many are general and would be useful in other contexts. Most higher-level functions come with a host of optional arguments - it's usually worth your time to know what they do!

Prerequisites

The following procedure files are activated by larpkg because they possess functionality it depends on. It is not necessary to install these files as they are included with Igor Pro.

- RemovePoints (WaveMetrics Procedures:Data Manipulation:)

Some functions require third-party binary tools; see Functions with External Dependencies

Licenses

- larpkg is covered by the MIT license, http://mit-license.org/
- CmdUtils is covered by the GNU General Public License version 2 (GPLv2), http://www.gnu.org/licenses/gpl-2.0.html
- 7-Zip is covered by the GNU Lesser General Public License version 2.1 (LGPL), http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html; 7z.dll is also covered by the unRAR

restriction which prohibits the re-creation of the RAR compression scheme; more details at http://www.7-zip.org/license.txt.

Installation

larpkg is available as a compressed archive (.7z or .zip) which can be extracted using your favorite file archive utility. The archive directory structure looks like this:

`larpkg-<version>.7z\`	
`bin\`	
`cmdutils\`	Collection of open-source Windows utilities
`...`	
`7zip920.exe`	Installation binary for open-source archive utility 7-zip version 9.2
`larpkg Help.ihf`	This help file, compiled for use with Igor Pro
`larpkg Help.pdf`	This help file, printed to PDF/A format for reading without Igor Pro
`larpkg.ipf`	Procedure file containing core function set

The procedure file (.ipf) and help file (.ihf) can be used independently or together. If the subdirectory *bin* is not present in the same directory as the procedure file, some advanced functions

Temporary Install

Opening the procedure file and selecting 'Macros > Compile' is sufficient to use the functions in larpkg. Be aware, however, that if you write procedures which use functions from larpkg and fail to reopen the procedure file next time the experiment is opened, your procedure will not compile. Re-opening the procedure file will fix this.

Per-experiment Install (Adoption)

There are several reasons you might want to use this package on a per-experiment basis, for example:

- You rarely use larpkg but remembering to load it with certain experiments is a pain
- You frequently share experiments with users who do not have/want larpkg permanently installed
- You explicitly archive all procedures (perhaps for auditing purposes) and do not want any ambiguity introduced by using linked files or do not want to track and archive specific versions of larpkg.
- You have made changes to larpkg for use with a specific experiment and do not want to retain those changes otherwise.

To achieve a standalone or per-experiment installation, open the procedure file and give it active focus either by clicking on it or selecting it from 'Windows > Procedure Windows', then select 'File > Adopt Procedure'. More information about this is provided in the help topic Adopting a Procedure File.

Permanent

If you use this package frequently and would prefer it open automatically as Igor Pro loads, it should be installed as a 'global' procedure file. Extract the archive somewhere and place a shortcut to the that directory in the "WaveMetrics/Igor Pro User Files/Igor Procedures" folder inside your user's My Documents directory. See help topics Global Procedure Files and Igor Pro User Files for more information.

Network users can extract the archive to a network location and avoid having to copy the files to several workstations. In some cases, this is as simple as extracting it to the desktop or your My Documents folder.

Additional Resources

Don't overlook the fact that Igor Pro doesn't have everything activated upon installation. These resources are listed for informational purposes only and do not constitute endorsement of any particular package.

Other built-in functions

Some of the other interesting built-in packages include:

- RosePlot (WaveMetrics Procedures:Graphing:)
- New Polar Graphs (WaveMetrics Procedures:Graphing:New Polar Graph Procedures:)
- VDT, for Data Logger package (More Extensions:Data Acquisition)
- WildPoint (More Extensions:Data Analysis:)
- MLLoadWave (More Extensions:File Loaders:)
- TDM, for loading LabView files (More Extensions:File Loaders:)
- IgorThief, for extracting data from scanned graphs (WaveMetrics Procedures:File Input Output:)

Third-party extensions and snippets

WaveMetrics maintains an online repository of packages, the IgorExchange (igorexchange.com), where users can exchange code snippets and full-blown packages.

Some interesting projects/snippets include:

- DAQ Procedures -- GPIB, NIDAQmx, traditional NIDAQ, serial port, and VISA
- easyHTTP -- Connectivity to the internet, http/ftp/etc
- Time-Frequency Toolkit -- For investigating time-frequency domain of time series
- ADWU 1.0.ipf -- Download weather data based on airport codes (requires easyHTTP)
- Progress window.ipf -- Dislpay a pop-up progress window for long computations
- simple_spectral.ipf -- functions for spectra-related computations

Function Groups

This section provides a broad overview of available functions by group.

Bitwise Functions

There are several functions designed to make bitwise operations more clear. The relevant Igor help topic is Using Bitwise Operators. Reading specific flags from an instrument's diagnostic code is one example of where these functions could be applied (check Instrument-specific Functions below for applications).

BitString, ClearBit, SetBit, ShiftBit, TestBit

Eddy Covariance

Finally yes we have a section on eddy covariance

Cov, EC_co2, EC_WPL80, EC_WPL80_TS, ECLatentHeat, ECSensibleHeat, EstimateLag, IntervalDespikeHaPe, IntervalEC_co2, IntervalEC_WPL80, IntervalECLatentHeat, IntervalECSensibleHeat, IntervaluvwRotation

Files and Folders

This group includes functions to load and export data from a variety of common formats, as well as to manipulate the underlying file system to make handling data files easier. A utility to load waves from compressed archives without manually extracting them is under development, too.

BaleWaves, FlattenDir, ListFilesIn, LoadCSI, LoadLGR, LoadPicarro, PromptForFileList, UnzipArchive, UnzipArchivesInList

Graphical User Interfaces

These functions can generally be reached by a menu option but some can be incorporated into scripts too. For most purposes, the quick-n-dirty Prompted* functions are sufficient but if modeless, multipurpose GUIs are your intent, look at the *_Panel functions and at Modal and Modeless User Interface Techniques. (There are old & unfinished GUIs in project sandbox procedure file too.)

ListFilesIn_Panel, LoadLGR_Panel, PromptedLoadCSI, PromptedLatLongDistance

Igor Functions

These functions accomplish some specific, yet generic kind of task.

AddWaveRef, AllFoldersHere, AllWavesHere, ConcatAcrossDFRs, CountNans, GetDataFolderList, ListDataFoldersIn, MinFieldWidth, NewDataFolderX, PromptSetDataFolder, RemoveBlanks, RemoveNansW, RemoveWaveRef, StringFromMaskedVar, WaveList2Refs, WaveRefs2List

Interval Operators

The interval operators are intended for analyzing time series in consecutive, adjacent, identically sized subintervals. Sliding windows are not currently available, nor are they planned. A variety of functions are ported to Interval* operators already and conversion of additional functions is relatively straightforward.

There's actually a fair amount to be said, so check out About the Interval Operators.

IntervalBoundaries, IntervalCov, IntervalDespikeHaPe, IntervalEC_co2, IntervalECLatentHeat, IntervalECSensibleHeat, IntervalEC_WPL80, IntervalFrictionVelocity, IntervalMaxPntsGone, IntervalMean, IntervalObukhovLength, IntervalSdev, IntervalTimestamps, IntervalTKE, IntervaluvwRotation, IntervalWindDirMardiaSdev, IntervalWindDirScalarMeanSdev, IntervalWindDirUnitVectorMean, IntervalWindDirVectorMean, IntervalWindDirYamartinoSdev, IntervalWindSpeedScalarMean, IntervalWindSpeedScalarHMean, IntervalWindSpeedScalarSdev, IntervalWindSpeedVectorMean, IntervalWindSpeedPersist

Instrument-specific Functions

Some functions are for use with proprietary sensors.

