Agilent Technologies

Advanced Design System 2011.01

Feburary 2011
Simulator Expressions

© Agilent Technologies, Inc. 2000-2011
5301 Stevens Creek Blvd., Santa Clara, CA 95052 USA
No part of this documentation may be reproduced in any form or by any means (including electronic storage and retrieval or translation into a foreign language) without prior agreement and written consent from Agilent Technologies, Inc. as governed by United States and international copyright laws.

Acknowledgments

Mentor Graphics is a trademark of Mentor Graphics Corporation in the U.S. and other countries. Mentor products and processes are registered trademarks of Mentor Graphics Corporation. * Calibre is a trademark of Mentor Graphics Corporation in the US and other countries. "Microsoft®, Windows®, MS Windows®, Windows NT®, Windows 2000® and Windows Internet Explorer® are U.S. registered trademarks of Microsoft Corporation. Pentium® is a U.S. registered trademark of Intel Corporation. PostScript® and Acrobat® are trademarks of Adobe Systems Incorporated. UNIX® is a registered trademark of the Open Group. Oracle and Java and registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. SystemC® is a registered trademark of Open SystemC Initiative, Inc. in the United States and other countries and is used with permission. MATLAB® is a U.S. registered trademark of The Math Works, Inc.. HiSIM2 source code, and all copyrights, trade secrets or other intellectual property rights in and to the source code in its entirety, is owned by Hiroshima University and STARC. FLEXlm is a trademark of Globetrotter Software, Incorporated. Layout Boolean Engine by Klaas Holwerda, v1.7 http://www.xs4all.nl/~kholwerd/bool.html . FreeType Project, Copyright (c) 1996-1999 by David Turner, Robert Wilhelm, and Werner Lemberg. QuestAgent search engine (c) 2000-2002, JObjects. Motif is a trademark of the Open Software Foundation. Netscape is a trademark of Netscape Communications Corporation. Netscape Portable Runtime (NSPR), Copyright (c) 1998-2003 The Mozilla Organization. A copy of the Mozilla Public License is at http://www.mozilla.org/MPL/ . FFTW, The Fastest Fourier Transform in the West, Copyright (c) 1997-1999 Massachusetts Institute of Technology. All rights reserved.

The following third-party libraries are used by the NlogN Momentum solver:

"This program includes Metis 4.0, Copyright © 1998, Regents of the University of Minnesota", http://www.cs.umn.edu/~metis , METIS was written by George Karypis (karypis@cs.umn.edu).

Intel@ Math Kernel Library, http://www.intel.com/software/products/mkl

SuperLU_MT version 2.0 - Copyright © 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy). All rights reserved. SuperLU Disclaimer: THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS

INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

7-zip - 7-Zip Copyright: Copyright (C) 1999-2009 Igor Pavlov. Licenses for files are: 7z.dll: GNU LGPL + unRAR restriction, All other files: GNU LGPL. 7-zip License: This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful,but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA. unRAR copyright: The decompression engine for RAR archives was developed using source code of unRAR program.All copyrights to original unRAR code are owned by Alexander Roshal. unRAR License: The unRAR sources cannot be used to re-create the RAR compression algorithm, which is proprietary. Distribution of modified unRAR sources in separate form or as a part of other software is permitted, provided that it is clearly stated in the documentation and source comments that the code may not be used to develop a RAR (WinRAR) compatible archiver. 7-zip Availability: http://www.7-zip.org/

AMD Version 2.2 - AMD Notice: The AMD code was modified. Used by permission. AMD copyright: AMD Version 2.2, Copyright © 2007 by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff. All Rights Reserved. AMD License: Your use or distribution of AMD or any modified version of AMD implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies.User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. AMD Availability: http://www.cise.ufl.edu/research/sparse/amd

UMFPACK 5.0.2 - UMFPACK Notice: The UMFPACK code was modified. Used by permission. UMFPACK Copyright: UMFPACK Copyright © 1995-2006 by Timothy A. Davis. All Rights Reserved. UMFPACK License: Your use or distribution of UMFPACK or any modified version of UMFPACK implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at

your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. UMFPACK Availability: http://www.cise.ufl.edu/research/sparse/umfpack UMFPACK (including versions 2.2.1 and earlier, in FORTRAN) is available at http://www.cise.ufl.edu/research/sparse . MA38 is available in the Harwell Subroutine Library. This version of UMFPACK includes a modified form of COLAMD Version 2.0, originally released on Jan. 31, 2000, also available at http://www.cise.ufl.edu/research/sparse . COLAMD V2.0 is also incorporated as a built-in function in MATLAB version 6.1, by The MathWorks, Inc. http://www.mathworks.com . COLAMD V1.0 appears as a column-preordering in SuperLU (SuperLU is available at http://www.netlib.org). UMFPACK v4.0 is a built-in routine in MATLAB 6.5. UMFPACK v4.3 is a built-in routine in MATLAB 7.1.

Qt Version 4.6.3 - Qt Notice: The Qt code was modified. Used by permission. Qt copyright: Qt Version 4.6.3, Copyright (c) 2010 by Nokia Corporation. All Rights Reserved. Qt License: Your use or distribution of Qt or any modified version of Qt implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the
terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies.User
documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. Qt Availability: http://www.qtsoftware.com/downloads Patches Applied to Qt can be found in the installation at: $HPEESOF_DIR/prod/licenses/thirdparty/qt/patches. You may also contact Brian Buchanan at Agilent Inc. at brian_buchanan@agilent.com for more information.

The HiSIM_HV source code, and all copyrights, trade secrets or other intellectual property rights in and to the source code, is owned by Hiroshima University and/or STARC.

Errata The ADS product may contain references to "HP" or "HPEESOF" such as in file names and directory names. The business entity formerly known as "HP EEsof" is now part of Agilent Technologies and is known as "Agilent EEsof". To avoid broken functionality and to maintain backward compatibility for our customers, we did not change all the names and labels that contain "HP" or "HPEESOF" references.

Warranty The material contained in this document is provided "as is", and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied, with regard to this documentation and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license. Portions of this product include the SystemC software licensed under Open Source terms, which are available for download at http://systemc.org/ . This software is redistributed by Agilent. The Contributors of the SystemC software provide this software "as is" and offer no warranty of any kind, express or implied, including without limitation warranties or conditions or title and non-infringement, and implied warranties or conditions merchantability and fitness for a particular purpose. Contributors shall not be liable for any damages of any kind including without limitation direct, indirect, special, incidental and consequential damages, such as lost profits. Any provisions that differ from this disclaimer are offered by Agilent only.

Restricted Rights Legend U.S. Government Restricted Rights. Software and technical data rights granted to the federal government include only those rights customarily provided to end user customers. Agilent provides this customary commercial license in Software and technical data pursuant to FAR 12.211 (Technical Data) and 12.212 (Computer Software) and, for the Department of Defense, DFARS 252.227-7015 (Technical Data - Commercial Items) and DFARS 227.7202-3 (Rights in Commercial Computer Software or Computer Software Documentation).

Introduction to Simulator Expressions . . . 9
Simulator Expressions Syntax . . . 11
Functions . . . 20
Using Simulator Expressions in Advanced Design System . . . 23
Using a VAR (Variables and Equations Component) . . . 23
Editing Component Parameters . . . 25
Setting up Frequency-Domain Defined Devices (FDD) . . . 25
Setting up a Symbolically Defined Devices (SDD) . . . 25
Alphabetical Listing of Simulator Functions . . . 26
Data Access Functions For Simulator Expressions . . . 29
access_data() . . . 29
dsexpr() . . . 30
dstoarray() . . . 31
get_array_size() . . . 31
index() . . . 32
length() . . . 33
list() . . . 33
makearray() . . . 35
scalearray() . . . 35
Harmonic Balance Functions for Simulator Expressions . . . 37
get_fund_freq() . . . 37
HSPICE Compatibility Functions . . . 38
_db_hspice() . . . 38
_log10_hspice() . . . 38
_log_hspice() . . . 39
_nint_hspice() . . . 39
_pow_hspice() . . . 40
_pwr_hspice() . . . 40
_sign_hspice() . . . 41
_sqrt_hspice() . . . 41
Math Functions for Simulator Expressions . . . 43
abs() . . . 44
acos() . . . 45
acosh() . . . 45
arcsinh() . . . 46
arctan() . . . 46
asin() . . . 47
asinh() . . . 47
atan2() . . . 48
atan() . . . 49
atanh() . . . 49
bin() . . . 50
ceil() . . . 50
complex() . . . 51
conj() . . . 51
cos() . . . 52
cosh() . . . 53
cot() . . . 53
coth() . . . 54
ctof() . . . 54

ctok() . . . 55
db() . . . 55
dbm() . . . 56
dbmtoa() . . . 57
dbmtov() . . . 58
dbmtow() . . . 59
dbpolar() . . . 59
dbwtow() . . . 60
deg() . . . 61
dphase() . . . 61
eval_pole_zero() . . . 62
eval_controlled_pwl() . . . 63
eval_miso_poly() . . . 65
eval_poly() . . . 69
exp() . . . 71
floor() . . . 73
fmod() . . . 73
ftoc() . . . 74
ftok() . . . 74
hypot() . . . 75
imag() . . . 75
int() . . . 76
itob() . . . 76
jn() . . . 77
ktoc() . . . 77
ktof() . . . 78
ln() . . . 78
log10() . . . 79
log() . . . 79
mag() . . . 80
max() . . . 80
min() . . . 81
phase() . . . 82
phasedeg() . . . 82
phaserad() . . . 83
phasewrap() . . . 83
polar() . . . 84
polarcpx() . . . 85
pow() . . . 85
rad() . . . 86
real() . . . 86
rem() . . . 87
sgn() . . . 87
sin() . . . 88
sinc() . . . 89
sinh() . . . 89
spectrum() . . . 90
sqrt() . . . 90
sum() . . . 91
tan() . . . 91

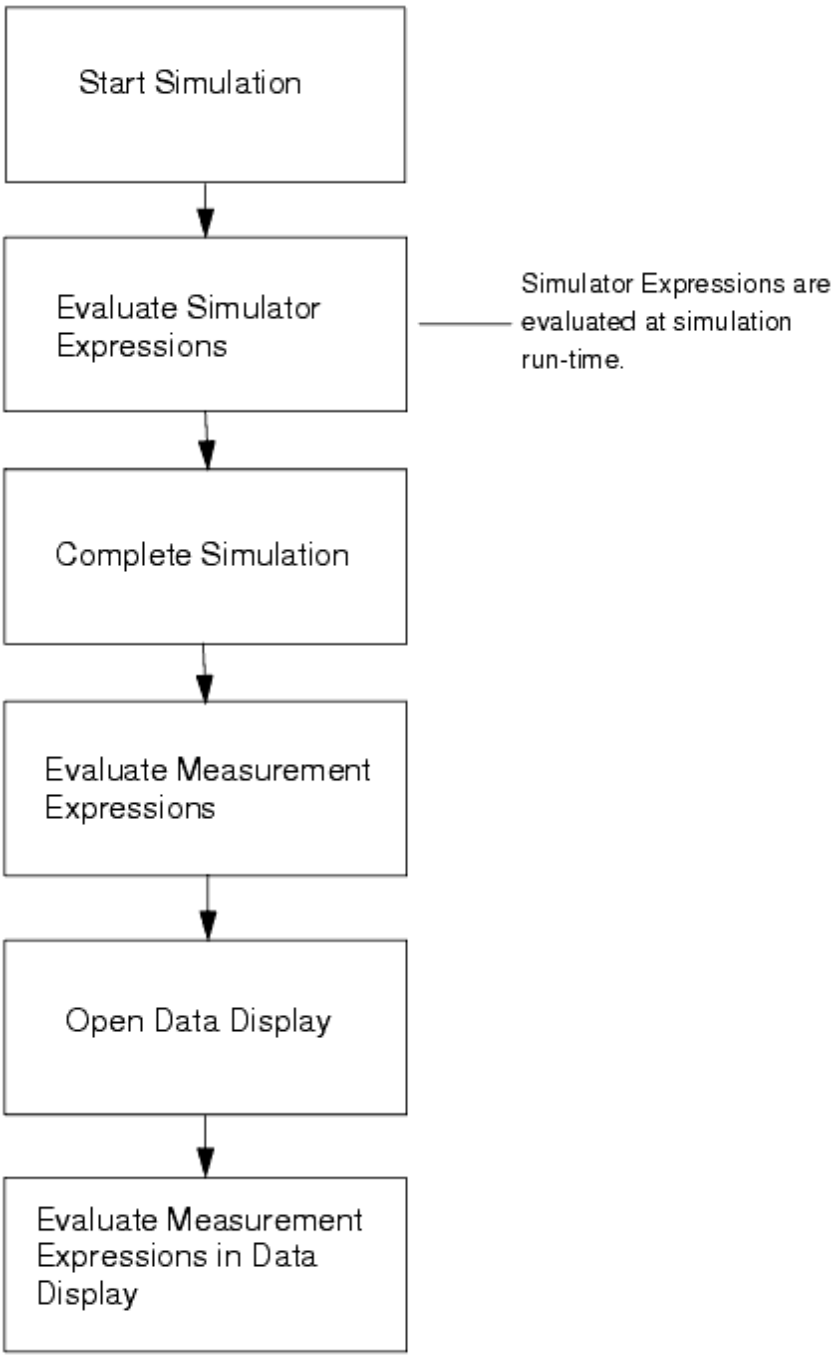
tanh() . . . 92
wtodbm() . . . 93
S-Parameter Analysis Functions for Simulator Expressions . . . 94
ripple() . . . 94
vswrpolar() . . . 94
Transient Source Functions . . . 96
bitseq() . . . 96
cos_pulse() . . . 97
damped_sin() . . . 99
erf_pulse() . . . 101
exp_pulse() . . . 103
impulse() . . . 105
lfsr() . . . 106
pulse() . . . 106
pwl() . . . 108
pwlr() . . . 110
ramp() . . . 111
rect() . . . 112
sffm() . . . 114
step() . . . 115
Utility Functions for Simulator Expressions . . . 118
echo() . . . 118
gridSmithChart() . . . 118
limit_warn() . . . 119
sprintf() . . . 120
strcat() . . . 121
value() . . . 122

Introduction to Simulator Expressions

This document describes the simulator expressions that are available for use with several Agilent EEsof EDA products. For a complete list of available simulator functions, refer to the *Alphabetical Listing of Simulator Functions*.