DiagnoseCPC3776, DiagnoseCSAT3, DiagnoseLI7500, LoadCSI, LoadLGR, LoadPicarro

Micrometeorology

These functions complement the eddy covariance and wind functions nicely.

AmbientTemp, DensityOfAir, DewPoint, DielAverage, FrictionVelocity, IntervalFrictionVelocity, IntervalObukhovLength, IntervalTKE, LatentHeatVapH2O, MixingRatio, MixingRatioMF, MixingRatioVP, MoleFraction, ObukhovLength, ObukhovLengthTS, PotentialTemp, RelativeHumidity, SatVP, SpecificHumidity, SpecificHumidityVP, TKE, VirtualTemp, VirtualTempVP

Statistics/Math

These supplement the built-in math and statistics functions.

BankerRound, Cov, DielAverage, IntervalMean, IntervalCov, IntervalSdev

Time Functions

Time can be difficult to handle efficiently. Computational manipulation is rarely straightforward. Timezones are only further complicated by daylight savings time. While none of those issues are resolved here, there are some functions to assist in dealing with timestamps.

daqfactory2secs, DayOfWeek, DielAverage, doy2sec, EstimateLag, excel2secs, IntervalBoundaries, IntervalTimestamps, IsntChronological, string2secs, TimeRegEx

Truth Tests

These functions all return the true/false truth of some specific condition.

HasDuplicateDFRefs, HasDuplicateWRefs, HasNans, HaveNans, IsntChronological, SameNumCols, SameNumColsW, SameNumChunks, SameNumChunksW, SameNumLayers, SameNumLayersW, SameNumPnts, SameNumPntsW, SameNumRows, SameNumRowsW, SameXscale, TestBit

Wave Manipulation

Pretty empty topic.

EstimateLag, ResampleXY

Wind Statistics

These functions handle some of the basic wind data reduction routines and provide a few convenient conversions.

Cardinal2D, D2Cardinal, D2R, ModWD, R2D, WindDir, WindDirMardiaSdev, WindDirScalarMeanSdev, WindDirUnitVectorMean, WindDirVectorMean, WindDirYamartinoSdev, WindSpeed, WindSpeedScalarMean, WindSpeedScalarHMean, WindSpeedScalarSdev, WindSpeedVectorMean, WindSpeedPersist

Additions to Built-in Menus

In addition to functions, several useful menus are added:

- Under Data > Load Waves, there are links to guided file loading interfaces
  - Use *Load Campbellsci TOA5 (long header)* to quickly import data from one or many data files generated by CRBASIC dataloggers. A dialog with some basic options will appear and data will be dumped in the current data folder or subfolders, as specified.
  - Use *Extended Load Campbell TOA5* to get an ehanced file selection dialog with subfolder search and file name filter capabilities. Found files are displayed in a listbox for hand-editing; several sort options are available. Upon continuing, the standard dialog is displayed and loading continues normally.
  - Another enhanced dialog is displayed upon selecting *Load Los Gatos analyzer file*. This one is modeless (doesn't 'freeze' other things out) and offers useful options such as resampling data to a constant frequency.
- A new submenu More Tools is added to Analysis. Presently, it contains a tool to calculate Haversine (as-the-crow-flies) distance from pair of lat-long coordinates.
- The plot context-menus are expanded with useful scaling options.

About the Interval Operators

The interval operators are intended for analyzing time series in consecutive, adjacent, identically sized subintervals. Sliding windows are not currently available, nor are they planned. A variety of functions are ported to Interval* operators already and conversion of additional functions is relatively straightforward.

Arguments

The basic options are interval size and aligned/not aligned from midnight with respect to interval boundaries. Each Interval* function has a similar signature consisting of the source wave(s), the timestamp wave, the interval size in seconds, a boolean indicating interval alignment, and an optional wave describing the point boundaries of intervals. For example:

```
IntervalSomeFunction(srcw1, ..., timewave, interval, aligned [, bp])
```

The timestamp wave must be double-precision and contain values in Igor date/time format. This means it is represented as the number of seconds since 00:00:00 January 1, 1904. The values must be sorted chronologically but consistent spacing is not required. NAN is not permitted, although generally, NAN is permissible in data waves.

Interval can be a number or expression such as 30 * 60.

If boolean *aligned* is non-zero (true), intervals are aligned to 'whole' values with respect to midnight, otherwise interval boundaries are meted out with respect to point 0. For example, if interval = 30min and timewave[0] = 10:48:24, then if *aligned* is:

- zero/false: first period starts at 10:48:24, ends on the point before 11:18:24
- non-zero/true: first period starts at 10:30:00, ends on the point before 11:00:00

Note the time is anchored at start-of-period; see ***Timestamp anchor*** below for an explanation.

The optional argument *bp* is a two-column wave describing the lower and upper boundary points of each output interval; starting timestamps of each output interval are stored as X-scale values. When using several Interval* functions, a bounding points wave can be generated beforehand using IntervalBoundaries, then passed to argument *bp* to avoid re-calculating the boundary points wave for

each Interval* function called.

Results

In general, results are provided as a reference to a free wave (see Free Waves). This puts the onus on the user to explicitly retain results using Duplicate or MoveWave. A few Interval* functions modify source waves in-place and return a summary in the results; such behavior is made explicit in the function reference when applicable. Free waves can be retained as global waves like this:

```
MoveWave IntervalMean(...), resultswave   // use MoveWave to 'save' a free wave
MoveWave IntervalMean(...), $"results2"
Duplicate/O IntervalMean(...), resultswave // MoveWave cannot overwrite so use
Duplicate/O IntervalMean(...), $"results2" // Duplicate instead
```

Some notes about results waves:

- For missing periods of data, the results wave contains NANs. Since timestamps are stored in the x-scaling, the function IntervalTimestamps should be used before removing NANs from results waves.
- Since time information is stored in results waves' X-scaling, no timestamp wave is generated. For plots, the default is to use scaling for the X-axis. Tables only show values by default though so use IntervalTimestamps to create a wave to restore the double-click table ability.
- No distinction is made as to the % of data present in a particular interval. If the minimum number of records for a calculation to succeed are available, a result will be generated. The function IntervalMaxPntsGone can aid in filtering final results based on the amount of data missing in each interval.

Timestamp anchor

Since Interval* functions work on existing time-series, they possess an 'early' perspective and assign timestamps to the start of intervals. This is beneficial when plotting (such as having the cityscape mode appear properly) but is different from the convention often used by datalogging devices. Microloggers using CRBASIC(TM), for example, assign the timestamp after a period has been collected thus representing end-of-intervals.

Internals

Most of the magic actually occurs within IntervalBoundaries -- the function responsible for identifying the start and end of time intervals of an arbitrary size. This function is called silently by the Interval* functions when the optional argument *bp* is omitted. It may also be called explicitly and the results can be saved, then passed in as *bp*. The details cannot be covered here--see its function reference to know more.

The interior structure of the Interval* functions are remarkably similar. The functions begin by checking the validity of the arguments and generating missing optional arguments, if necessary. The size of the results wave is calculated from the look-up wave of interval boundary points, the results wave is created, and the wave's X-scaling is modified to represent start-of-period timestamps. Then, the wave of interval boundary points is stepped through one output interval at-a-time. For each, the lower and upper boundary point values are checked -- if either is NAN, then that interval is missing from the source data and the results wave receives a NAN for that interval. If both boundaries exist, then the source data wave is checked for NANs within that interval. When NANs are found, temporary copies of the data from that interval are made and NANs are removed before proceeding. If NANs are not found, the original data from that interval is used to proceed. At this point, the selected subset of data is dispatched to the "real" eponymous function. The results received from the dispatched function are stored in the results wave for that interval and the loop steps forward.