Simulator expressions are functions that you define before they are used internally during simulation run-time. They can be entered into the program using various methods, depending on which product you are using. Unlike the expressions described in the *Measurement Expressions* documentation, simulator expressions are evaluated at the beginning of a simulation, not after the simulation has completed.
Although there is some overlap among many of the more commonly used functions, *simulator expressions* and *measurement expressions* are derived from separate sources, evaluated at different times, and can have subtle differences in their usages. Thus, these two types of expressions need to be considered separately. For an overview of how simulator expressions are evaluated, refer to How Simulator Expressions are Evaluated.

Note that if a particular term is used in a simulator expression, the term must be defined before the simulation is run. For example, if you use the variable *R* in a simulator expression, and R = S(1,1), where the results of S(1,1) will not be known until after the simulation is complete, an error will be returned by the simulator.



How Simulator Expressions Are Evaluated

Within this document you will find:

- Information on simulator expressions syntax.
- A functions reference table that provides a complete list of all available simulator functions. Individual functions are also listed in the index for your convenience.
- Information specific to entering simulator expressions in your product.

You will also find a complete list of functions that can be used as simulator expressions

individually, or combined together as a nested expression. These expressions have been separated into libraries and are listed in alphabetical order within each library. The expressions available include:

- *Data Access Functions* (expsim)
- *Harmonic Balance Functions* (expsim)
- *Math Functions* (expsim)
- *S-Parameter Analysis Functions* (expsim)
- *Transient Source Functions* (expsim)
- *Utility Functions* (expsim)

Note
All predefined expressions, functions, constants, and variables listed in this section are reserved names. You can use them in your expressions; however, you cannot redefine them to something else.

Simulator Expressions Syntax

Use the following guidelines when creating simulator expressions:

- Simulator expressions are based on the mathematical syntax in Application Extension Language (AEL).
- Function names, variable names, and constant names are all case sensitive in simulator expressions.
- Use commas to separate arguments.
- White space between arguments is acceptable.

The general form of an expression is:
expressionName = nonconstantExpression

For example:
x1 = 4.3 + freq;
syc_a = cos(1.0+sin(pi*3 + 2.0*x1))
Zin = 7.8 ohm + j*freq * 1.9 ph
y = if (x equals 0) then 1.0e100 else 1/x endif

Case Sensitivity

All variable names, functions names, and equation names are case sensitive in simulator expressions.

Predefined Expressions

The following expressions are predefined:

gaussian	=	_gaussian_tol(10.0)	default gaussian distribution
nfmin	=	_nfmin()	the minimum noise figure
omega	=	2.0*pi*freq	the analysis frequency
rn	=	_rn()	the noise resistance
sopt	=	_sopt	the optimum noise match
tempkelvin	=	temp + 273.15	the analysis temperature
uniform	=	_uniform_tol(10.0)	default uniform distribution

Constants and Variables

Many predefined constants and predefined global variables are available. Predefined Simulator Variables and Constants Reserved Names lists the simulator constants and variables and provides a brief description of each.

Constants

An integer constant is represented by a sequence of digits optionally preceded by a negative sign (e.g. `14` , `-3`).

A real number contains a decimal point and/or an exponential suffix using the `e` notation (e.g. `14.0` , `-13e-10`).

The only complex constant is the predefined constant `j` , which is equal to the square root of -1. It can be used to generate complex constants from real and integer constants (e.g., `j*3` , `9.1 + j*1.2e-2`). The predefined functions *complex()* (expmeas) and *polar()* (expsim) can also be used to enter complex constants into an expression.

A string constant is delimited by single or double quotes (e.g. `'string'` , `"this is a string"`). Always use vertical quotes (e.g. ' *<string>* ') as opposed to open quotes (e.g. ` *<string>'*).

Variables

Variables can be modified and swept. The main difference between expressions and variables is that a variable can be directly swept and modified by an analysis but an expression cannot. Note however, that any instance parameter that depends on an expression is updated whenever one of the variables that the expression depends upon is

changed (e.g. by a sweep).

The general form of variables is:
variableName = constantExpression

Note
Once a variable has been defined, it cannot be redefined with another *variableName = constantExpression* statement. This will create an error.

For example:
x1 = 4.3inches + 3mils
syc_a = cos(1.0+sin(pi*3))
Zin = 7.8k - j*3.2k

The type of a variable is determined by the type of its value. For example, `x=1` is an integer, `x=1+j` is complex, and `x = "Tuesday"` is a string.

Predefined Constants and Variables Reserved Names

When you are using simulator expressions, keep in mind that predefined constants and variables are *reserved words*. You can use these constants and variables, but you cannot redefine them to something else. Predefined Simulator Variables and Constants Reserved Names lists the simulator constants and variables available and provides a brief description of each.

Predefined Simulator Variables and Constants Reserved Names

Variable/Constant Name	Description
__fdd	Flag to indicate a new FDD instance
__fdd_v	Flag to indicate updated FDD state vars
_ABM_Phase †††	Phase for ABM cosim modeling (internal use)
_ABM_SourceLevel †††	Linear amplitude scaling for ABM cosim modeling (internal use)
_ac_state	Is analyses in ac state
_c1 to _c30	Symbolic controlling current
_dc_state	Is analyses in dc state
_default	Used to set parameter to inbuilt default value.
_freq1 to _freq12	Fundamental frequency
_harm	Harmonic number index for sources and FDD
_hb_state	Is analyses in harmonic balance state
_i1 to _i19 †	State variable currents used by the sdd device
_M	Multiplicity factor.
_p2dInputPower	Port input power for P2D simulation

_sigproc_state	Is analyses in signal processing state
_sm_state	Is analyses in sm state
_sp_state	Is analyses in sparameter analysis state
_tr_state	Is analyses in transient state
_v1 to _v19 †	State variable voltages used by the sdd device
time = 0 s	the analysis time
timestep = 1 s	the analysis time step
tranorder = 1	the transient analysis integration order
freq = 1e+006 Hz	the analysis frequency
noisefreq = 1e+006 Hz	the spectral noise analysis frequency
ssfreq = 1e+006 Hz	the small-signal mixer analysis frequency
temp = 25 C	the analysis temperature
tnom = 25 C	default nominal temperature for models
e = 2.71828	2.71828...
j	Square root of -1
ln10 = 2.30259	ln(10)
pi	3.14...
c0 = 2.99792e+008 m/s	the speed of light
e0 = 8.85419e-012	vacuum permittivity
u0 = 1.25664e-006	vacuum permeability
boltzmann = 1.38066e-023	Boltzmann's constant
qelectron = 1.60218e-019	the charge of an electron
planck = 6.62608e-034	Planck's constant
hugereal = 1.79769e+308	largest real number
tinyreal = 2.22507e-308	smallest real number
sourceLevel = 1 ††	used for source-level sweeping
dcSourceLevel = 1	used for DC source-level sweeping
logRshunt = 0	used for DC Rshunt sweeping
logNodesetScale = 0	used for DC nodeset simulation
logRforce = 0	used for HB Rforce sweeping
mcindex = 0	index for Monte Carlo sweeps
doeindex = 0	index for Design of Experiment sweeps
CostIndex = 0	index for optimization cost plots
mcTrial = 0	trial counter for Monte Carlo based simulations
optIter = 0	optimization job iteration counter
doeIter = 0	doe experiment iteration counter
DeviceIndex = 0	device Index used for noise contribution or DC OP output
LinearizedElementIndex = 0	index for BudLinearization sweep
DF_Value = -1e+009	reference to corresponding value defined in Data Flow controller

DF_ZERO_OHMS = 1e-013	symbol for use as zero ohms
DF_DefaultInt = -1e+009	reference to default int value defined in Data Flow controller

† The _i and _v variables should only be used in the context of the SDD device.

†† The sourcelevel variable is used by the spectral analysis when it needs to gradually increase source power from 0 to full scale to obtain convergence. It can be used by the user to sweep the level of ALL spectral source components, but is not recommended.

††† The _ABM_Phase and _ABM_SourceLevel variables are only visible in an AVM/FastCosim dataset; however, they are not supported for regular use. These are used as the independent swept variables for the nonlinear characterization data.

Units and Scale Factors

The fundamental units for the simulator are shown in Fundamental Units in the Simulator. A parameter with a given dimension assumes its value has the corresponding units. For example, for a resistance, `R=10` its assumed to be 10 Ohms.

Fundamental Units in the Simulator

Dimension	Fundamental Unit
Frequency	Hertz
Resistance	Ohms
Conductance	Siemens
Capacitance	Farads
Inductance	Henries
Length	meters
Time	seconds
Voltage	Volts
Current	Amperes
Power	Watts
Distance	meters
Temperature	Celsius

Recognizing Scale Factors

Variations on the fundamental units in the simulator are referred to as *scale factors*. A scale factor is a single word that begins with a letter or an underscore character (_). The remaining characters, if any, consist of letters, digits, and underscores. The value of a scale factor is resolved using the following rules in the order shown:

1. If the scale factor exactly matches one of the predefined *scale-factor words* (Predefined Scale Factor Words), then use the numerical equivalent; otherwise, go to rule 2.

Predefined Scale Factor Words

Scale Factor Word	Numerical Equivalent	Meaning
mil	$2.54*10^{-5}$	mils
mils	$2.54*10^{-5}$	mils
in	$2.54*10^{-2}$	inches
ft	$12*2.54*10^{-2}$	feet
mi	$5280*12*2.54*10^{-2}$	miles
cm	$1.0*10^{-2}$	centimeters
PHz	$1.0*10^{15}$	petahertz
dB	1.0	decibels
nmi	1852	nautical miles

2. If the scale factor exactly matches one of the *scale-factor units* (Scale Factor Units) except for *m* , then use the numerical equivalent; otherwise, go to rule 3.

Scale Factor Units

Scale Factor Unit	Numerical Equivalent	Meaning
A	1.0	Amperes
F	1.0	Farads
H	1.0	Henries
Hz	1.0	Hertz
metermetersmetremetres	1.0	meters
OhmOhms	1.0	Ohms
S	1.0	Siemens
sec	1.0	seconds
V	1.0	Volts
W	1.0	Watts

3. If the first character of the scale factor is one of the legal *scale-factor prefixes* (Scale Factor Prefixes), then use the numerical equivalent; otherwise, go to rule 4.

Scale Factor Prefixes

Prefix	Numerical Equivalent	Meaning
T	10^{12}	Tera
G	10^{9}	Giga
M	10^{6}	Mega
K	10^{3}	kilo
k	10^{3}	kilo
_ (underscore)	1	(no scale)
m	10^{-3}	milli
u	10^{-6}	micro
n	10^{-9}	nano
p	10^{-12}	pico
f	10^{-15}	femto
a	10^{-18}	atto

4. The scale factor is not recognized.
   Important considerations include:
   - Scale factors are case sensitive.
   - A single `m` means `milli`, not `meters`.
   - A lower case `f` by itself means `femto`. An upper case `F` by itself means `Farad`.
   - A lower case `a` by itself means `atto`. An upper case `A` by itself means `Ampere`.
   - The imperial units (`mils` , `in` , `ft` , `mi` , `nmi`) do not accept prefixes.
   - The simulator will report a warning if an unrecognized scale factor is encountered, and use a scale-factor value of 1.0.
   - It is not required that the characters following a scale-factor prefix match one of the scale-factor units.
   - There are no scale factors for `dBm` , `dBW` , or temperature. Simulator functions are provided to convert these values to the corresponding fundamental units (Watts and Celsius).

Mathematical Operators and Hierarchy

Simulator expressions are evaluated from left to right, unless there are parentheses. Operators are listed from higher to lower precedence in Arithmetic and Boolean Operator Precedence. Operators on the same line have the same precedence. For example, a+bc means a+(bc), because multiplication has a higher precedence than addition. Similarly, a+b-c means (a+b)-c, because addition and subtraction have the same precedence (and because + is left-associative).

The operators !, &&, and || work with the logical values. The operands are tested for the values TRUE and FALSE, and the result of the operation is either TRUE or FALSE. A logical test of a value is TRUE for non-zero numbers or strings with non-zero length, and FALSE for 0.0 (real), 0 (integer), NULL or empty strings. Note that the right hand operand of && is only evaluated if the left hand operand tests TRUE, and the right hand operand of || is only evaluated if the left hand operand tests FALSE.

The Boolean operators >=, <=, >, <, ==, != , and, or, equals, and not equals also produce logical results, producing a logical TRUE or FALSE upon comparing the values of two expressions. These operators are most often used to compare two real numbers or integers. These operators operate differently than C with string expressions in that they actually perform the equivalent of strcmp() between the first and second operands, and test the return value against 0 using the specified operator.

Arithmetic and Boolean Operator Precedence

Operator	Name	Example
( )	function call	foo(expr_list)
[ ]	indexer, array	X[expr_list]
**	exponentiation	expr**expr [1]
*	multiply	expr * expr
+	add	expr + expr
::	sequence operator	exp::expr::expr
<	less than	expr < expr
==	equal	expr == expr
&&	logical and	expr && expr
||	logical or	expr || expr

[1] To avoid returning incorrect results when using ** with very large integers, convert integers to real numbers first.

Conditional Expressions

The simulator supports simple in-line conditional expressions:

`A = if` *boolExpr* `then` *expr* `else` *expr* `endif`

`A = if` *boolExpr* `then` *expr* `elseif` *boolExpr* `then` *expr* `else` *expr* `endif`

boolExpr is a valid Boolean expression, that is, an expression that evaluates to TRUE or FALSE.

expr is any valid non-Boolean expression.

The *else* is required because the conditional expression must always evaluate to some value.

There can be any number of occurrences of `elseif` *expr* `then` *expr*.

A conditional expression can legally occur as the right-hand side of an expression or function definition or, if parenthesized, anywhere in an expression that a variable can occur. The dimensionality and number of points in these expressions follow the same matching conditions required for the basic operators. The type of the result depends on the type of the true and false expressions. The size of the result depends on the size of the condition, the true expression, and the false expression.

Boolean expressions

A Boolean expression must evaluate to TRUE or FALSE and, therefore, must contain a relational operator (`equals`, `=`, `==`, `notequals`, `!=`, `<`, `>`, `<=`, or `>=`).
The only legal place for a Boolean expression is directly after an `if` or an `elseif`.
A Boolean expression cannot stand alone, that is,
x = a > b
is illegal.

Precedence

Tightest binding: `equals`, `=`, `==`, `notequals`, `!=`, `>`, `<`, `>=`, `<=`
`not`, `!`
`and`
Loosest binding: `or`, `||`
All arithmetic operators have tighter binding than the Boolean operators.

Evaluation

Boolean expressions are short-circuit evaluated. For example, if when evaluating *a* `and` *b* , expression *a* evaluates to FALSE, expression *b* will not be evaluated.
During evaluation of Boolean expressions with arithmetic operands, the operand with the lower type is promoted to the type of the other operand. For example, in `3 equals x +j*b, 3` is promoted to complex.
A complex number cannot be used with `<` , `>` , `<=` , or `>=`. Nor can an array (and remember that strings are arrays). This will cause an evaluation-time error.
Pointers can be compared only with pointers.

Examples

Protect against divide by zero:

```
f(a) = if a equals 0 then 1.0e100 else 1.0/a endif
```

Nested if's #1:

```
f(mode) = if mode equals 0 then 1-a else f2(mode) endif
f2(mode) = if mode equals 1 then log(1-a) else f3(mode) endif
f3(mode) = if mode equals 2 then exp(1-a) else 0.0 endif
```

Nested if's #2:

```
f(mode) = if mode equals 0 then 1-a elseif mode equals 1 then log(1-a)
elseif mode equals 2 then exp(1-a) else 0.0 endif
```

Soft exponential:

```
exp_max = 1.0e16
x_max = ln(exp_max)
```

```
exp_soft(x) = if x<x_max then exp(x) else (x+1-x_max)*exp_max endif
```

Functions

You can define your own functions in simulator expressions. These functions can then be used in other expressions.

The general form of a function is:

functionName (*[arg1, ..., argn]*) = *expression*

For example:

```
y_srl(freq, r, l) = 1.0/(r + j*freq*l)
expl(a,b) = exp(a)*step(b-a) + exp(b)*(a-b-1)*step(a-b)
```

In *expression*, the function's arguments can be used, as can any other ADS Simulator variables, expressions, or functions. For a complete list of available simulator functions, refer to the *Alphabetical Listing of Simulator Functions* or consult the index.

Note
The trigonometric functions always expect the argument to be specified in radians. If you want to specify the angle in degrees, then use the function *deg()* (expsim) to convert radians to degrees, or you can use the function *rad()* (expsim) to convert degrees to radians.

Another example of defining and using a function is:

```
B(x) = makearray(1,x*1.0,x*2.1,x*3.0)
B_2 = B(2)[2]
```

which returns 4.2

You can also define custom functions to be used as simulator expressions defined in a VAR item. Consider the following example:

```
my_fcn(X) = 2*X
```

This function can be evaluated in two different ways:

- Explicit definition of X in a VAR item, where:

  ```
  X = 5
  R = my_fcn(X)
  ```

  R is evaluated as 2*5=10 ohms
- Definition of X using a component parameter value, where:

  ```
  R = my_fcn(25)
  ```

R is evaluated as 2*25=50 ohms

Predefined Functions Reserved Names

Predefined Function Reserved Names lists predefined functions which are reserved names. Many functions in the list are used only internally, or are obsolete, so they are not described in the documentation. However, the names are still reserved.

Predefined Function Reserved Names

abs access_all_data† access_data acos acosh amp_harm_coef† arcsinh arctan asin asinh atan atan2 atanh awg_dia†	bin bitseq
ceil coef_count† complex compute_poly_coef† conj cos cos_pulse cosh cot coth cpx_gain_poly† ctof ctok cxform†	damped_sin db dbm dbmtoa dbmtov dbmtow dbpolar dbwtow deembed† deg delay† dep_data† deriv† dphase dsexpr dstoarray† d_atan2†
echo embedded_ptolemy_exec† erf_pulse eval_controlled_pwl eval_miso_poly eval_poly exp exp_pulse	floor fmod fread† freq_mult_coef† freq_mult_poly† ftoc ftok
gcdata_to_poly† generate_gmsk_iq_spectra† generate_gmsk_pulse_spectra† generate_piqpsk_spectra† generate_pulse_train_spectra† generate_qam16_spectra† generate_qpsk_pulse_spectra† get_array_size get_attribute† get_block† get_fund_freq get_max_points†	hypot
i† ilsb† imag impulse imt_hbdata_to_array† imt_hpvar_to_array† index innerprod† inoise† int internal_generate_gmsk_iq_spectra† internal_generate_gmsk_pulse_spectra† internal_generate_piqpsk_spectra† internal_generate_pulse_train_spectra† internal_generate_qam16_spectra† internal_generate_qpsk_pulse_spectra† internal_get_fund_freq† internal_window† interp† interp1† interp2† interp3† interp4† iss† itob iusb†	jn
ktoc ktof	length lfsr limit_warn list ln log log10 log_amp† log_amp_cas† lookup†
mag makearray max min miximt_coef† miximt_poly† multi_freq†	names† nf† norm†
phase phasedeg phaserad phasewrap phase_noise_pwl† polar polarcpx pow pulse pwl pwlr pwlr_tr†	qinterp†
rad ramp rawtoarray† readdata† readlib† readraw† read_data† read_lib† real rect rem repeat† ripple rms† rpsmooth†	scalearray sens† setDT† sffm sgn sin sinc sinh spectrum sprintf sqrt step strcat stypexform† sym_set† system†
tan tanh thd† toi† transform†	v† value vlsb† vnoise† vss† vswrpolar vusb†
window† wtodbm	_discrete_density† _divn† _gaussian† _gaussian_tol† _get_fnom_freq† _get_fund_freq_for_fdd† _lfsr† _mvgaussian† _mvgaussian_cov† _nfmin† _n_state† _phase_freq† _pwl_density† _pwl_distribution† _randvar† _rn† _shift_reg† _si†† _si_bb† _si_d†† _si_e† _sopt† _sv†† _sv_bb† _sv_d†† _sv_e† _tn† _to† _tt† _uniform† _uniform_tol† _xcross†
† These simulator functions are for Agilent internal use only and are not supported for regular use. †† These simulator functions are used for frequency-domain defined devices (FDDs). For more information, refer to "Retrieving Values from Port Variables" in the "User-Defined Models" documentation.	

Using Simulator Expressions in Advanced Design System

Simulator expressions can be entered into Advanced Design System using the following methods:

- Using a VAR (Variables and Equations Component)
- Editing Component Parameters
- Setting up a Frequency-domain Defined Devices (FDD)
- Setting up a Symbolically Defined Devices (SDD)

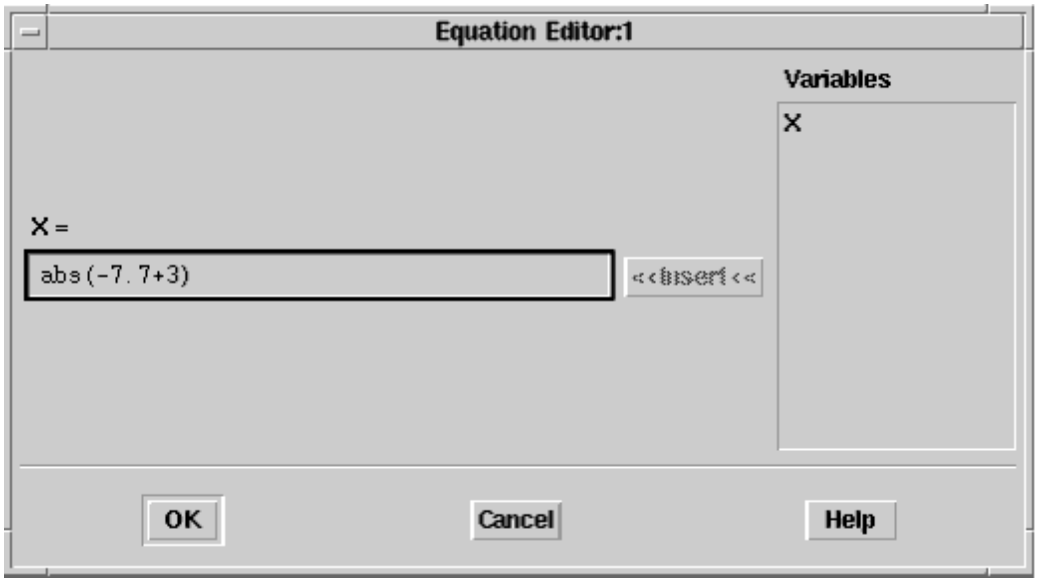
Using a VAR (Variables and Equations Component)

Simulator expressions are sometimes referred to as VARs in Advanced Design System. The VAR (Variables and Equations Component) is available in the Data Items palette in an Analog/RF Systems Schematic window, or from the Controllers palette in a Signal Processing Schematic window. For more information, refer to Variables and Equations Component in the *Introduction to Circuit Components* (ccsim) documentation.

Caution
Do not use more than 1800 characters in a VAR equation. If longer VARs are required, consider dividing up the long equation into multiple shorter equations and then summing the right hand side of the shorter equations. You may also want to consider using a netlist fragment instead of a VAR.

To add a simulator expression in an Analog/RF Systems Schematic window using a VAR:

1. Click the VAR in the Data Items palette.
2. Place a VAR on the schematic.
3. Double-click the VAR to display the dialog box.
4. Click the **Equation Editor** button. The Equation Editor dialog box is displayed.



5. Enter your expression in the field provided and click **OK** .
   For more information on using the Equation Editor, click the **Help** button in the lower right hand corner of the dialog box. This will refer you to "Using the Equation Editor" in the *Schematic Capture and Layout (usrguide)* documentation.

VarEqn Data Types

The four basic data types that *VarEqn* supports are integer, real, complex, and string. There is a fifth data type, pointer, that is also supported. Pointers are not allowed in an algebraic expression, except as an argument to a function that is expecting a pointer. Strings are not allowed in algebraic expressions either except that addition of strings is equivalent to catenation of the strings. String catenation is not commutative, and since *VarEqn's* simplification routines can internally change the order of operands of commutative operators, this feature should be used cautiously. It will most likely be replaced by an explicit catenation function.

Type conversion

The data type of a *VarEqn* expression is determined at the time the expression is evaluated and depends on the data types of the terms in the expression. For example, let y=3*x^2. If x is an integer, then y is integer-valued. If x is real, then y is real-valued. If x is complex, then y is complex-valued.

As another example, let y=sqrt(2.5*x). If x is a positive integer, then y evaluates to a real number. If, however, x is a negative integer, then y evaluates to a complex number.

There are some special cases of type conversion:

- If either operand of a division is integer-valued, it is promoted to a real before the division occurs. Thus, 2/3 evaluates to 0.6666....

- The built-in trigonometric, hyperbolic, and logarithmic functions never return an integer, only a real or complex number.

Editing Component Parameters

Simulator expressions can be entered in place of most component parameters in Advanced Design System. The components are available from the component palette in an Analog/RF Systems Schematic window, or from the Controllers palette in a Signal Processing Schematic window. For more information on ADS components, refer to the *Components* category of your online documentation set.
To add a simulator expression to a component parameter:

1. In a schematic window, select the component from the component palette and place it on the schematic.
2. Double-click the component to display the dialog box.
3. If available, set the Parameter Entry Mode to **Standard** in the component dialog box.

   Note
   The Standard Parameter Entry Mode is not available for all component parameters. If the Standard Parameter Entry Mode is not available, the Equation Editor is not available for that particular parameter.
4. Click the **Equation Editor** button. The Equation Editor dialog box appears with the component parameter displayed on the left.
5. Enter your expression in the field provided and click **OK** .
   For more information on using the Equation Editor, click the **Help** button in the lower right hand corner of the dialog box. This will refer you to *Using the Equation Editor* in the *Schematic Capture and Layout* (usrguide) documentation.

Setting up Frequency-Domain Defined Devices (FDD)

The frequency-domain defined device (FDD) enables you to create equation-based, user-defined, nonlinear components. The FDD is a multi-port device that describes current and voltage spectral values in terms of algebraic relationships of other voltage and current spectral values. It is for developing nonlinear, behavioral models that are more easily defined in the frequency domain.

For more information on setting up an FDD, refer to *Nonlinear Devices* (ccnld).

Setting up a Symbolically Defined Devices (SDD)

The symbolically-defined device (SDD) enables you to create equation based, user-defined, nonlinear components. The SDD is a multi-port device which is defined by specifying algebraic relationships that relate the port voltages, currents, and their derivatives, plus currents from certain other devices.

For more information on setting up an SDD, refer to *Nonlinear Devices* (ccnld).

Alphabetical Listing of Simulator Functions

Consult the Index for an alternate method of accessing simulator functions.

For information on measurement functions, refer to the *Measurement Expressions* (expmeas) documentation.