The 'skeleton' of Interval* functions:

```
Function/WAVE IntervalDoSomething( w1, w2, tstamp, interval, aligned [, bp] )
    wave w1, w2          // data sources
    wave/D tstamp        // double precision timestamp wave
    variable interval       // size of subinterval in seconds
    variable aligned     // nonzero to start/stop on multiples of <interval>
    wave bp              // optional wave of interval start/stop points
                         // as returned by IntervalBoundaries
```

```
    If ( !SameNumRows(tstamp, w1) || !SameNumRows(w1, w2) )
        print "IntervalDoSomething: input waves had different lengths - aborting"
        return NAN
    elseif ( !WaveExists(bp) )
        // if not provided, calculate interval boundaries
        wave bp = IntervalBoundaries( tstamp, interval, aligned )
    endif
    // 0-dimension (rows) of bp = # of output intervals
    Make/FREE/N=(DimSize(bp,0)) results
    // this maps timestamps into x-scaling; makes plotting sooo nice
    SetScale/P x, leftx(bp), deltax(bp), "dat", results

    variable oi, lo, hi
    for (oi=0; oi<DimSize(bp,0); oi+=1) // for each Output Interval
        lo = bp[oi][%lo]           // retrieve lower/upper boundaries
        hi = bp[oi][%hi]
        If ( numtype(lo) || numtype(hi) ) // if either boundary = NAN
            results[oi] = NAN   // then no source data this interval
            continue            // skip to next interval
        endif
        If ( HasNans(w1, p1=lo, p2=hi) || HasNans(w2, p1=lo, p2=hi) )
            // if there are NANs in this interval, we make temp. copies
            Duplicate/FREE/R=[lo,hi] w1, sub1
            Duplicate/FREE/R=[lo,hi] w2, sub2
            Make/FREE/WAVE/N=2 nanlist = {sub1, sub2}
            // and use special NAN-remover to preserve row alignment
            RemoveNansW( nanlist )
            // then save results from clean data
            results[oi] = DoSomething( sub1, sub2 )
        else
            // if no NANs, use data subset as-is
            results[oi] = DoSomething( w1, w2, p1=lo, p2=hi )
        endif
    endfor
    return results
End
```

Some notes about the example:

- For functions/operations without convenience window arguments (*p1* and *p2*) making temporary copies of a subrange may be necessary even if no NANs are detected.
- A special NAN-removal routine RemoveNansW is used to remove NANs from both waves while preserving the alignment of rows. That is, the corresponding row is removed from both waves for a NAN encountered in either (or both).
- The source data waves and timestamp wave are expected to be the same length. An error is displayed are different lengths.

Functions with External Dependencies

Some functions require external, third-party binaries for successful operation:

- UnzipArchive
- UnzipArchivesInList

Specifically, for the case of these two, the executable "recycle.exe" must be available system-wide. That is, larpkg expects recycle.exe to be findable using the system path (%PATH%). Extracting the cmdutils.7z archive into C:\WINDOWS\system32 will achieve this.

==========

- **LAR Package Reference**

AddWaveRef(*addref, wrefs, beforePoint*)

Returns *wrefs* after inserting wave reference *addref* at point *beforePoint*; remaining points

are shifted.

See Also:

InsertPoints, WAVE References

AllDataFoldersHere(*sortBy*)

Returns wave of data folder references to all data folders in the current data folder, sorted according to the value of *sortBy.* All folders named *Packages* are omitted.

type is a literal number which controls the sorting method:

-1:No sort (effectively sorts by creation date)
0:Default sort (ascending case-sensitive alphabetic ASCII sort)
1:Descending sort
2:Numeric sort
4:Case-insensitive sort
8:Case-sensitive alphanumeric sort using system script
16: Case-insensitive alphanumeric sort that sorts wave0 and wave9 before wave10.

or a bitwise combination of the above with the following restriction: only one of 2, 4, 8, or 16 may be specified. The legal values are thus -1, 0, 1, 2, 3, 4, 5, 8, 9, 16, and 17. Other values will produce undefined sorting criteria.

Examples

```
function baz()
    wave/DF w = AllFoldersHere()
    variable i
    for (i=0; i<numpnts(w); i+=1)
        print i, DataFolderDir(2, w[i])
    endfor
end
```

See Also:

SortList, AllWavesHere

AllWavesHere(*sortBy*)

Returns wave of references to all waves in the current data folder, sorted according to *sortBy,* which is a literal number controlling the sorting method:

-1: No sort (effectively sorts by creation date)
0:Default sort (ascending case-sensitive alphabetic ASCII sort)
1:Descending sort
2:Numeric sort
4:Case-insensitive sort
8:Case-sensitive alphanumeric sort using system script
16: Case-insensitive alphanumeric sort that sorts wave0 and wave9 before wave10.

or a bitwise combination of the above with the following restriction: only one of 2, 4, 8, or 16 may be specified. The legal values are thus 0, 1, 2, 3, 4, 5, 8, 9, 16, and 17. Other values will produce undefined sorting criteria.

See Also:

WaveList, SortList, WAVE References

AmbientTemp(*Ts, Q_*)

ThreadSafe

Return ambient temperature in Celcius, derived from sonic anemometer/thermometer measurement *Ts* and ambient specific humidity *Q_*. Function does internal conversion to perform calculation in Kelvin.

BaleWaves(*tstamp, refw, interval, options, formatTableName, destNameMask, destPath*)

The BaleWaves function writes each subinterval of the waves included in *refw* to a comma separated file. One file is generated for each interval.

****this documentation needs review!****

Parameters

tstamp is a double precision wave of sequential timestamps. It should not contain any empty values (NAN).

refw is a wave of wave references to include in the output file. If a timestamp column is desired, then *tstamp* should also be the first element of *refw*. The order of *refw* determines left-to-right order of columns in the output file.

interval is the length, in seconds, of each output file.

options is a literal number representing various bit combinations of:

Bit #	Bit Value	Option

1

formatTableName is a string containing the name of an existing table which reflects the desired formatting for each column. Formatting is copied according to column number so the order in this table should be the same as in *refw*.

destNameMask is a string file name mask following the same field code conventions as StringFromMaskedVar. A file extension should be included since none is appended. If no field code is used to distinguish output files, it is likely each file will overwrite the last and only the final file will remain.

destPath is a string containing a fully-qualified path to the desired output directory, or an empty string ("") to cause a prompt.

Details

The order of columns is determined by the order of waves in *refw.* The formatting of each column is copied from an existing table named *formatTableName.* If the table is not found, then -1 is returned.

The size of the interval is specified, in seconds, by *interval*. If *aligned* is nonzero, then intervals will start/stop on whole multiples of the interval counting from midnight; the default (0) is to start/stop relative to the value of *tstamp[0]*.

If *overwrite* is a positive non-zero value, files with conflicting names will be overwritten without prompt. A value of zero will result in a Save As.. prompt if file names conflict. Negative non-zero values are reserved for future use.

See Also:

SaveTableCopy, SaveData, Save, Tables

BankerRound(*inVal, place [, toOdd]*)

Rounds a numerical expression *inVal* to decimal column represented by 10^*place* using round-to-even rules. A nonzero value for optional parameter *toOdd* will cause round-to-odd behavior instead, if desired.

Details

Under normal conventions, the remainder one-half (0.5) is rounded upwards to the next whole number but this operation is not symmetric and such rounding can introduce an upwards bias, especially in large data sets. One solution is to round one-half towards the nearest even integer, resulting in equal probabilities the rounding will occur upwards versus downwards.

This is the default rounding mode used in IEEE 754 computing functions and operators.