_	
_db_hspice() (expsim)	*_pow_hspice()* (expsim)
_log10_hspice() (expsim)	*_pwr_hspice()* (expsim)
_log_hspice() (expsim)	*_sign_hspice()* (expsim)
_nint_hspice() (expsim)	*_sqrt_hspice()* (expsim)
A	
abs() (expsim)	*asin()* (expsim)
access_data() (expsim)	*asinh()* (expsim)
acos() (expsim)	*atan()* (expsim)
acosh() (expsim)	*atan2()* (expsim)
arcsinh() (expsim)	*atanh()* (expsim)
arctan() (expsim)	
B, C	
bin() (expsim)	*cos_pulse()* (expsim)
bitseq() (expsim)	*cosh()* (expsim)
ceil() (expsim)	*cot()* (expsim)
complex() (expsim)	*coth()* (expsim)
conj() (expsim)	*ctof()* (expsim)
cos() (expsim)	*ctok()* (expsim)
D	
damped_sin() (expsim)	*dbpolar()* (expsim)
db() (expsim)	*dbwtow()* (expsim)
dbm() (expsim)	*deg()* (expsim)
dbmtoa() (expsim)	*dphase()* (expsim)
dbmtov() (expsim)	*dsexpr()* (expsim)
dbmtow() (expsim)	
E	
echo() (expsim)	*eval_poly()* (expsim)

erf_pulse() (expsim)	*exp()* (expsim)
eval_controlled_pwl() (expsim)	*exp_pulse()* (expsim)
eval_miso_poly() (expsim)	
F	
floor() (expsim)	*ftoc()* (expsim)
fmod() (expsim)	*ftok()* (expsim)
G	
get_array_size() (expsim)	*get_fund_freq()* (expsim)
gridSmithChart() (expsim)	
H, I	
hypot() (expsim)	*index()* (expsim)
imag() (expsim)	*int()* (expsim)
impulse() (expsim)	*itob()* (expsim)
J, K, L	
jn() (expsim)	*limit_warn()* (expsim)
ktoc() (expsim)	*list()* (expsim)
ktof() (expsim)	*ln()* (expsim)
length() (expsim)	*log()* (expsim)
lfsr() (expsim)	*log10()* (expsim)
M, N	
mag() (expsim)	*max()* (expmeas)
makearray() (expsim)	*min()* (expmeas)
P, Q	
phase() (expsim)	*polarcpx()* (expsim)
phasedeg() (expsim)	*pow()* (expsim)
phaserad() (expsim)	*pulse()* (expsim)
phasewrap() (expsim)	*pwl()* (expsim)
polar() (expsim)	*pwlr()* (expsim)
R	
rad() (expsim)	*rect()* (expsim)
ramp() (expsim)	*rem()* (expsim)
real() (expsim)	*ripple()* (expsim)
S	
scalearray() (expsim)	*spectrum()* (expsim)
sffm() (expsim)	*sprintf()* (expsim)
sgn() (expsim)	*sqrt()* (expsim)

sin() (expsim)	*step()* (expsim)
sinc() (expsim)	*strcat()* (expsim)
sinh() (expsim)	
T, V, W	
tan() (expsim)	*vswrpolar()* (expsim)
tanh() (expsim)	*wtodbm()* (expsim)
value() (expsim)	

Data Access Functions For Simulator Expressions

This section describes the data access functions in detail. Data Access Functions are used to manipulate data types, such as arrays, lists, datasets, etc. The functions are listed in alphabetical order.

- *access data()* (expsim)
- *dsexpr()* (expsim)
- *dstoarray* (expsim)
- *get array size()* (expsim)
- *index() Expression* (expsim)
- *length()* (expsim)
- *list() Expression* (expsim)
- *makearray()* (expsim)
- *scalearray()* (expsim)

access_data()

Datafile dependents lookup or interpolation function

Syntax

access_data(AccessType, DataSetVar, Ivar1, IValue1, Ivar2, Ivalue2, ...)

Arguments

Name	Description	Default	Range	Type	Required
AccessType	Type of access to data	None	[0-6] †	Integer, String	Yes
DataSetVar	Dataset variable that points to dataset that has been accessed	None	None	Dataset Variable	Yes
Ivar1, Ivar2, ...	Independent variable to be accessed	None	None	String, Integer	No
Ivalue1, Ivalue2, ...	Independent variable value to be accessed	None	None	Integer, Real, Complex, String	No

† alternately use [linear,spline,cubic,index_lookup,value_lookup,ceil_value_lookup,floor_value_lookup]

Examples

```
In this example we assume that the dataset spar.ds contains S-parameters at 2
frequencies 1 and 2 GHz.
dsS11=dsexpr("S(1,1)", "spar.ds") returns the S(1,1) dataset variable
adV0=access_data("linear", dsS11, "freq", 1.5 GHz) returns linearly
interpolated S11 at 1.5 GHz
adV3=access_data(3, dsS11, "freq", 1) returns S11 at 2 GHz
adV4=access_data(4, dsS11, "freq", 1 GHz) returns S11 at 1 GHz
```

```
adV5=access_data(5, dsS11, "freq", 1.9 GHz) returns S11 at 2 GHz
adV6=access_data(6, dsS11, "freq", 1.1 GHz) returns S11 at 1 GHz
```

See Also

dsexpr() (expsim)

Notes/Equations

The access_data() function is used to access the dependent data of a specific type from a dataset. The accessed data can then be used in the design as a parameter value or in other simulation expressions. The data can be accessed either by interpolation or lookup. The access type choices are:

Access Type	Description
0 or linear	access data by linear interpolation
1 or spline	access data by spline interpolation
2 or cubic	access data by cubic interpolation
3 or index_lookup	access data by index value truncation, average of end indices if midway
4 or value_lookup	access data by value - nearest value, average of end points if midway
5 or ceil_value_lookup	access data by nearest value not less than given value, except maximum of value for value greater than maximum value
6 or floor_value_lookup	access data by nearest value not greater than given value, except minimum of value for value less than minimum value

dsexpr()

Evaluate a dataset expression to a dataset variable

Syntax

dsexpr(Expression, DataSet, UseCache)

Arguments

Name	Description	Default	Range	Type	Required
Expression	Expression to be evaluated	None	None	String	Yes
DataSet	Full-path to the dataset file	None	None	String	Yes
UseCache	Use Dataset that has already been cached	False	[True:False]	Boolean	No

Examples

```
In this example we assume that the dataset spar.ds contains S-parameters at
frequencies of 1 and 2 GHz.
S11Var=dsexpr("S(1,1)", "spar.ds") returns a dataset variable conatining S(1,1)
S11A=dstoarray(S11Var) returns an array of S(1,1) value
```

See Also

access_data() (expsim)

Notes/Equations

The dsexpr() function executes a simulation expression on the dataset and returns the results as a dataset variable, which can be processed further to access the required data. If the dataset has already been accessed before and has been cached, then the cached value can be accessed by setting the UseCache argument as True.

dstoarray()

Accesses the independent or dependent values of a variable from a dataset variable vector as an array.

Syntax

dstoarray(DataSetVar, "VarName",VarIndex)

Arguments

Name	Description	Default	Range	Type	Required
DataSetVar	Dataset variable that has already been read or accessed	None	None	Dataset Variable	Yes
"VarName",VarIndex	Independent or dependent name or index	1	[1:∞)	String or Integer	No
VarType	Specifies independent or dependent variable to get	0	[0:1] †	Integer	No

† 0 - dependent data, 1 - independent data

Examples

```
In this example we assume that the dataset spar.ds contains S-parameters at 2
frequencies 1 and 2 GHz.
S11Var=dsexpr("S(1,1)", "spar.ds") returns a dataset variable containing S(1,1)
S11A=dstoarray(S11Var) returns an array of S(1,1) value
fData=read_data("dataset", "spar.ds") reads the dataset
dsS11=dstoarray(fData, 1) returns an array of S(1,1) value
fData=read_data("dataset", "spar.ds") reads the dataset
freqV=dstoarray(fData, 1, 1) returns an array of freq values
```

get_array_size()

Get the size of the array

Syntax

get_array_size(Array)

Arguments

Name	Description	Default	Range	Type	Required
Array	array	None	None	Integer, Real, Complex or String Array	Yes

See Also

makearray() (expsim),*list()* (expsim),*length()* (expsim)

index()

Get index of name in an array

Syntax

index(Array, Name, CaseSensitive, Length)

Arguments

Name	Description	Default	Range	Type	Required
Array	Array of strings	NONE	NONE	String Array	Yes
Name	Name or name of a simulator expression variable to find in array	None	None	String	Yes
CaseSensitive	specifies if a case sensitive search is to be done	YES	[NO : YES]	Boolean	No
Length	compares length number of characters	0	[0:∞)	Integer	No

Examples

```
sA = makearray(3, "Zero", "One", "Two") returns an array of three strings
indx = index(sA, "One") returns 2
indx = index(sA, "one", 1) returns -1
indx = index(sA, "On", 1, 2) returns 2

varName = "One"
indx = index(sA, varName)
returns 2
```

See Also

makearray() (expsim), *get_array_size()* (expsim)

Notes/Equations

This function finds the index of a search string or name of a Simulator Expression variable

in the given string array. If a match is found, the index (starts at 1) is returned. Otherwise, a -1 is returned.

Note
The function name index() is used for more than one type of expression. For comparison, see the AEL Function *index() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

length()

Returns number of elements in array

Syntax

length(Array)

Arguments

Name	Description	Default	Range	Type	Required
Array	array	None	None	Integer, Real, Complex or String Array	Yes

Examples

```
rA = makearray(1, 1, 2.0, 3.0) returns a real array of three numbers
rSize = length(rA) returns 3
cA = list(1+j*1, 2.0+j*2, 3.0+j*3) returns a complex array of three numbers
cSize = length(cA) returns 3
```

See Also

get_array_size() (expsim), *list()* (expsim), *scalearray()* (expsim), *makearray()* (expsim)

Notes/Equations

The length() function returns the number of elements in an array that has been created using the makearray() or list() expression.

list()

Creates a list of values

Syntax

list(A0,A1,A2,..)
list("One", "Two", "Three")
list(list(1,2), list(3,4)) returns list(1,3,2,4)

Arguments

Name	Description	Default	Range	Type	Required
A0,A1,A2,...	individual values of the array	None	NA	Integer, Real, Complex, string or array †	Yes

If the arguments are arrays, they must all be the same length.

Examples

```
y = list(1.0, 2.0, 3.0) returns a real list of three numbers
y = list(1+j*1, 2+j*2, 3+j*3) returns a complex list of three complex numbers
```

See Also

get_array_size() (expsim), *length()* (expsim), *makearray()* (expsim), *scalearray()* (expsim)

Notes/Equations

The list() function is nothing but an array in Simulator Expressions. It is similar to the makearray() function. The function can be used to create a list of integer, real, or complex values. With the list() function, the data type does not need to be specified. If any of the array entries are real, the array returned is of type real. If any of the entries are complex, the array is of type complex. There should be a minimum of one entry in the list. Unlike the makearray() function, the list() function cannot be used to create an array of string or text values.

Note
Important Note Regarding Use of makearray() or list() VAR Equations As Direct vs. Indirect Variables In Post-Simulation Data Display:

- You can use makearray() or list() to create a pre-simulation (in design schematic) temporary arbitrary array variable, but you cannot directly access/use the same, intended variable access array in the post-simulation resulting Data Display window.
- Instead, you are free to use the temporary arbitrary data array (created directly by makearray() or list() like "X = makearray(1,.95,.99,1,2)") and then re-cast also in schematic to another indirect variable that references the original temporary variable (Ex: create another variable VAR "Y = X[1]" in design before simulation). Thus, using the X and Y examples here, you will NOT be able to get evaluated direct data "X" variable as intended in data display, but you can get access to the indirect "Y" data.
  - As shown in the following example, you can use an indirect variable like "Y" (Y=X[int_A]) that references the original direct "X" arbirary data array ("X = makearray(1,.95,.99,1,2)") that will be available also in post simulation Data Display:
    - **ADS design VAR Simulator Expression makearray list create arbitrary data array no DataAccessComponent DAC required**

Note
The list() function defined for Simulator Expressions has an index starting at 1; in contrast to the AEL list() function, which has a starting index of 0. This function does not support mixed value types in the same area (e.g. complex, real, integer).

Note
The function name list() is used for more than one type of expression. For comparison, see the AEL Function *list() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

makearray()

Creates an array of real, complex or string values

Syntax

makearray(Type, A1, A2..)
makearray(Array, StartIndex, StopIndex)

Arguments

Name	Description	Default	Range	Type	Required
Type	array type 0:integer, 1:real, 2:complex, or 3:string	None	[1:3]	Integer	Yes
A1,A2, ...	Values in the array	None	(-∞:∞)	Real, Complex or String	Yes
Array	Array from which the sub-array is to be created	None	None	Real, Complex or String Array	Yes
StartIndex	Start index of the array	None	[1:∞)	Integer	Yes
StopIndex	Stop index of the array	None	[1:∞)	Integer	Yes

Note
Important Note Regarding Use of makearray() or list() VAR Equations As Direct vs. Indirect Variables In Post-Simulation Data Display:

- You can use makearray() or list() to create a pre-simulation (in design schematic) temporary arbitrary array variable, but you cannot directly access/use the same, intended variable access array in the post-simulation resulting Data Display window.
- Instead, you are free to use the temporary arbitrary data array (created directly by makearray() or list() like "X = makearray(1,.95,.99,1,2)") and then re-cast also in schematic to another indirect variable that references the original temporary variable (Ex: create another variable VAR "Y = X[1]" in design before simulation). Thus, using the X and Y examples here, you will NOT be able to get evaluated direct data "X" variable as intended in data display, but you can get access to the indirect "Y" data.
  - As shown in the following example, you can use an indirect variable like "Y" (Y=X[int_A]) that references the original direct "X" arbirary data array ("X = makearray(1,.95,.99,1,2)") that will be available also in post simulation Data Display:
    - **ADS design VAR Simulator Expression makearray list create arbitrary data array no DataAccessComponent DAC required**

See Also

get_array_size() (expsim), *list()* (expsim), *length()* (expsim)

scalearray()

Scalar times a vector (array) function

Syntax

scalearray(Scalar, Array)

Arguments

Name	Description	Default	Range	Type	Required
Scalar	Scalar value	None	(-∞:∞)	Integer, Real or Complex	Yes
Array	Array to be scaled	None	None	Integer, Real or Complex Array	Yes

Examples

```
rA=makearray(1, 1,2.0,3.0) returns an array of 3 real numbers
srA=scalearray(10,rA) returns array (10,20,30)
cA=makearray(2,1+j*1, 2+j*2) returns an array of 3 complex numbers
scA=scalearray(10, cA) returns array (10+j*10,20+j*20)
```

See Also

makearray() (expsim), *get_array_size()* (expsim)

Harmonic Balance Functions for Simulator Expressions

This section describes the Harmonic Balance functions in detail.

- *get fund freq()* (expsim)

get_fund_freq()

Get the frequency associated with a specified fundamental index

Syntax

get_fund_freq(fundamental)

Arguments

Name	Description	Default	Range	Type	Required
fundamental	the fundamental index of the frequency	None	[1:12]	Integer, Real	Yes

Examples

```
Assumes that a single-tone harmonic balance analysis is being run at 1GHz
get_fund_freq(1) returns the 1GHz
```

Notes/Equations

The get_fund_freq() function is used during harmonic balance analysis to get the frequency of the specified fundamental.

HSPICE Compatibility Functions

This section describes the HSPICE Compatibility Functions in detail. HSPICE Compatibility Functions are math functions used to with HSPICE netlist simulations. The functions are listed in alphabetical order.

- *_db_hspice()* (expsim)
- *_log10_hspice()* (expsim)
- *_log_hspice()* (expsim)
- *_nint_hspice()* (expsim)
- *_pow_hspice()* (expsim)
- *_pwr_hspice()* (expsim)
- *_sign_hspice()* (expsim)
- *_sqrt_hspice()* (expsim)

_db_hspice()

Return the base 10 log of the absolute value of x, mulitple by 20, with the sign of x

Syntax

_db_hspice(x)

Arguments

Name	Description	Default	Range	Type	Required
x	value for finding decibel	None	(-∞:∞)	Integer, Real	Yes

Examples

```
y=db(10) returns 20.0
y=db(-10) returns -20.0
```

See Also

_log_hspice() (expsim), *_log10_hspice()* (expsim), *_nint_hspice()* (expsim), *_pow_hspice()* (expsim), *_pwr_hspice()* (expsim), *_sign_hspice()* (expsim), *_sqrt_hspice()* (expsim)

_log10_hspice()

Returns the base 10 logarithm of the absolute value of x, with the sign of x

Syntax

_log10_hspice(x)

Arguments

Name	Description	Default	Range	Type	Required
x	real number	None	(-∞:∞)	Real	Yes

Examples

```
log10V=log10(100) returns 2.0
log10V=log10(-100) returns -2.0
```

See Also

_db_hspice() (expsim), *_log_hspice()* (expsim), *_nint_hspice()* (expsim), *_pow_hspice()* (expsim), *_pwr_hspice()* (expsim), *_sign_hspice()* (expsim), *_sqrt_hspice()* (expsim)

_log_hspice()

Returns the natural log of the absolute value of x, with the sign of x

Syntax

_log_hspice(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real	None	(-∞:∞)	Integer, real	Yes

Examples

```
lnV=ln(100) returns 4.605
ln2V=ln(-100) returns -4.605
```

See Also

_db_hspice() (expsim), *_log10_hspice()* (expsim), *_nint_hspice()* (expsim), *_pow_hspice()* (expsim), *_pwr_hspice()* (expsim), *_sign_hspice()* (expsim), *_sqrt_hspice()* (expsim)

_nint_hspice()

Round x up or down, to the nearest integer

Syntax

_nint_hspice(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer or real number to convert	None	(∞:∞)	Integer, real	Yes

Examples

```
intV=nint(2.3) returns 2
intV=nint(2.7) returns 3
```

See Also

_db_hspice() (expsim), *_log_hspice()* (expsim), *_log10_hspice()* (expsim), *_pow_hspice()* (expsim), *_pwr_hspice()* (expsim), *_sign_hspice()* (expsim), *_sqrt_hspice()* (expsim)

_pow_hspice()

Returns the value of x raised to the integer part of y, x^(int(y))

Syntax

_pow_hspice(x, y)

Arguments

Name	Description	Default	Range	Type	Required
x	Integer or real number	None	(-∞:∞)	Integer, real	Yes
y	exponent of the number	None	(-∞:∞)	Integer, real	Yes

Examples

```
powI=_pow_hspice(10, 3.2) returns 1000.0
```

See Also

_db_hspice() (expsim), *_log_hspice()* (expsim), *_log10_hspice()* (expsim), *_nint_hspice()* (expsim), *_pwr_hspice()* (expsim), *_sign_hspice()* (expsim), *_sqrt_hspice()* (expsim)

_pwr_hspice()

Returns the absolute value of x, raised to the y power, with the sign of x

Syntax

_pwr_hspice(x, y)

Arguments

Name	Description	Default	Range	Type	Required
x	Integer or real number	None	(-∞:∞)	Integer, real	Yes
y	exponent of the number	None	(-∞:∞)	Integer, real	Yes

Examples

```
pwrI=_pwr_hspice(10, 3) returns 1000.0
pwrI=_pwr_hspice(-10, 3) returns -1000.0
```

See Also

_db_hspice() (expsim), *_log_hspice()* (expsim), *_log10_hspice()* (expsim), *_nint_hspice()* (expsim), *_pow_hspice()* (expsim), *_sign_hspice()* (expsim), *_sqrt_hspice()* (expsim)

_sign_hspice()

Returns the absolute value of x, with the sign of y

Syntax

_sign_hspice(x, y)

Arguments

Name	Description	Default	Range	Type	Required
x	Integer or real number	None	(-∞:∞)	Integer, real	Yes
y	exponent of the number	None	(-∞:∞)	Integer, real	Yes

Examples

```
intV= returns 2
intV=sign(2.5, -5) returns -2.5
```

See Also

_db_hspice() (expsim), *_log_hspice()* (expsim), *_log10_hspice()* (expsim), *_nint_hspice()* (expsim), *_pow_hspice()* (expsim), *_pwr_hspice()* (expsim), *_sqrt_hspice()* (expsim)

_sqrt_hspice()

Returns the square root of the absolute value of x:

Syntax

_sqrt_hspice(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real	None	(-∞:∞)	real	Yes

Examples

```
sqrtV=sqrt(100) returns 10
sqrtCV=sqrt(-100) returns 10
```

See Also

_db_hspice() (expsim), *_log_hspice()* (expsim), *_log10_hspice()* (expsim), *_nint_hspice()* (expsim), *_pow_hspice()* (expsim), *_pwr_hspice()* (expsim), *_sign_hspice()* (expsim)

Math Functions for Simulator Expressions

This section describes the math functions in detail. Math Functions are used for matrix conversion, trigonometry, absolute value, etc. The functions are listed in alphabetical order.

- *abs() Expression* (expsim)
- *acos() Expression* (expsim)
- *acosh() Expression* (expsim)
- *arcsinh()* (expsim)
- *arctan()* (expsim)
- *asin() Expression* (expsim)
- *asinh() Expression* (expsim)
- *atan2() Expression* (expsim)
- *atan() Expression* (expsim)
- *atanh() Expression* (expsim)
- *bin()* (expsim)
- *ceil() Expression* (expsim)
- *complex() Expression* (expsim)
- *conj() Expression* (expsim)
- *cos() Expression* (expsim)
- *cosh() Expression* (expsim)
- *cot() Expression* (expsim)
- *coth() Expression* (expsim)
- *ctof()* (expsim)
- *ctok()* (expsim)
- *db() Expression* (expsim)
- *dbm() Expression* (expsim)
- *dbmtoa()* (expsim)
- *dbmtov()* (expsim)
- *dbmtow() Expression* (expsim)
- *dbpolar()* (expsim)
- *dbwtow()* (expsim)
- *deg() Expression* (expsim)
- *dphase()* (expsim)
- *eval_pole_zero()* (expsim)
- *eval controlled pwl()* (expsim)
- *eval miso poly()* (expsim)
- *eval poly()* (expsim)
- *exp() Expression* (expsim)
- *floor() Expression* (expsim)
- *fmod() Expression* (expsim)
- *ftoc()* (expsim)
- *ftok()* (expsim)
- *hypot() Expression* (expsim)
- *imag() Expression* (expsim)
- *int() Expression* (expsim)

- *itob()* (expsim)
- *jn() Expression* (expsim)
- *ktoc()* (expsim)
- *ktof()* (expsim)
- *ln() Expression* (expsim)
- *log10() Expression* (expsim)
- *log() Expression* (expsim)
- *mag() Expression* (expsim)
- *max() Expression* (expsim)
- *min() Expression* (expsim)
- *phase() Expression* (expsim)
- *phasedeg() Expression* (expsim)
- *phaserad() Expression* (expsim)
- *phasewrap()* (expsim)
- *polar() Expression* (expsim)
- *polarcpx()* (expsim)
- *pow() Expression* (expsim)
- *rad() Expression* (expsim)
- *real() Expression* (expsim)
- *rem()* (expsim)
- *sgn() Expression* (expsim)
- *sin() Expression* (expsim)
- *sinc() Expression* (expsim)
- *sinh() Expression* (expsim)
- *spectrum()* (expsim)
- *sqrt() Expression* (expsim)
- *sum() Expression* (expsim)
- *tan() Expression* (expsim)
- *tanh() Expression* (expsim)
- *wtodbm() Expression* (expsim)

abs()

Returns the absolute value of an integer or real number

Syntax

abs(x)

Arguments

Name	Description	Default	Range	Type	Required
x	value to find abs	None	(-∞:∞)	Integer, real, complex, or an array of those types	Yes

Examples

```
absRV=abs(-0.3) returns 0.3
absCV=abs(0.3-j*0.4) returns 0.5
absRVA=abs(list(-0.3,-0.4)) returns list(0.3,0.4)
absCVA=abs(list(-0.3-j*0.4,-0.3+j*0.4)) returns list(0.5,0.5)
```

See Also

exp() (expsim), *int()* (expsim), *log()* (expsim), *log10()* (expsim), *mag()* (expsim), *pow()* (expsim), *sgn()* (expsim), *sqrt()* (expsim)

Note
The function name abs() is used for more than one type of expression. For comparison, see the Measurement Expression *abs() Measurement* (expmeas) and the AEL Function *abs() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

acos()

Returns the arc-cosine of an integer or real number

Syntax

acos(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer or real number	None	[-1:1] †	Integer, real	Yes

† A warning will be issued if value is out of this range.

Examples

```
acosV=acos(0.9) returns 0.451
acosV=acos(-0.5) returns 2.094
```

See Also

asin() (expsim), *atan()* (expsim)

Note
The function name acos() is used for more than one type of expression. For comparison, see the Measurement Expression *acos() Measurement* (expmeas) and the AEL Function *acos() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

acosh()

Returns the inverse hyperbolic cosine of an integer or real number

Syntax

acosh(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real number	None	[1:∞) †	Integer, real	Yes

† A warning will be issued if value is out of this range.

Examples

```
acoshV=acosh(1.8) returns 1.193
```

See Also

arcsinh() (expsim), *asinh()* (expsim), *atanh()* (expsim)

Note
The function name acosh() is used for more than one type of expression. For comparison, see the Measurement Expression *acosh() Measurement* (expmeas) and the AEL Function *acosh() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

arcsinh()

Returns the inverse hyperbolic sine of an integer, real or complex number

Syntax

arcsinh(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real or complex number	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
arcsinhV=arcsinh(0.8) returns 0.733
arcsinhV=arcsinh(-0.5) returns -0.481
arcsinhCV=arcsinh(1+j*0.4) returns 0.952/17.151
```

See Also

acosh() (expsim), *asinh()* (expsim), *atanh()* (expsim)

arctan()

Returns the arc-tangent of an integer or real number

Syntax

Syntax

arctan(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer or real number	None	(-∞:∞)	Integer, real	Yes

Examples

```
atanV=atan(0.9) returns 0.733
atanV=atan(-0.5) returns -0.464
```

See Also

acos() (expsim), *asin()* (expsim), *tan()* (expsim)

asin()

Returns the arc-sine of an integer or real number

Syntax

asin(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer or real number	None	[-1:1] †	Integer, real	Yes

† A warning will be issued if value is out of this range.

Examples

```
asinV=asin(0.9) returns 1.12
asinV=asin(-0.5) returns -0.524
```

See Also

acos() (expsim), *atan()* (expsim), *atan2()* (expsim)

Note
The function name asin() is used for more than one type of expression. For comparison, see the Measurement Expression *asin() Measurement* (expmeas) and the AEL Function *asin() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

asinh()

Returns the inverse hyperbolic sine of an integer, real or complex number

Syntax

asinh(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real or complex number	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
asinhV=asinh(0.8) returns 0.733
asinhV=asinh(-0.5) returns -0.481
asinhCV=asinh(1+j*0.4) returns 0.952/17.151
```

See Also

acosh() (expsim), *arcsinh()* (expsim), *atanh()* (expsim)

Note
The function name asinh() is used for more than one type of expression. For comparison, see the Measurement Expression *asinh() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

atan2()

Returns the arctangent of y/x

Syntax

atan2(y,x)

Arguments

Name	Description	Default	Range	Type	Required
y	y value e.g. imaginary	None	(-∞:∞)	Integer, real	Yes
x	x value e.g. real	None	(-∞:∞)	Integer, real	Yes

Examples

```
y=atan(1,2) returns 0.464
y=atan(0.001,2) returns 5e-4
```

See Also

acos() (expsim), *asin()* (expsim), *atan()* (expsim), *cos()* (expsim), *sin()* (expsim), *tan()* (expsim)

Note
The function name atan2() is used for more than one type of expression. For comparison, see the Measurement Expression *atan2() Measurement* (expmeas) and the AEL Function *atan2() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

atan()

Returns the arg-tangent of an integer or real number

Syntax

atan(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer or real number	None	(-∞:∞)	Integer, real	Yes

Examples

```
atanV=atan(0.9) returns 0.733
atanV=atan(-0.5) returns -0.464
```

See Also

acos() (expsim), *asin()* (expsim), *atan2()* (expsim)

Note
The function name atan() is used for more than one type of expression. For comparison, see the Measurement Expression *atan() Measurement* (expmeas) and the AEL Function *atan() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

atanh()

Returns the inverse hyperbolic tangent of an integer or real number

Syntax

atanh(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real number	None	[-1:1) †	Integer, real	Yes

† A warning will be issued if value is out of this range.
If x equals 1, the returned value is 1e20
If x equals -1, the returned value is -1e20

Examples

```
atanhV=atanh(0.8) returns 1.099
atanhV=atanh(-0.5) returns -0.549
```

See Also

acosh() (expsim), *arcsinh()* (expsim), *asinh()* (expsim)

Note
The function name atanh() is used for more than one type of expression. For comparison, see the Measurement Expression *atanh() Measurement* (expmeas) and the AEL Function *atanh() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

bin()

Converts a binary to an integer and returns a real number

Syntax

bin(Input)

Arguments

Name	Description	Default	Range	Type	Required
Input	binary input as a string	None	[0:1] †	String	Yes

† Binary representation conatining 1's and 0's

Examples

```
binV=bin("11011") returns 27.0
```

See Also

itob() (expsim)

ceil()

Returns the ceil as a real number

Syntax

ceil(x)

Arguments

Name	Description	Default	Range	Type	Required
x	real number to find ceil	None	(-∞:∞)	Real	Yes

Examples

```
ceilV=ceil(1.8) returns 2.0
```

See Also

floor() (expsim)

Note
The function name ceil() is used for more than one type of expression. For comparison, see the Measurement Expression *ceil() Measurement* (expmeas) and the AEL Function *ceil() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

complex()

Forms a complex number in rectangular format

Syntax

complex(realValue, imageValue)

Arguments

Name	Description	Default	Range	Type	Required
realValue	real part of the complex number	None	(-∞:∞)	Integer, real	Yes
imagValue	imaginary part of the complex number	None	(-∞:∞)	Integer, real	Yes

Examples

```
complexV=complex(1,2) returns 1+j*2
```

See Also

imag() (expsim), *real()* (expsim)

Note
The function name complex() is used for more than one type of expression. For comparison, see the Measurement Expression *complex() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

conj()

Returns the complex conjugate

Syntax

conj(x)

Arguments

Name	Description	Default	Range	Type	Required
x	Value to operate on	None	(-∞:∞)	Integer, real, complex, or an array of those types	Yes

Examples

```
conjV=conj(1+j*2) returns 1-j*2
```

See Also

mag() (expsim)

Note
The function name conj() is used for more than one type of expression. For comparison, see the Measurement Expression *conj() Measurement* (expmeas) and the AEL Function *conj() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

cos()

Returns the cosine as an integer, real or complex number.