Examples

```
foovar = BankerRound(foovar, 0)  // round to integer (10^0=1s)
foovar = BankerRound(foovar, 3)  // round to thousands (10^3=1000s)
```

```
wave0 = BankerRound( wave0[p], 3)  // round whole wave to thousands
```

References

Rounding <https://secure.wikimedia.org/wikipedia/en/wiki/Banker%27s_rounding>

See Also:

round, trunc, floor, ceil

BitString(*var, howMany [, maxLen]*)

Returns a string representation of *howMany* bits in *var*, starting with the least significant bit, written from right to left in 4 digit groups.

Details

Since bitwise operations only make sense on integers, *var* is treated as one.

The default behavior is to limit *howMany* to between 1 and 32 bits. Since most variables cannot hold more information, this makes sense but the optional parameter *maxLen* is provided as a way to bypass this if desired.

Examples

```
print WriteBits(21, 8)     //  0001 0101
print WriteBits(3021, 8)   //  1100 1101
print WriteBits(1+2+4, 4)  //       0111
print WriteBits(1+2+4, 6)  //    00 0111
```

See Also:

Using BitWise Operators, ClearBit, SetBit, ShiftBit, TestBit

Cardinal2D(*inStr*)

Returns numeric interpretation of cardinal wind direction (NW, S, SSE) in string *inStr* or returns NAN if *inStr* is not understood.

Details

The comparison to *inStr* is done case-insensitive with trailing spaces removed. Acceptable combinations of cardinal wind direction include, going clockwise:

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

Tip

This function can be used effectively in a wave assignment:

```
Make/N=(numpnts(cardWD)) numWD = Cardinal2D(cardWD[p]) // one fell swoop
```

See Also:

D2Cardinal, D2R, R2D, Wave Assignment

ClearBit(*var, bit*)

Returns *var* with bit number *bit* set to zero. Bit numbering is zero-indexed.

Details

Since bitwise operators only make sense on integers, *var* is treated as one and an integer is returned.

This is basically a wrapper for the bitwise complement (~) followed by the bitwise AND (&). It was derived from the example under Using Bitwise Operators.

See Also:

SetBit, ShiftBit, TestBit, BitString

ConcatAcrossDFRs(*DFRlist, destPath, overwrite, kill, [wfilter]*)

Returns

See Also:

???

CountNans(*theWave*)

Returns the total number of NANs in *theWave*.

See Also:

NAN, numtype

Cov(*wx, wy [, p1, p2]*)

Returns the covariance of waves *wx* and *wy*. If these waves are different lengths or contain empty values (NANs) then NAN is returned. The covariance is computed as

$$Cov(\vec{wx}, \vec{wy}) = \overline{wx' \cdot wy'} - \overline{wx} \cdot \overline{wy}$$

A point subrange may be specified using optional parameters *p1* and *p2*.

See Also:

IntervalCov, Mean, Variance

D2Cardinal(*inVal*)

Returns string containing the nearest cardinal wind direction (NW, S, SSE) to *inVal*, which is wrapped into the range of 0 <= *inVal* < 360. Valid output wind directions include:

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

Tip

This function could be used effectively in a wave assignment:

```
Make/N=(numpnts(WD)) labels = D2Cardinal( WD[p] )     // maybe for a graph
```

See Also:

Cardinal2D, D2R, R2D, Wave Assignment

D2R(*inVal*)

ThreadSafe

Returns *inVal* after converting from degrees to radians. Useful in wave assignments.

See Also:

R2D, Cardinal2D, D2Cardinal, Wave Assignment

daqfactory2secs(*timeval*)

ThreadSafe

Returns DAQFactory (TM) timestamps converted to Igor date/time value

See Also:

date2secs

DayOfWeek(*tstamp*)

Returns day of the week (1=Sunday,...,7=Saturday) based on Igor date/time value.

See Also:

????

DensityOfAir(*T_, P_, [, inMoles]*)

Returns density of air in g/m^3 or, if *inMoles* is non-zero, mol/m^3 using ideal gas law.

more detail here

See Also:

????

DespikeHaPe(...)

Performs 'soft' spike despiking routine.

more detail here

See Also:

????

DewPoint(*e_*)

Returns dew point based on vapor pressure.

more detail here

See Also:

????

DiagnoseCPC3776(*diagWord, option*)

Creates boolean waves denoting presence or absence of diagnostics flags.

more detail here

See Also:

????

DiagnoseCSAT3(*diagWord, option*)

Creates boolean waves denoting presence or absence of diagnostics flags.

more detail here

See Also:

????

DiagnoseLI7500(*diagWord, option*)

Creates boolean waves denoting presence or absence of diagnostics flags.

more detail here

See Also:

????

DielAverage(*wname, tstamp, mode*)

Computes means over 24-hour periods. No internal nan handling.

more detail here

See Also:

????

doy2sec(*doy, year*)

Converts a decimal day-of-year into Igor date/time value.

more detail here

See Also:

????

ECLatentHeat(*h2o, w_ [, T_, p1, p2]*)

Unverified

Return latent heat flux using eddy covariance.

more detail here

See Also:

????

ECSensibleHeat(*T_, w_, P_, Q_ [, p1, p2]*)

Unverified

Return sensible heat flux in W/m^2 using eddy covariance.

more detail here

See Also:

????

EC_WPL80(*meanRHOc, meanRHOv, meanRHOd, meanT, cov_w_rhoV, cov_w_T*)

Unverified

Return eddy covariance density corrections according to Webb, Pearman, Leuning (1980).

more detail here

See Also:

????

EC_WPL80_TS(*rhoC, rhoV, rhoD, T_, w_ [, p1, p2]*)

Unverified

Return eddy covariance density corrections for a time series according to Webb, Pearman, Leuning (1980).

more detail here

See Also:

????

EstimateLag(*baseWave, targetWave, keepResults*)

Returns estimate of record lag between two waves or NAN for error.

more detail here

See Also:

????

excel2secs(*serialdate [, use1904mode]*)

ThreadSafe

Converts "serial date" used by Microsoft Excel (TM) into Igor date/time value.

more detail here

See Also:

????

FlattenDir(*pathName, recurse, overwrite [, fileFilter, kill]*)

Flattens file directories by recursively lifting contents out of subfolders.

more detail here

See Also:

????

FrictionVelocity(*u_, v_, w_ [, p1, p2]*)

Returns friction velocity following the AMS definition.

more detail here

See Also:

????

GetDataFolderList(*[seePkgs]*)

Returns sorted, hierarchal list of all datafolders

more detail here

See Also:

????

HasDuplicateDFRefs(*dfrefs*)

Returns the element number of the first duplicate data folder reference in wave *refw*. If no duplicates are found, 0 is returned.

See Also:

DataFolderRefsEqual

HasDuplicateWRefs(*wrefs*)

Returns the element number of the first duplicate wave reference in wave *refw*. If no duplicates are found, 0 is returned.

See Also:

WaveRefsEqual

HasNans(*wname*)

Returns the element number of the first NAN found in *wname* or 0 if none are found. The special case of *wname[0]* = NAN returns -1.

See Also:

RemoveNaNs, BatchRemoveNANs

HaveNans(*wrefs [, p1, p2]*)

Returns the element number of the first NAN found in *wname* or 0 if none are found. The special case of *wname[0]* = NAN returns -1.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

IntervalBoundaries(*tstamp, interval, aligned*)

Returns a wave describing the starting point of each subinterval of length *interval* in *tstamp*. If *tstamp* has empty fields or out-of-order elements, NAN is returned.