Syntax

cos(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real or complex number	None	(-∞:∞)	Integer, real or complex	Yes

Examples

```
cosRV=cos(pi) returns -1
cosRV=cos(-pi/4) returns 0.707
cosCV=cos(0.8+j*0.5) returns 0.87/-25.446
```

See Also

sin() (expsim), *tan()* (expsim)

Note
The function name cos() is used for more than one type of expression. For comparison, see the Measurement Expression *cos() Measurement* (expmeas) and the AEL Function *cos() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

cosh()

Returns the hyperbolic cosine as an integer, real or complex number

Syntax

cosh(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real, or complex number	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
coshV=cosh(0.8) returns 1.337
coshV=cosh(-0.5) returns 1.128
coshCV=cosh(0.8+j*0.5) returns 1.249/19.939
```

See Also

sinh() (expsim), *tanh()* (expsim)

Note
The function name cosh() is used for more than one type of expression. For comparison, see the Measurement Expression *cosh() Measurement* (expmeas) and the AEL Function *cosh() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

cot()

Returns the cotangent as a real or complex number

Syntax

cot(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real or complex number	None	(-∞:∞)	Real, Complex	Yes

Examples

```
cotRV=cot(pi/2) returns 6.123e-17
```

```
cotCV=cot(0.5-j*0.5) returns 1.441/54.396
```

See Also

cos() (expsim), *sin()* (expsim), *tan()* (expsim)

Note
The function name cot() is used for more than one type of expression. For comparison, see the Measurement Expression *cot() Measurement* (expmeas) and the AEL Function *cot() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

coth()

Returns the hyperbolic cotangent as a real or complex number.

Syntax

coth(x)

Arguments

Name	Description	Default	Range	Type	Required
x	Real or complex number	None	(-∞:∞)	Real, complex	Yes

Examples

```
cothV=coth(0.8) returns 1.506
cothCV=coth(0.5-j*0.5) returns 1.441/35.604
```

See Also

cosh() (expsim), *sinh()* (expsim), *tanh()* (expsim)

Note
The function name coth() is used for more than one type of expression. For comparison, see the Measurement Expression *coth() Measurement* (expmeas) and the AEL Function *coth() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

ctof()

Converts Celsius to Fahrenheit and returns a real number

Syntax

ctof(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	value in Celsius	None	(-∞:∞)	Real	Yes

Examples

```
ctofV=ctof(100) returns 212.0
```

See Also

ftoc() (expsim), *ktof()* (expsim)

ctok()

Converts Celsius to Kelvin and returns a real number

Syntax

ctok(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	Value in Celsius	None	(-∞:∞)	Real	Yes

Examples

```
ctokV=ctok(100) returns 373.15
```

See Also

ftok() (expsim), *ktoc()* (expsim)

db()

Returns the decibel as an integer or real.

Syntax

db(x)

Arguments

Name	Description	Default	Range	Type	Required
x	value for finding decibel	None	(-∞:∞)	Integer, Real, Complex	Yes

Examples

```
y=db(10) returns 20.0
```

```
y=db(1+j*2) returns 6.99
```

See Also

dbm() (expsim), *dbmtoa()* (expsim), *dbmtov()* (expsim), *dbmtow()* (expsim), *dbpolar()* (expsim), *dbwtow()* (expsim), *wtodbm()* (expsim)

Notes/Equations

This expression calculates the decibel of the given value.
The value in dB is calculated as follows:

$$Value = 20.0\log(mag(x) + tinyreal)$$

where *tinyreal* = 2.22507e-308 (see *Predefined Simulator Variables and Constants Reserved Names* (expsim))

Note
The function name db() is used for more than one type of expression. For comparison, see the Measurement Expression *db() Measurement* (expmeas) and the AEL Function *dB() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

dbm()

Converts voltage to decibel referenced to a 1 milliwatt signal and returns a real number

Syntax

dbm(Value, Zref)

Arguments

Name	Description	Default	Range	Type	Required
Value	Voltage	None	(-∞:∞)	Integer, Real, Complex	Yes
Zref	Reference impedance	None	(-∞:∞)	Integer, Real, Complex	Yes

Examples

```
y=dbm(0, 50) returns -3046.527
y=dbm(1+j*2, 25) returns 20.0
y=dbm(-30, 1+j*1) returns 53.522
```

See Also

db() (expsim), *dbmtoa()* (expsim), *dbmtov()* (expsim), *dbmtow()* (expsim), *dbpolar()* (expsim), *dbwtow()* (expsim), *wtodbm()* (expsim)

Notes/Equations

This expression calculates the decibel measure of a voltage referenced to a 1 milliwatt signal.

The value in dBm is calculated as follows:

$$Value = 10\log\left(0.5\,\mathrm{real}(Zref)\left(\frac{mag(v)}{mag(Zref)}\right)^2 + tinyreal\right)$$

where *tinyreal* = 2.22507e-308 (see *Predefined Simulator Variables and Constants Reserved Names* (expsim))

Given a power Po in Watts, the power in dB is:

$$\mathrm{Po_dB} = 10\log\left(mag\left(\frac{Po}{1\,W}\right)\right)$$

while the power in dBm is:

$$\mathrm{Po_dBm} = 10\log\left(mag\left(\frac{Po}{1m\,W}\right)\right)$$

$$= 10\log\left(\left(mag\left(\frac{Po}{1\,W}\right)\right) + 30\right)$$

$$= \mathrm{Po_dB} + 30$$

Note
The function name dbm() is used for more than one type of expression. For comparison, see the Measurement Expression *dbm() Measurement* (expmeas) and the AEL Function *dBm() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

dbmtoa()

Converts dbm into short circuit current, given the reference impedance and returns a real or complex number

Syntax

dbmtoa(Value, Zref)

Arguments

Name	Description	Default	Range	Type	Required
Value	value in dBm	None	(-∞:∞)	Real, Complex	Yes
Zref	Reference Impedance	None	(-∞:∞)	Integer, Real or Complex	Yes

Examples

```
y=dbmtoa(1+j*1,50) returns 0.014+j0.002
y=dbmtoa(0, 50) returns 0.013
y=dbmtoa(-30, 1+j*1) returns 0.002
```

See Also

db() (expsim), *dbm()* (expsim), *dbmtov()* (expsim), *dbmtow()* (expsim), *dbpolar()* (expsim), *dbwtow()* (expsim), *wtodbm()* (expsim)

Notes/Equations

This expression converts the dBm measure into short circuit current given the reference impedance. The value is calculated as follows:

$$\text{value} = \frac{\sqrt{8 \times 10^{\frac{v-30}{10}}}\ \text{real}(Zref)}{mag(Zref)}$$

dbmtov()

Converts dbm into open circuit voltage, given the reference impedance and returns a real or complex number

Syntax

dbmtov(Value, Zref)

Arguments

Name	Description	Default	Range	Type	Required
Value	value in dBm	None	(-∞:∞)	Real, Complex	Yes
Zref	Reference Impdeance	None	(-∞:∞)	Real, Complex	Yes

Examples

```
y=dbmtov(1+j*1,50) returns 0.705+j0.082
y=dbmtov(10, 50) returns 2.0
y=dbmtov(-30, 1+j*1) returns 0.003
```

See Also

db() (expsim), *dbm()* (expsim), *dbmtoa()* (expsim), *dbmtow()* (expsim), *dbpolar()* (expsim), *dbwtow()* (expsim), *wtodbm()* (expsim)

Notes/Equations

This expression converts the dBm measure into open circuit voltage given the reference impedance. The value is calculated as follows:

$$Value = \sqrt{8 \times 10^{\frac{v-30}{10}} \text{ real}(Zref)}$$

dbmtow()

Converts dBm to Watts and returns a real or complex number

Syntax

dbmtow(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	Value in dBm	None	(-∞:∞)	Real, Complex	Yes

Examples

```
y=dbmtow(1+j*1) returns 0.001+j2.873e-4
y=dbmtow(50) returns 100.0
y=dbmtow(30) returns 1.0
```

See Also

db() (expsim), *dbm()* (expsim), *dbmtoa()* (expsim), *dbmtov()* (expsim), *dbpolar()* (expsim), *dbwtow()* (expsim), *wtodbm()* (expsim)

Notes/Equations

This expression converts the dBm measure into Watts. The value is calculated as follows:

$$ValueW = \exp\left[\left(\frac{\ln(10)}{10}\right) \times (Value - 30)\right]$$

Note
The function name dbmtow() is used for more than one type of expression. For comparison, see the Measurement Expression *dbmtow() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

dbpolar()

Converts (dB,angle) to rectangular coordinates and returns a complex number

Syntax

Syntax

dbpolar(dB, Angle)

Arguments

Name	Description	Default	Range	Type	Required
dB	value in decibel	None	(-∞:∞)	Real	Yes
Angle	Angle to convert	None	[-360:360]	Real	Yes

Examples

```
dbV=6.99
value=dbpolar(dbV, 63.435) returns 1+j*2
```

See Also

db() (expsim), *dbm()* (expsim), *dbmtoa()* (expsim), *dbmtov()* (expsim), *dbmtow()* (expsim), *dbwtow()* (expsim), *vswrpolar()* (expsim), *wtodbm()* (expsim)

Notes/Equations

This expression converts the (dB, angle) to rectangular format. The dbpolar is calculated as follows:

$$Value = 10^{\frac{dB}{20}} \cos\left(\frac{(Angle)\pi}{180}\right) + j10^{\frac{dB}{20}} \sin\left(\frac{(Angle)\pi}{180}\right)$$

dbwtow()

Converts dBW to Watts and returns a real or complex number

Syntax

dbwtow(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	Value in dBW	None	(-∞:∞)	Real, Complex	Yes

Examples

```
y=dbwtow(1+j*1) returns 1.226+j0.287
y=dbwtow(50) returns 100000.0
y=dbwtow(-30) returns 10e-4
```

See Also

db() (expsim), *dbm()* (expsim), *dbmtoa()* (expsim), *dbmtov()* (expsim), *dbmtow()* (expsim), *dbpolar()* (expsim), *wtodbm()* (expsim)

Notes/Equations

This expression converts the dBw measure into Watts. The value is calculated as follows:

$$dbwtowValue = \exp\left(\frac{\ln(10)}{10 \times Value}\right)$$

deg()

Converts radian to degree and returns an integer or real

Syntax

deg(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer or real number	None	(-∞:∞)	Integer, Real	Yes

Examples

```
degV=deg(pi) returns 180.0
```

See Also

rad() (expsim)

Note
The function name deg() is used for more than one type of expression. For comparison, see the Measurement Expression *deg() Measurement* (expmeas) and the AEL Function *deg() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

dphase()

Returns the continuous phase difference (radians) between two numbers as a real number

Syntax

dphase(x, y)

Arguments

Name	Description	Default	Range	Type	Required
x	first number	None	(-∞:∞)	Integer, Real or Complex.	Yes
y	second number	None	(-∞:∞)	Integer, Real or Complex.	Yes

Examples