Parameters

tstamp is a double-precision wave, ordered chronologically, without empty fields (NANs)

interval is the length of subinterval, in seconds

If *aligned* is zero, the first subinterval begins at the value of *tstamp[0]* and the following

subintervals start at *tstamp[0]+interval * n* where *n* increments for each following interval. If *aligned* is nonzero, subintervals start on whole multiples of the interval starting from the previous midnight.

Details

The wave *tstamp* is checked to ensure chronological order; if it fails this check, the value NAN is returned.

The lower boundary of the first interval is taken as *tstamp[0].* If *aligned* is nonzero, the upper boundary of the first interval is set to the next whole multiple of *interval* after the lower boundary; otherwise, the upper boundary is the lower boundary plus *interval*. Values in *tstamp* are compared to the upper boundary and when the next row value is >= upper boundary (or doesn't exist, ie. end of wave), the row corresponding to the lower boundary is recorded. The lower boundary is then set equal to the upper boundary, the upper boundary is recomputed and the procedure continues.

Examples

```
Function seehalfhours()
    wave timestamp
    wave/D w = TimeIntervalBoundaries(timestamp, 30*60, 1)
    variable i
    for (i=0; i<DimSize(w,0); i+=1)
        string t1 = secs2time(timestamp[ w[i][%lo] ],3,1)
        string t2 = secs2time(timestamp[ w[i][%hi] ],3,1)
        print/D "Interval",i,"Points",w[i][%lo],"-",w[i][%hi],"Time",t1,"-",t2
    endfor
End
```

Which produces output like this:

```
Interval 0 Points 0 - 13043 Time 12:08:15.6 - 12:29:59.9
Interval 1 Points 13044 - 31043 Time 12:30:00.0 - 12:59:59.9
Interval 2 Points 31044 - 49043 Time 13:00:00.0 - 13:29:59.9
 ...
Interval 525 Points 9445044 - 9463043 Time 10:30:00.0 - 10:59:59.9
Interval 526 Points 9463044 - 9481043 Time 11:00:00.0 - 11:29:59.9
Interval 527 Points 9481044 - 9490026 Time 11:30:00.0 - 11:44:58.2
```

See Also:

About the Interval Operators

IntervalCov(*wx, wy, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalDespikeHaPe(*wname, tstamp, interval, aligned [, multiplier, increment, passes, duration, bp]*)

Apply "soft spike" detection algorithm as described by Schmid, et al. (2000) to consecutive, equal intervals. Input wave is modified. This one is unique among Interval* functions because the source data is modified in-place and the number of points is not reduced by aggregation.

Parameters

The source data and timestamp waves are specified by *wname* and *tstamp*, respectively. The timestamp wave should be double-precision, chronological and contain no NANs.

Arguments *interval* and *aligned* represent the size of intervals considered, in seconds, and point of interval alignment. If *aligned* is non-zero, time boundaries of interval are determined with respect to midnight.

Specific behavior of the despiking algorithm is controlled through optional arguments *multiplier*, *increment*, *passes* and *duration*. These parameters are given the default values

described by Schmid, et al. if no value is explicitly provided.

Optional argument *bp* is a reference to wave of interval boundary points as would be returned by IntervalBoundaries.

Returns

Wave describing number of spikes detected and points removed in each interval. Each row of the output wave corresponds to an interval evaluated. The start-of-period timestamp for intervals is contained in the wave X-scaling. Two columns, "spikes" and "points", contain the total number of spikes identified and total number of points removed per interval, respectively. These two values will only be the same if every spike consists of exactly one point.

Example

```
wave tsw      // double-precision wave of Igor date/times (ie seconds since 1904)
              // times must be chronological but consistent spacing not required,
              // must not contain NANs
wave dataw    // data time series, same length as `timestamp`, might contain NANs
variable int = 5*60   //5min window
variable align = 1
wave bpw = IntervalBoundaries(tswave, int, align)

Duplicate/FREE dataw, dcpy
// IntervalDespikeHaPe returns free wave reference; use Duplicate to retain
// results (convert free wave to global wave)
Duplicate/O IntervalDespikeHaPe(dcpy, tsw, int, align, passes=5, bp=bpw), $"out"
```

Details

Excerpt from the original article:

Schmid, HaPe, C. Susan B. Grimmond, Ford Cropley, Brian Offerle, and Hong-Bing Su. "Measurements of CO2 and energy fluxes over a mixed hardwood forest in the mid-western United States." Agricultural and Forest Meteorology. 103 (2000): 357-374.

"For each 15min period and variable, the means and variances are calculated. From these diagnostics, a threshold for spikes is determined as a multiple of the standard deviation (3.6 S.D. initially, increased by 0.3 after each pass). On each pass, a soft spike is registered if the fluctuation from the mean is larger than the threshold value, and if the duration of the spike is three or fewer records, corresponding to a persistence of 0.3s, for the 10Hz sampling rate. Longer-lasting departures from the period mean are taken to indicate possible physical events. After each pass, if spikes are detected, the mean and variance are adjusted to exclude data marked as spikes and the process repeated, until either there are no more new spikes or the maximum or three iterations is completed (which is rarely the case)."

The function operates true to the above-quoted description but provides the means to operate using different numerical parameters. Avoid temptation to abuse the user-defined parameters (ie duration=5s would be bad).

See Also:

IntervalBoundaries, DespikeHaPe

IntervalECLatentheat(*h2o, w_, tstamp, interval, aligned [, T_, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalECSensibleHeat(*T_, w_, P_, Q_, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalEC_WPL80(*rhoC, rhoV, rhoD, T_, w_, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalFrictionVelocity(*u_, v_, w_, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalMaxPntsGone(*wrefs, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalMean(*wname, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalObukhovLength(*u_, v_, w_, Tv, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalSdev(*wname, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalTimestamps(*tstamp, interval, aligned, edge [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalTKE(*u_, v_, w_, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervaluvwRotation(*uvwMatrix, type, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalWindDirMardiaSdev(*Ux, Uy, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalWindDirScalarMeanSdev(*Ux, Uy, azimuth, flag, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalWindDirUnitVectorMean(*Ux, Uy, azimuth, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalWindDirVectorMean(*Ux, Uy, azimuth, flag, tstamp, interval, aligned [, bp]*)

Calculate resultant mean wind direction for consecutive intervals from continuous time series data. Returns free wave with time information embedded in X-scaling.

Parameters

Ux and *Uy* are waves representing time-series of horizontal wind speed components in an orthogonal space defined by *type*.

The variable *azimuth* represents the angle, measured (+) clockwise in degrees, between true north and the orientation of the sensor array.

The variable *type* defines the appropriate sensor coordinate system. Acceptable values are defined in detail by the WindDir function; currently the only available option is:

0 For use with:
Campbell Scientific Inc: CSAT3, CSAT3A
Applied Technologies: SATI models

See Also:

Another Topic

IntervalWindDirYamartinoSdev(*Ux, Uy, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalWindSpeedScalarMean(*Ux, Uy, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalWindSpeedScalarHMean(*Ux, Uy, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalWindSpeedScalarSdev(*Ux, Uy, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalWindSpeedVectorMean(*Ux, Uy, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IntervalWindSpeedPersist(*Ux, Uy, tstamp, interval, aligned [, bp]*)

This is a subtopic.

See Also:

Another Topic

IsntChronological(*wname*)

ThreadSafe

Returns the row number of the first element of *wname* to violate chronological order or of the first empty field (NAN), otherwise returns false (0). Special case of *wname[0]=NAN* returns -1.

Examples

```
If ( IsntChronological(timestamp) )
   // fix the wave or complain to user and quit
endif
```

See Also:

Sort, RemoveNaNs, BatchRemoveNaNs

LatentHeatVapH2O(*T_*)

Returns latent heat of vaporization of water in J/g based on temperature in Celcius.