```
diffPC = dphase(complex(0.333,0.01),complex(0.633,0.1)) returns -0.127
diffPR = dphase(90,100) returns 0
diffP = dphase(complex(0.333,0.01),100) returns 0.03
```

See Also

phase() (expsim), *phasedeg()* (expsim), *phaserad()* (expsim)

Notes/Equations

This expression takes into account any potential 2*PI phase jumps. The jump threshold is fixed at PI degrees and is calculated as follows:

$$diff = \begin{cases} phase(x) - phase(y) - 2\pi & (phase(x) - phase(y)) > \pi \\ phase(x) - phase(y) + 2\pi & -\pi \le (phase(x) - phase(y)) \le \pi \end{cases}$$

Note
The phase() function used here assumes a value returned in radians.

eval_pole_zero()

Optionally evaluates rational polynomial function when supplied with zeros and poles and input values.

Syntax

eval_pole_zero(zeros, poles, x, type) where zeros and poles are either in list(...) or one-dimensional makearray(1,...) form.

Arguments

Name	Description	Default	Range	Type	Required
zeros	one-dimensional array of coefficients	NONE	(-∞:∞) †	Integer, Real † †	Yes
poles	one-dimensional array of coefficients	NONE	(-∞:∞) †	Integer, Real † †	Yes
x	input variable of polynomial function	NONE	(-∞:∞) †	Integer, Real, Complex † † †	Yes
type	specifies the type of input	NONE	(-∞:∞) † † † †	Integer	Yes

† Individual elements of array may have values in this range.

† † Individual elements of array may be integer or real valued but not complex.
† † † Complex input cannot be used for derivative or integral operations.
† † † † Input type = 0 the simulator will create the complex conjugate
for each complex zero/pole; = 1 the simulator assumes all poles and zeros are specified in the
input arguments.

Examples

1. **Complex poles/zeros with type = 0**
   ```
   result = eval_pole_zero( list(2,1,2), list(5,3,7), x, 0 )
   = 2*(x + 1 + j*2*PI*2)(x + 1 - j*2*PI*2)/[5(x + 3 + j*2*PI*7)*(x + 3 - j*2*PI*7)]
   ```

2. **Complex poles/zeros with type = 1**
   ```
   result = eval_pole_zero( list(2,1,2), list(5,3,7), x, 1 )
   = 2*(x + 1 + j*2*PI*2)/[5*(x + 3 + j*2*PI*7)]
   ```

3. **Real poles/zeros**
   ```
   result = eval_pole_zero( list(2,1,0), list(5,3,0), x, 1 )
   = 2*(x + 1)/[5*(x + 3)]
   ```
   Real poles/zeros are not affected by the value of type.

4. **Duplicate poles/zeros**
   ```
   result = eval_pole_zero( list(2,1,0,1,0), list(5,3,0,2,5,2,5), x, 0 )
   = 2*(x + 1)^2/[5*(x + 3)*(x + 2 + j*2*PI*5)^2*(x + 2 - j*2*PI*5)^2]
   ```

Notes / Equations

1. eval_pole_zero(list(a az1, fz1, ..., azn, fzn), list(b, ap1, fp1, ..., apm, fpm), s, 0) specifies the following function:

$$H(s)=\frac{a\cdot(s+\alpha_{z1}-j2\pi f_{z1})...(s+\alpha_{zn}-j2\pi f_{zn})(s+\alpha_{zn}+j2\pi f_{zn})}{b\cdot(s+\alpha_{p1}-j2\pi f_{p1})...(s+\alpha_{pm}-j2\pi f_{pm})(s+\alpha_{pm}+j2\pi f_{pm})}$$

2. If the type is set to 1 instead of 0, that is, eval_pole_zero(list(a az1, fz1, ..., azn, fzn), list(b, ap1, fp1, ..., apm, fpm), s, 1), the function will be:

$$H(s)=\frac{a\cdot(s+\alpha_{z1}+j2\pi f_{z1})...(s+\alpha_{zn}+j2\pi f_{zn})}{b\cdot(s+\alpha_{p1}+j2\pi f_{p1})...(s+\alpha_{pm}+j2\pi f_{pm})}$$

eval_controlled_pwl()

Evaluates the piece-wise linear response of system when supplied with N pairs of data points in vector format.

Syntax

eval_controlled_pwl(pwlvector, xin, [stretch], [scale]) where pwlvector is a list(in1, out1, ..., inN, outN)

Arguments

Name	Description	Default	Range	Type	Required
pwlvector	list of pairs of datapoints defining corners of the PWL function	NONE	list of (-∞:∞) † † †	Integer, Real † †	Yes †
xin	scalar input signal	0.0	(-∞:∞)	Integer, Real, Complex	Yes
stretch	used for scaling input signal prior to PWL computation	1.0	(-∞:0) U (0:∞)	Integer, Real	No
scale	used for scaling output signal following PWL computation	1.0	(-∞:∞)	Integer, Real	No

† At least one pair of data points should be supplied.
† † Only scalar inputs should be supplied.
† † † First element should always be listed. If Ith element is listed then (I-1)th element should also be explicitly listed.

Examples

The response of a system described by the three-point PWL vector list(in 1 , out 1 , in 2 , out 2 , in 3 , out 3) to input *xin* , *stretch* , and *scale* is
xout = eval_controlled_pwl(list(in 1 , out 1 , in 2 , out 2 , in 3 , out 3), xin, stretch, scale)

- The input is stretched up to
  xin~ = min(xin*stretch, stretchEps) where *stretchEps* is an internal limit generated when *stretch* is too close to zero.
- The proper bin for *xin~* is detected using the three corner points along input axis: in_1, in_2, and in_3, which may be out of order in the list() or file forms.
- If *xin~* falls within bounded limits, e.g., in_1 and in_3, then *xout~* is computed using linear equation:
  (xout~ - out 3) * (in 1 - in 3) = (xin~ - in 3) * (out 1 - out 3) else, if it falls in either of the two unbounded sections with nearest data point in_i , out_i, then constant extrapolation is done as follows:
  xout~ = out i
- If scale factor is defined, it is used on the raw output. If not, *scale* is assumed to be unity. Thus:
  xout = xout~ * scale
  Given: pwlvector = list(2, 4, 3, -1, -1, 3)
  xin=1

```
stretch=0.75
scale=3.0
```

xin~ {{= 3/4

which falls within input side range {{(-1,2] with output side range (3, 4] . Thus,

```
xout~ = (3/4 - (-1))/(2 - (-1)) * (4 - 3) + 3 = 7/12 * 1 + 3 = 43/3
xout = 43/3 * 3.0 = 43
```

Notes/Equations

1. This function exists in ADS primarily to emulate the behavior of piece-wise linear (PWL) behavior of single input, single output controlled voltage and current sources of the Cadence Spectre library. For further information about the sources *cccs* , *ccvs* , *vccs* , and *vcvs* , see the Cadence Spectre simulator's analog library documentation.
2. This function supports PWL values either directly in list() or indirectly in file format. For PWL files, the following approach is recommended to create a list-based entry for the SYM function:
   pwlData = read_data("file","<pwlvector_filename>","sppwl")
   pwlIn = dstoarray(pwlData,"inputPWL")
   pwlOut = dstoarray(pwlData,"outputPWL")
   xout = eval_controlled_pwl(list(pwlIn, pwlOut), xin[, stretch[, scale]])
   In this case the list() function concatenates the two vectors *pwlIn* and *pwlOut* by interleaving the values as expected by eval_controlled_pwl().
   A PWL file is a simple ASCII data file containing two columns, for input and output values marking corner points of the function. This file may contain comment lines preceded by the ";" character. No explicit format information is expected from such a file. An example of a PWL file is:
   ; PWLfile
   ; Input Output
   2 4.0
   3 -1.0
   -1 3.0
   The file contents are equivalent to
   list(2, 4.0, 3, -1.0, -1, 3.0) Data points may be supplied out of order in either list or file format. The function automatically checks for duplicate points, which it tolerates, for ambiguity, such as one-to-many input-output mapping causing an error when encountered.
3. Default value of *stretch* is 1.0, but if a value is supplied and found to be too close to zero, its effective value is limited to a small non-zero number. This prevents interpolation errors due to infinitesimal shrinkage of input *xin* by the stretch factor. It ensures that *xin*~ is truly 0.0 only when *xin* is 0.0.
4. Only the real part of a complex input variable is affected by the PWL. The imaginary part merely gets stretched and scaled.

eval_miso_poly()

Evaluates the multi-input-single-output (miso) polynomial response of an M-input system when supplied with N+1 coefficients list format.

Syntax

eval_miso_poly(coefs, x1, ... ,xM) where coefs is a list(c0, ..., cN)

Arguments

Name	Description	Default	Range	Type	Required
coefs	list of numbers or scalar variables formed using simulator expression list()	list(1.0)	list of (-∞:∞) †	Integer, Real † †	Yes † † †
xJ	scalar input signal at Jth port of system, J >= 1.	x1=0.0	(-∞:∞)	Integer, Real, Complex †	Yes † † †

† At least one non-zero coefficient should be present.
† † Individual coefficients in list may be integer or real valued.
† † † First element should always be defined. If Ith element is listed then (I-1)th element should also be explicitly listed.

Examples

1. The response of a three-input, 13-coefficient system:
   $y=eval_miso_poly(list(c_0,c_1,c_2,c_3,c_4,c_5,c_6,c_7,c_8,c_9,c_{10},c_{11},c_{12}), x_1, x_2, x_3)$
   returns
   $y=c_0 + c_1 \cdot x_1 + c_2 \cdot x_2 + c_3 \cdot x_3 + c_4 \cdot x_1^2 + c_5 \cdot x_1 \cdot x_2 + c_6 \cdot x_1 \cdot x_3 + c_7 \cdot x_2^2 + c_8 \cdot x_2 \cdot x_3 + c_9 \cdot x_3^2 + c_{10} \cdot x_1^3 + c_{11} \cdot x_1^2 \cdot x_2 + c_{12} \cdot x_1^2 \cdot x_3$
   Therefore, if all coefficients were 1 and the three inputs were 2, 3, and 5 respectively,
   $y=1 + 2 + 3 + 5 + 4 + 6 + 10 + 9 + 15 + 25 + 8 + 12 + 20 = 120$
2. The response of a four-input, 13-coefficient system:
   $y=eval_miso_poly(list(c_0,c_1,c_2,c_3,c_4,c_5,c_6,c_7,c_8,c_9,c_{10},c_{11},c_{12}), x_1, x_2, x_3, x_4)$
   returns
   $y=c_0 + c_1 \cdot x_1 + c_2 \cdot x_2 + c_3 \cdot x_3 + c_4 \cdot x_4 + c_5 \cdot x_1^2 + c_6 \cdot x_1 \cdot x_2 + c_7 \cdot x_1 \cdot x_3 + c_8 \cdot x_1 \cdot x_4 + c_9 \cdot x_2^2 + c_{10} \cdot x_2 \cdot x_3 + c_{11} \cdot x_2 \cdot x_4 + c_{12} \cdot x_3^2$
   Therefore, if all coefficients were 1 and the four inputs were 2, 3, 5, and 7 respectively,
   $y=1 + 2 + 3 + 5 + 7 + 4 + 6 + 10 + 14 + 9 + 15 + 21 + 25 = 122$

See Also

eval_poly() (expsim)

Notes/Equations

1. This function exists in ADS primarily to emulate the behavior of polynomially-controlled voltage and current sources of the Cadence Spectre library. For further information about the sources *pcccs* , *pccvs* , *pvccs* , and *pvcvs* , see the Cadence Spectre simulator's analog library documentation.
2. This function does not support array-based inputs. All x_J must be scalar numbers or

variables. Contrast this with the *eval_poly()* function which does allow array-based inputs. The functionality of these two expressions is vastly different as is evidenced by notes and examples.

3. This function supports coefficient values either directly in list() format or indirectly via a coefficient file format. If coefficient files must be used, the following approach is recommended:
   {{coefData = read_data("file","<coefvector_filename>","spcoef")
   coefVect = dstoarray(coefData,"coef")
   y = eval_miso_poly(coefVect, x}} 1 , ... ,× M)
   A coefficient file is a simple ASCII data file containing two columns, for coefficient indices and values respectively. This file may contain comment lines preceded by the ";" character. No explicit format information is expected from such a file. An example of a coefficient file is:
   ; Coeffile
   ; Col 1 Col2
   0 3.4
   3 -2.7
   1 6.2
   4 1.3
   The file contents are equivalent to:
   `list(3.4, 6.2, 0.0, -2.7, 1.3)`
   Within a file, the coefficients may be supplied out of order and absent coefficients are noted as zero. However, when presenting coefficients in list format, the values should be explicitly mentioned and in sequential order.
4. Only contiguous sequences of coefficients and inputs are supported. If coefficient c_I is given, all its predecessors must be explicitly specified in the list. No blank spaces signifying missing elements are supported. On the other hand, if x_J is specified in the function, then all J-1 inputs before it either must be specified explicitly or represented by a space between successive delimiter instances, i.e. ", ,".
5. The distribution of polynomial coefficients among inputs and their higher order combinations depends the number of input variables M and the number of coefficients N+1 where M, N are integers > 0. For a system with a fixed M-number of inputs, the degree of polynomial is determined by the number of coefficients supplied. Conversely, given a fixed list of coefficients, the system response depends on number of inputs listed, even if certain inputs are listed as zero-valued.
6. To understand the mathematical framework on which *eval_miso_poly()* is based, consider the order in which polynomial summands are created in the example section above. Each polynomial summand is a product of all the input signals raised to various integral exponents, e.g. $x_1^2 \cdot x_3$ for a three-input system can be interpreted as $x_1^2 \cdot x_2^0 \cdot x_3^1$, where the vector of exponents is [2, 0, 1]. The same term, for a four-input system would be interpreted as $x_1^2 \cdot x_2^0 \cdot x_3^1 \cdot x_4^0$, where the vector of exponents is [2, 0, 1, 0].
   Each polynomial summand for a system with fixed number of inputs has a 1-to-1 relationship with the vector of coefficients. The order in which coefficients are distributed among summands and therefore among exponent vectors is dependent on the order in which summands are produced by left-longhand multiplication (without multiplicative coefficients) of successive orders of polynomial summands.

An example of such multiplication is shown using three scalar numbers a, b, c. The first order vector formed by these numbers is [a b c]. Left long-hand multiplication of this vector with itself would mean an inner product of the vector with its transpose such that the duplicate terms are removed from the sequence.