See Also:

Another Topic

LatLongDistance(*lat1, long1, lat2, long2*)

Returns distance as-the-crow-flies (Haversine formula) between lat/long pairs 1&2, in meters.

See Also:

Another Topic

ListFilesIn(*pathName, fileFilter, fileExt, recurse, sortBy*)

The ListFilesIn function returns a semicolon separated list of all the files in *pathName* with extensions matching *fileExt*, optionally searching subfolders and sorting file/folder names.

Parameters

pathName is a string containing the name of an existing path as might be created using NewPath or choosing Misc.->New Path... from the Igor menus.

fileFilter is a string containing a regular expression which returned file names must match or a zero-length string ("") to match all files. The format of the regular expression is the same as for Grep and GrepList. See Regular Expressions for more details.

fileExt is a string, up to four-characters, specifying the extension of the file type to list or "????" to list all file types. See the *extension* parameter of IndexedFile for more details.

recurse is a variable indicating maximum level of depth to use while searching subfolders. If its value is zero, no subfolders are searched. A positive integer *n* will result in *n* levels of subfolders being searched in a recursive manner; a negative integer will result in a search of all levels.

sortBy is a variable specifying the sort mode to apply: a value of -1 will sort files and folders

according to creation date while other valid values are identical to *options* in SortList.

See Also:

IndexedFile, IndexedDir, NewPath, SortList, File Types and Extensions, ParseFilePath, Path Seperators, Regular Expressions

ListDataFoldersIn(*folder*)

Recursively searches for data folders in *folder* and returns hierarchal, semicolon-separated list.

See Also:

Another Topic

LoadCSI(*fileList, fileType, overwrite, convertTS, options [, baseSFname]*)

The LoadCSI function loads data from one or more Campbell Scientific Loggernet files into Igor waves and, optionally, converts string timestamps into their double-precision representation.

If successful, 0 is returned; otherwise, -1 is returned.

Details

The string *fileList* contains a semicolon-separated list of full file paths (as might be returned by PromptForFileList or ListFilesIn). Each file is loaded assuming the first column contains quoted timestamps and subsequent columns contain numeric data. Most of the time this is desired since Campbell places quotes around NANs and a forced numeric load prevents columns beginning with NAN from being interpreted as text. If the data file contains a legitimate text column, such as a log file might, then LoadCSI will not work for your situation.

If the list contains one file, all waves are loaded into the current directory; otherwise, one subfolder will be created in the current directory for each file. Subfolder creation can be forced for a single-item *fileList* using bit 5 of *options*. If subfolders are created, the name of each is derived from a strict cleanup of each file name. In the event many identically-named files are loaded, the name of each file's parent folder can be used instead by setting bit 2 of *options*.

authors note: explore and describe what happens if multiple files are loaded from the same directory and the subfolder names are set to file's parent subfolder. does behavior change in combination with overwrite/skip parameters?

maybe add a convert variable since it's so common to want to do and not quite the same as the subfolder arguments

Using bit 3 of *options* and optional parameter *baseSFname* provides a third, independent method of specifying subfolder names. In this case, each subfolder is given the name returned by StringFromMaskedVar(*baseSFname, <index>, fixNNwidth=<minWidth>*) where *<index>* is the zero-based index of the current file in *fileList*, *<minWidth>* is automatically set to the shortest necessary field width, and *baseSFname* is, by default, "loadCSI_file\nn".

If you use bit 3 and change *baseSFname*, do not forget to include a field code or the same subfolder name will be generated for each file. Depending on combinations of other settings, this could result in each new file overwriting the last, only the first file loading or some other, undefined behavior.

fileType specifies the file format. Presently, only two file tyes are supported.

0: TOA5: long (4-line) header
1: TOACI1: short (2-line) header
2: TOB1: table-oriented binary [**not implemented**]
3: TOB2: table oriented binary [**not implemented**]
4: TOB3: table-oriented binary [**not implemented**]
5: CSIXML: extensible markup language [**not implemented**]

If *overwrite* is nonzero, existing subfolders and waves will be overwritten; otherwise, unique

names will be used. Setting bit 4 of *options* will cause a file to be skipped if the target subfolder already exists, rather than creating a unique subfolder. This could speed up loading new data into an existing experiment.

options is a bitwise combination controlling several auxilary options. See Setting Bit Parameters for details about bit settings.

Bit 0:	1	Convert loaded timestamps into an Igor date/time wave named "timestamp". Since Campbell Scientific places double-quotes around timestamps, it is not possible to load them directly as Igor date/time values.
Bit 1:	2	Keep the original string timestamp wave, renamed "timestamp_STR". This bit is ignored if Bit 0 is not set.
Bit 2:	4	Derive subfolder names from file's parent folder name instead of using file's name.
Bit 3:	8	Derive subfolder names from *baseSFname* and the file's index # in *fileList*. See above and StringFromMaskedVar for details.
Bit _:	16	will be reserved for concatenation method
Bit _:	32	will be reserved for something else
Bit 4:	64	Skip existing subfolders instead of generating unique name when *overwrite* is false.
Bit 5:	128	Force individual files to load in subfolders as if *fileList* contained multiple items.

The above bits may be combined with these exceptions:

- if bit 0 is not set, bit 1 is ignored
- if bit 2 is set, bit 3 is ignored
- if *overwrite* is true, bit 4 is ignored

See Also:

LoadWave, Setting Bit Parameters

LoadLGR(*fileList, modelName, overwrite, concat, resamp, options [, baseSFname, B]*)

Reads data file from Los Gatos Research analyzer

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

LoadPicarro(*fileList, modelName, overwrite, concat, tsconv, options [, baseSFname, B]*)

Reads data file from Picarro trace gas analyzers; only CO2/CH4/H2O (G2301-f) supported right now.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

MinFieldWidth(*num*)

Returns the minimum number of fields necessary to hold the absolute value of integer *num*.

Examples

```
print MinFieldWidth( 0 )      // 0
print MinFieldWidth( 0.5 )    // 0      fractional portions are ignored
print MinFieldWidth( 2 )      // 1
print MinFieldWidth( -2 )     // 1      negative signs are ignored
print MinFieldWidth( 1000 )      // 4
print MinFieldWidth( 999 )    // 3
print MinFieldWidth( 999.99 ) // 3
```

See Also:

StringFromMaskedVar

MixingRatio(*C_, T_, P_, h2o [, inMass]*)

Returns dimensionless molar mixing ratio of constituent C as the ratio of moles of C to moles of dry air. Pass non-zero value to *inMass* to receive mass mixing ratio.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

MixingRatioMF(*C_, T_, P_, MFw*)

Returns dimensionless molar mixing ratio of constituent C as the ratio of moles of C to moles of dry air.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

MixingRatioVP(*e_, P_ [, MW]*)

Returns dimensionless mass mixing ratio of a vapor as the ratio of mass of vapor to the mass of dry air. Assumes vapor is water; specify molecular weight of alternate vapor using *MW* if desired.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

ModWD(*inVal*)

Returns *inVal* after adding or subtracting 360 to bring within the range 0 <= *inVal* < 360.

See Also:

Mod

MoleFraction(*C_, T_, P_*)

Returns dimensionless mole fraction of constituent C as ratio of moles of C to total moles in system.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

NewDataFolderX(*destPath*)

Returns a data folder reference to an eXtended folder path specified by string *destPath*. This path can be absolute (from root:) or relative (start with :) and may contain multiple levels separated by a colon (:).

Details

If the folder does not exist it is created, along with all necessary parent folders; if it does exist, it is switched to quietly. Liberal folder names can quoted or unquoted. Pairs of colons are interpreted as 'up directory' so :: refers to a parent and ::: refers to a parent's parent.

Examples

```
SetDataFolder NewDataFolderX("root:Packages:mynewfolder:temp")
DFREF outputDir = NewDataFolderX(":group"+num2istr(grp)+":run"+num2istr(run))
DFREF inputDF = NewDataFolderX($inputStr)
```

See Also:

Data Folders, Data Folder References, NewDataFolder, SetDataFolder

NewProgressWindow()

Creates a new progress window with 0% progress and returns name of the window.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

ObukhovLength(*frictionVelocity, meanTv, cov_w_Tv*)

Returns the Monin-Obukhov length in meters from scalar values.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

ObukhovLengthTS(*u_, v_, w_, Tv [, p1, p2]*)

Returns the Monin-Obukhov length in meters from a time-series.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

NotAllSameLength(*refw`*)

Returns the element of the first wave in wave *refw* to have a different length. If all waves are the same length, a 0 is returned.

See Also:

numpnts

PotentialTemp(*T_, P_ [, P0]*)

Returns the potential temperature; standard pressure used is P0 = 1000mb by default.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

PromptSetDataFolder()

Prompts user to select a datafolder, switches to chosen folder and returns a DFREF to the user's **starting** location. This may be useful if you want to switch back later.

Examples

```
DFREF sav0 = PromptChooseFolder()
// mess with some data
SetDataFolder sav0
```

See Also:

Data Folder References, SetDataFolder, GetDataFolderDFR

PromptForFileList(*msg*)

Displays an Open File dialog box to user. Returns semicolon separated list of files selected or an empty string ("") if the user cancels. String parameter *msg* is shown in the title bar of the dialog box.

See Also:

Open, Displaying a Multi-Selection Open File Dialog

R2D(*inVal*)

ThreadSafe

Returns *inVal* after converting from radians to degrees. Useful in wave assignments.

See Also:

D2R, Cardinal2D, D2Cardinal, Wave Assignment

RelativeHumidity(*e_, T_*)

Returns relative humidity based on ambient temperature in Celcius and water vapor pressure in mbar or hPa.

more detail here

See Also:

RemoveNaNs, BatchRemoveNANs

RemoveBlanks(*theWave*)

ThreadSafe

Removes blank ("") rows in text wave *theWave* and returns the number of rows removed.

This function was inspired by RemoveNaNs in <Remove Points.ipf>

See Also:

RemoveNaNs

RemoveNansW(*wrefs*)

Removes NANs from waves while preserving correct alignment of values across rows.

Details

Checks each wave in *refw* for NAN values and, if found, removes that row from each wave in *refw*. The operation is performed so the alignment of data values remains consistent. This may be an important consideration when preparing sets of XY data for operations which do not accept NAN (such as Mean).

See Also:

RemoveNaNs

RemoveWaveRef(*remref, wrefs*)

Remove any references in *wrefs* which are equivalent to *remref*.

See Also:

????

ReplaceWaveValues(*wrefs, withVal, mode, val1, val2 [, p1, p2]*)

need description

See Also:

????

ResampleXY(*tstamp, wrefs, newRate*)

need description

See Also:

????

SameNumCols(*w1, w2*)

ThreadSafe

Returns truth or falsehood of whether waves *w1* and *w2* have equal number of columns.

See Also:

numpnts

SameNumColsW(*wrefs*)

ThreadSafe

Returns truth or falsehood of whether all waves in *wrefs* have equal number of columns.

See Also:

numpnts

SameNumChunks(*w1, w2*)

ThreadSafe

Returns truth or falsehood of whether waves *w1* and *w2* have equal number of chunks.

See Also:

numpnts

SameNumChunksW(*wrefs*)

ThreadSafe

Returns truth or falsehood of whether all waves in *wrefs* have equal number of chunks.

See Also:

numpnts

SameNumLayers(*w1, w2*)

ThreadSafe

Returns truth or falsehood of whether waves *w1* and *w2* have equal number of layers.

See Also:

numpnts

SameNumLayersW(*wrefs*)

ThreadSafe

Returns truth or falsehood of whether all waves in *wrefs* have equal number of layers.

See Also:

numpnts

SameNumPnts(*w1, w2*)

Returns truth or falsehood of whether waves *w1* and *w2* have equal number of points.

See Also:

numpnts

SameNumPntsW(*wrefs*)

Returns truth or falsehood of whether all waves in *wrefs* have equal number of points.

See Also:

numpnts

SameNumRows(*w1, w2*)

ThreadSafe

Returns truth or falsehood of whether waves *w1* and *w2* have equal number of rows.

See Also:

numpnts

SameNumRowsW(*wrefs*)

ThreadSafe

Returns truth or falsehood of whether all waves in *wrefs* have equal number of rows.

See Also:

numpnts

SameXscaling(*w1, w2*)

ThreadSafe

Returns truth or falsehood of whether waves *w1* and *w2* have equivalent X scaling.

See Also:

wave scaling

SatVP(*T_*)

Returns saturation water vapor pressure based on ambient temperature in Celcius.

See Also:

????

serial2secs(*serialdate*)

Returns Igor date/time value corresponding to the serial date *serialdate*, which is defined as the number of seconds since midnight, January 1, 1970. Serial dates are used by Excel and DAQFactory.

This function is useful in wave assignments.

Examples

```
timestamp = serial2secs( timeW[p] )
```

See Also:

Date/Time Waves, date2secs

SetBit(*var, bit*)

Returns *var* with bit number *bit* set to 1. Bit numbering is zero-indexed.

Details

Since bitwise operators only make sense on integers, *var* is treated as one and an integer is returned.

This is basically a wrapper for the bitwise OR (|). It was derived from the example under Using Bitwise Operators.

See Also:

ClearBit, ShiftBit, TestBit, BitString

ShiftBit(*var, by*)

Returns variable *var* after shifting bits *by* number of places. The shift will occur leftwards, increasing *var* if *by* is positive; rightwards, decreasing *var,* if *by* is negative.

Details

Since bitwise operators only make sense on integers, *var* is treated as one. However, it is still possible to recieve fractional values when shifting rightwards. [*authors note: find out why*]

This is basically a wrapper for multiplication and division by powers of 2, which has the same effect as multiplication and division by powers of 10 in decimal. It is derived from the example under Using Bitwise Operators.

See Also:

ClearBit, SetBit, TestBit, BitString

SpecificHumidity(*R_*)

Returns dimensionless specific humidity ratio, defined as mass of water vapor to total mass of system.

See Also:

wave scaling

SpecificHumidityVP(*e_, P_*)

Returns an approximation of specific humidity.

See Also:

MixingRatioVP, SpecificHumidity

string2secs(*timestring, format*)

Returns Igor date/time value of timestamp represented in string *timestring* using the regular expression in string *format*.

Any double quotes in *timestring* are ignored. The regular expression in *format* follows the conventions of sscanf.

Tips

A suitable regular expression for *format* is probably already available in TimeRegEx.

This function is useful in wave assignments.

Examples

```
Make/D/N=(numpnts(timestampStr)) timestampVal
timestampVal = string2secs( timestampStr[p], TimeRegEx(0) )
```

See Also:

Date/Time Waves, date2secs, sscanf, TimeRegEx

StringFromMaskedVar(*maskStr, inVal, [, fixNNwidth]*)

Returns *maskStr* after replacing appropriate field codes with values derived from *inVal*.

Details

The string *maskStr* can contain a combination of literal text and zero, one or more field codes described below. Each instance of a field code will be replaced with the indicated value derived from *inVal*. Field codes are case-sensitive.

Field Code

Fixed width	Variable width	Value interpreted from *inVal*
\nn	\n	*inVal* as integer *
\YYYY		four-digit year
\YY		two-digit year
\MM	\M	month
\DD	\D	day of month
\DDD	\ddd	day of year **not implemented**

\hh	\h	hours, military style
\hhn	\hn	hours, normal style
\mm	\m	minute
\ss	\s	second

*__Note:__ If optional parameter *fixNNwidth* is not specified, a variable width field will be used instead. If the necessary field width is unknown, it can be found by passing the highest possible value of *inVal* to MinFieldWidth.

Tip

This function was designed to generate an output file name containing timestamp elements but it will create sequential file names too.

Examples

```
print StringFromMaskedVar("test_\YYYY\MM\DD_\hhn\mm", date2secs(1937, 07, 18)+
    (14*3600+8*60+37))            // test_19370718_0208      << note: 2PM

print StringFromMaskedVar("test_\YYYY\MM\DD_\hh\mm", date2secs(1937, 07, 18)+
    (14*3600+8*60+37))            // test_19370718_1408

print StringFromMaskedVar("\YY_\M_\D_\h_\m_\s.dat", date2secs(1937, 07, 18)+
    (14*3600+8*60+37))            // 37_7_18_14_8_37.dat

print StringFromMaskedVar("file_\nn.txt", 4, fixNNwidth=3) //  file_004.txt
```

See Also

MinFieldWidth, ReplaceString

TestBit(*var, bit*)

Returns the truth (1) or nontruth (0) of whether bit number *bit* is set in *var*.

Details

Since bitwise operands only make sense on integers, *var* is treated as one.

This is basically a wrapper for bitwise AND (&). It was derived from the example under Using Bitwise Operators.

See Also:

ClearBit, SetBit, ShiftBit, BitString

TimeRegEx(*choice*)

Returns string containing one of several time stamp regular expressions compatible with sscanf.

choice	Format matching regular expression
0	YYYY-MM-DD hh:mm:ss.sss (ISO, CampbellSci)
1	reserved
2	reserved
3	reserved
4	reserved
5	reserved

See Also:

string2secs, sscanf

TKE(*u_, v_, w_ [, p1, p2]*)

Returns turbulent kinetic energy derived from orthogonal wind components.

See Also:

????

UnzipArchive(*srcFileStr, destFolderStr, overwrite, flatten*)

Unzips archive *srcFileStr* to directory *destFolderStr* and returns semicolon-separated list of

unzipped files.

See Also:

????

UnzipArchivesInList(*fileList*)

Unzips any archive found in *fileList* to a temp directory, then replaces name of archive in *fileList* with list of files inside that archive.

See Also:

????

UpdateProgressWindow(*name, val1, val2 [, msg, noKill]*)

Updates the named progress window.

See Also:

????

uvwRotation(*uvwMatrix, type*)

rotates waves

References

See Also:

another topic

VirtualTemp(*T_, R_*)

Returns virtual temperature in Celcius given ambient temp in Celcius and water vapor mixing ratio.

See Also:

????

VirtualTempVP(*T_, e_, P_*)

Returns virtual temperature in Celcius given ambient temp in Celcius and water vapor pressure and ambient pressure.

See Also:

????

WaveList2Refs(*wlist, makeFreeCopies*)

Returns wave of references to waves listed *wlist*.

See Also:

????

WaveRefs2List(*wrefs, fullName*)

Returns string list of waves in *wrefs*, possibly quoting.

See Also:

????

WindDir(*Ux, Uy, azimuth, type*)

Returns direction wind is coming from in the range 0º <= *WD* < 360º based on horizontal components, *Ux* and *Uy*, and sensor orientation, *azimuth*. If either *Ux* or *Uy* is NAN, then

NAN is returned.

Parameters

Ux and *Uy* are variables representing horizontal wind components in an orthogonal space defined by the variable *type*.

Variable *azimuth* represents the angle, measured (+) clockwise in degrees, between north and the orientation of the sensor array. The function has no way to distinguish between true and magnetic north and, frankly, does not care--that is the responsibility of the user.

Variable *type* defines the coordinate system of *Ux* and *Uy* with respect to different sensor geometries. This determines exactly how the calculation occurs.

Type	Description
0	For use with: - Campbell Scientific Inc.: CSAT3, CSAT3A - Applied Technologies, Inc.: SATI-* models This right-handed coordinate system defines +*Ux* as wind into the array, parallel to the sensor boom. Looking into the array along the boom, +*Uy* is oriented leftwards while +*Uz* is upward. The wind direction is calculated as: $$WD = atan2(U_x, U_y) * \frac{180}{\pi} + azimuth + 90°$$
*	No other *type* values are defined yet. It could be expanded as necessary.

References

See Also:

IntervalWindDirUnitVectorMean, IntervalWindDirVectorMean, WindDirUnitVectorMean, WindDirVectorMean

WindDirMardiaSdev(*Ux, Uy [, p1, p2]*)

This is a subtopic.

See Also:

Another Topic

WindDirScalarMeanSdev(*Ux, Uy, azimuth, flag [, p1, p2]*)

This is a subtopic.

See Also:

Another Topic

WindDirUnitVectorMean(*Ux, Uy, azimuth, flag [, p1, p2]*)

Returns the direction wind is coming from in the range 0 <= WD < 360

See Also:

Another Topic

WindDirVectorMean(*Ux, Uy, azimuth, type [, p1, p2]*)

Return resultant wind direction in the range 0º <= WD < 360º based on time-series of horizontal wind components, *Ux* and *Uy*, and sensor orientation, *azimuth*. If *Ux* or *Uy* contains NAN points or are not the same length, NAN is returned.

Parameters

Ux and *Uy* are waves representing time-series of horizontal wind components in an orthogonal space defined by the variable *type*.

Variable *azimuth* represents the angle, measured (+) clockwise in degrees, between north

and the orientation of the sensor array. The function has no way to distinguish between true and magnetic north and, frankly, does not care--that is the responsibility of the user.

Variable *type* defines the coordinate sytem of Ux and Uy with respect to different sensor geometries. This determines exactly how the calculation occurs.

Type	Description
0	For use with: - Campbell Scientific, Inc.: CSAT3, CSAT3A - Applied Technologi, Inc.: SATI-* models This right-handed coordinate system defines +*Ux* as wind into the array, parallel to the sensor boom. Looking into the array along the boom, +*Uy* is oriented leftwards while +*Uz* is upward. The wind direction is calculated as: $$WD = atan2(\bar{U}_x, \bar{U}_y) * \frac{180}{\pi} + azimuth + 90°$$
*	No other types are defined yet.

Variables *p1* and *p2* are optional values which permit the user to confine the calculation to a specific range of points. Both *p1* and *p2* are treated as inclusive boundaries; their default values correspond to the entire wave. These arguments are used internally by IntervalWindDirVectorMean.

See Also:

IntervalWindDirVectorMean, WindDir

WindDirYamartinoSdev(*Ux, Uy [, p1, p2]*)

This is a subtopic.

See Also:

Another Topic

WindSpeed(*Ux, Uy*)

This is a subtopic.

See Also:

Another Topic

WindSpeedScalarMean(*Ux, Uy [, p1, p2]*)

This is a subtopic.

See Also:

Another Topic

WindSpeedScalarHMean(*Ux, Uy [, p1, p2]*)

This is a subtopic.

See Also:

Another Topic

WindSpeedScalarSdev(*Ux, Uy [, p1, p2]*)

This is a subtopic.

See Also:

Another Topic

WindSpeedVectorMean(*Ux, Uy [, p1, p2]*)

This is a subtopic.

See Also:

Another Topic

WindSpeedPersist(*Ux, Uy [, p1, p2]*)

This is a subtopic.

See Also:

Another Topic