```
a b c
a b c
-----------
```

`a.a a.b a.c` - Pivoting at 1st element *a* of the lower or multiplier vector

`b.a b.b b.c` - Pivoting at 2nd element *b* of the lower or multiplier vector

`c.a c.b c.c` - Pivoting at 3rd element *c* of the lower or multiplier vector

Ignore duplicate scalar values `b.a, c.a` and `c.b` because `a.b` , `a.c` , and `b.c` have already been computed in advance.

The remaining summands forming the 2nd-order vector are

`a^2 a.b a.c b^2 b.c c^2`

This is the result of 2nd-order left long-hand multiplication using two variables. For this square matrix, is the equivalent of considering only diagonal and upper triangle elements scanned in raster form.

Given a three-input system,

- There is only one 0th-order summand.
  $x_1^0 \cdot x_2^0 \cdot x_3^0 = 1$ which is assigned coefficient c_0.
- There are three 1st-order summands.
  $x_1^1 \cdot x_2^0 \cdot x_3^0 = x_1$,
  $x_1^0 \cdot x_2^1 \cdot x_3^0 = x_2$,
  $x_1^0 \cdot x_2^0 \cdot x_3^1 = x_3$,
  which are assigned coefficients $c_{1..3}$ respectively.
- There are six 2nd-order summands listed without multiplicative coefficients in order of appearance due to left long-hand multiplication.
  $x_1^2 \cdot x_2^0 \cdot x_3^0 = x_1^2$,
  $x_1^1 \cdot x_2^1 \cdot x_3^0 = x_1 \cdot x_2$,
  $x_1^1 \cdot x_2^0 \cdot x_3^1 = x_1 \cdot x_3$,
  $x_1^0 \cdot x_2^2 \cdot x_3^0 = x_2^2$,
  $x_1^0 \cdot x_2^1 \cdot x_3^1 = x_2 \cdot x_3$,
  $x_1^0 \cdot x_2^0 \cdot x_3^2 = x_3^2$,
  which are assigned coefficients $c_{4..9}$ respectively.
- There are ten 3rd-order summands listed without multiplicative coefficients in order of appearance due to left long-hand multiplication of the 2nd-order list by the original input list.

$x_1^3 \cdot x_2^0 \cdot x_3^0 = x_1^3$,
$x_1^2 \cdot x_2^1 \cdot x_3^0 = x_1^2 \cdot x_2$,
$x_1^2 \cdot x_2^0 \cdot x_3^1 = x_1^2 \cdot x_3$,
$x_1^1 \cdot x_2^2 \cdot x_3^0 = x_1 \cdot x_2^2$,
$x_1^1 \cdot x_2^1 \cdot x_3^1 = x_1 \cdot x_2 \cdot x_3$,
$x_1^1 \cdot x_2^0 \cdot x_3^2 = x_1 \cdot x_3^2$,
$x_1^0 \cdot x_2^3 \cdot x_3^0 = x_2^3$,
$x_1^0 \cdot x_2^2 \cdot x_3^2 = x_2^2 \cdot x_3$,
$x_1^0 \cdot x_2^1 \cdot x_3^2 = x_2 \cdot x_3^2$,
$x_1^0 \cdot x_2^0 \cdot x_3^3 = x_3^3$,
which are assigned coefficients $c_{10..19}$ respectively.

eval_poly()

Optionally evaluates polynomial function, or derivative or integral of polynomial when supplied with coefficients and input values.

Syntax

eval_poly(coefs, x, type) where coefs is either in list(...) or one-dimensional makearray(1,...) form.

Arguments

Name	Description	Default	Range	Type	Required
coefs	one-dimensional array of coefficients	NONE	(-∞:∞) †	Integer, Real † †	Yes
x	input variable of polynomial function	NONE	(-∞:∞) †	Integer, Real, Complex † † †	Yes
type	specifies the type of operation	NONE	(-∞:∞) † † † †	Integer	Yes

† Individual elements of array may have values in this range.
† † Individual elements of array may be integer or real valued but not complex.
† † † Complex input cannot be used for derivative or integral operations.
† † † † Function type = 0 for direct polynomial, < 0 for integral of polynomial, > 0 for derivative of polynomial.

Examples

```
The following variables are used to demonstrate the behavior of this function:
c = list(7)
Carr = list(7,11,13,17)
x = 2
Xarr = list(2,3,5)
y = 2+j*2
Yarr = list(2+j*2,3+j*3,5+j*5)
```

1. **Direct polynomial evaluation**
   Recommended use is for a vector of real coefficients and a scalar input

which may be integer, real or complex:

result = eval_poly(Carr, x, 0)

$= 7 + 11 * 2 + 13 * 2^2 + 17 * 2^3\}\}$

= 217

result = eval_poly(Carr, y, 0)

$= 7 + 11 * (2+j*2) + 13 * (2+j*2)^2 + 17 * (2+j*2)^3$

= 243 + j*398

Other combinations yield degenerate cases:

result = eval_poly(c, x, 0) = 7

result = eval_poly(c, y, 0) = 7

result = eval_poly(c, Xarr, 0) = 7

result = eval_poly(c, Yarr, 0) = 7

An overloaded use is made of the eval_poly() function of type=0 for partial derivative computation when both coefficient and input are in vector form. When both the first and second arguments are vectors, an inner or dot product of the two is generated instead of the algebraic polynomial explained thus far. The dot product is performed as far as possible along the shorter of the two vectors. In this example, it is done until the third element of either vector. This is useful for several specialized operations in ADS.

Thus: result = eval_poly(Carr, Xarr, 0)

$= d/dx\,0\,(7 * x\,0 + 11 * x\,1 + 13 * x\,2)\}\}$

= 7

Likewise, given a complex input, the result is the same:

result = eval_poly(Carr, Yarr, 0) = 7

2. Derivative of evaluated polynomial

Recommended use is for a vector of real coefficients and a scalar input which may be integer or real but not complex:

result = eval_poly(Carr, x, 1)

$= 11 + 2 * (13 * 2) + 3 * (17 * 2^2)$

= 267

Using a scalar coefficient yields degenerate cases which correspond to the direct function responses highlighted above:

result = eval_poly(c, x, 1)

= d/dx(eval_poly(c, x, 0))

= 0

result = eval_poly(c, Xarr, 1)

= d/dx(eval_poly(c, Xarr, 0))

= 0

In keeping with the idea of partial derivative behavior signaled by vector on vector operation using this function, when type = 1, the partial derivative of the polynomial is computed with respect to x_1:

result = eval_poly(Carr, Xarr, 1)

$= d/dx\,1\,(7 * x\,0 + 11 * x\,1 + 13 * x\,2)\}\}$

= 11

This use model is only restricted to type = 1. Setting type > 1 yields zero output.

3. **Integral of evaluated polynomial**

   Recommended use is for a vector of real coefficients and a scalar input which may be integer or real but not complex:

   result = eval_poly(Carr, x, -1)

   = 7 * 2 + 11/2 * 2^2 {{+ 13/3 * 2^3 + 17/4 * 2^4}}

   = 138.667

   Using a scalar coefficient yields degenerate cases which correspond to the direct function responses highlighted above:

   result = eval_poly(c, x, -1)

   = integral x (eval_poly(c, x, 0))

   = c 0 * x

   result = eval_poly(c, Xarr, -1)

   = integral x (eval_poly(c, Xarr, 0))

   = c 0 {{* x 0}}

See Also

eval_miso_poly() (expsim)

Notes / Equations

1. It is recommended that general use of this function be restricted to scalar inputs and coefficient vectors of length greater than 1. Under this condition, the direct polynomial, derivative and integral functions of the polynomial work as expected algebraically. All other variations of argument types exist to support specialized functionalities for various components and designs and may not be of interest to the average user.
2. The differences between this and the eval_miso_poly() function are as follows:
   - The function eval_miso_poly() can operate on dynamically varying inputs since each is supplied as an independent argument of the function and not as a pre-compiled list as is necessary for eval_poly().
   - The function eval_poly() can perform differentiation and integration operations but eval_miso_poly() cannot.
   - When working with vector input rather than scalar input the eval_miso_poly() function is recommended over eval_poly() if the intention is to use the left long-hand multiplication approach to distributing coefficients. Otherwise any arbitrary composite function should be devised using several simple eval_poly() operations.

exp()

Returns the exponential as an integer, real or complex number

Syntax

exp(x, MaxExpArg)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real or complex number	None	(-∞:∞)	Integer, real, complex	Yes
MaxExpArg	maximum value of argument	60	(-∞:709]	Integer, real	No

Examples

```
expV=exp(10) returns 22026.466
exp2V=exp(-0.001) returns 0.999
exp3V=exp(0.5) returns 1.649
expCV=exp(0.5+j*0.5) returns 1.649/28.648
```

See Also

abs() (expsim), *int()* (expsim), *log()* (expsim), *log10()* (expsim), *pow()* (expsim), *sgn()* (expsim), *sqrt()* (expsim)

Notes/Equations

The output value is calculated as follows:

```
exp\(x,\[MaxExpArg\]\) = if \(x<MaxExpArg\) then exp\(x\) else
\(1\+x-MaxExpArg\)\*exp\(MaxExpArg\)
```

If you are using Advanced Design System, the MaxExpArg argument can also be set using the Options Component. You can set the MaxExpArg argument in the ADS Options component using the *Other* parameter as follows:

```
Other = "MaxExpArg=default-value"
```

Modification to the exp() function

The exp() simulator expression was modified in order to make symbolically defined device (SDD) modeling more robust. However, the new default behavior can interact and break existing user implementations of soft_exp() or equivalent functions.

For real values, it is now always a soft limited exponential that turns into a linear function above the maximum argument value.

```
exp(x,[max_arg]) = if (x < max_arg) then exp(x) else (1 + x -
max_arg)*exp(max_arg)
```

It is only continuous to the first derivative. The maximum argument value can be explicitly specified by the optional second argument to the exp() function. If this argument is not provided, it uses a global gCircuit value that can be set by the Options parameter MaxExpArg. The default value for this is 60.0. The MaxExpArg parameter must be entered using the Other = parameter.

Note
The function name exp() is used for more than one type of expression. For comparison, see the Measurement Expression *exp() Measurement* (expmeas) and the AEL Function *exp() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

floor()

Returns the floor as an real number

Syntax

floor(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer or real number	None	(-∞:∞)	Integer, real	Yes

Examples

```
floorV=floor(1.8) returns 1.0
floorV=floor(-1.8) returns -2.0
```

See Also

ceil() (expsim)

Note
The function name floor() is used for more than one type of expression. For comparison, see the Measurement Expression *floor() Measurement* (expmeas) and the AEL Function *floor() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

fmod()

Returns the remainder of the division as a real number

Syntax

fmod(a, b)

Arguments

Name	Description	Default	Range	Type	Required
a	dividend	None	(-∞:∞)	Integer, real	Yes
b	divisor	None	(-∞:∞)	Integer, real	Yes

Examples

```
fmodV=fmod(1.2, 0.31) returns 0.27
```

See Also

rem() (expsim)

Note
The function name fmod() is used for more than one type of expression. For comparison, see the Measurement Expression *fmod() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

ftoc()

Converts Fahrenheit to Celsius and returns a real number

Syntax

ftoc(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	value in Fahrenheit	None	(-∞:∞)	Real	Yes

Examples

```
ftocV=ftoc(32) returns 0
```

See Also

ctof() (expsim), *ktoc()* (expsim)

ftok()

Converts Fahrenheit to Kelvin and returns a real number

Syntax

ftok(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	value to convert in Fahrenheit	None	(-∞:∞)	Real	Yes

Examples

```
ftokV=ftok(32) returns 273.15
```

See Also

ctok() (expsim), *ktof()* (expsim)

hypot()

Returns the hypotenuse as a real or complex number

Syntax

hypot(x, y)

Arguments

Name	Description	Default	Range	Type	Required
x	x value	None	(-∞:∞)	Integer, real, complex	Yes
y	y value	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
hypotV=hypot(1,2) returns 2.236
hypotCV=hypot(1+j*0.2,2-j*1.1) returns 2.342/-23.424
```

Note
The function name hypot() is used for more than one type of expression. For comparison, see the Measurement Expression *hypot() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

imag()

Returns the imaginary part of a complex number as a real number

Syntax

imag(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real or complex number	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
imagV=imag(1+j*2) returns 2.0
```

See Also

complex() (expsim), *real()* (expsim)

Note
The function name imag() is used for more than one type of expression. For comparison, see the Measurement Expression *imag() Measurement* (expmeas) and the AEL Function *imag() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

int()

Converts to integer number

Syntax

int(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer or real number to convert	None	(∞:∞)	Integer, real	Yes

Examples

```
intV=int(2.3) returns 2
```

See Also

abs() (expsim), *exp()* (expsim), *log10()* (expsim), *pow()* (expsim), *sgn()* (expsim), *sqrt()* (expsim)

Note
The function name int() is used for more than one type of expression. For comparison, see the Measurement Expression *int() Measurement* (expmeas) and the AEL Function *int() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

itob()

Converts integer to binary and returns a string

Syntax

itob(IntegerValue, NumberBits)

Arguments

Name	Description	Default	Range	Type	Required
IntegerValue	Integer to convert to binary	int(log(IntegerValue)/log(2))	(-∞:∞)	Integer	Yes
NumberBits	Number of bits	None	[1:∞)	Integer	No

Examples

```
itobV=itob(27) returns "11011"
```

See Also

bin() (expsim)

jn()

Computes the bessel function of the first kind and returns a real number.

Syntax

jn(n, x)

Arguments

Name	Description	Default	Range	Type	Required
n	Order	None	[0:∞)	Integer, Real	Yes
x	Value	None	(-∞:∞)	Integer, Real	Yes

Examples

```
j0_15 = jn(0,15) returns -0.014
j10_15 = jn(10, 15) returns -0.09
```

Note
The function name jn() is used for more than one type of expression. For comparison, see the Measurement Expression *jn() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

ktoc()

Converts Kelvin to Celsius and returns a real number

Syntax

ktoc(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	Value in Kelvin	None	(-∞:∞)	Real	Yes

Examples

```
ktocV=ktoc(212) returns -61.15
```

See Also

ctok() (expsim), *ftoc()* (expsim)

ktof()

Converts Kelvin to Fahrenheit and returns a real number

Syntax

ktof(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	value to convert in Kelvin	None	(-∞:∞)	Real	Yes

Examples

```
ktofV=ktof(273.15) returns 32.00
```

See Also

ctof() (expsim), *ftok()* (expsim)

ln()

Returns the natural log as an integer, real or complex

Syntax

ln(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real, complex number	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
lnV=ln(100) returns 4.605
ln2V=ln(-100) returns 5.575/34.301
lnCV=ln(0.3-j*0.9) returns 1.25/-92.415
```

See Also

log() (expsim), *log10()* (expsim)

Note
The function name ln() is used for more than one type of expression. For comparison, see the Measurement Expression *ln() Measurement* (expmeas) and the AEL Function *ln() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

log10()

Returns the log to base 10 as an real or complex number

Syntax

log10(x)

Arguments

Name	Description	Default	Range	Type	Required
x	real or complex number	None	(-∞:∞)	Real, complex	Yes

Examples

```
log10V=log10(100) returns 2.0
log10V=log10(-100) returns 2.421/34.301
log10CV=log10(1+j*0.9) returns 0.343/67.961
```

See Also

ln() (expsim), *log()* (expsim)

Note
The function name log10() is used for more than one type of expression. For comparison, see the Measurement Expression *log10() Measurement* (expmeas) and the AEL Function *log10() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

log()

Returns the log to base 10 as a real or complex number

Syntax

log(x)

Arguments

Name	Description	Default	Range	Type	Required
x	real or complex number	None	(-∞:∞)	Real, complex	Yes

Examples

```
logV=log(100) returns 2.0
```

```
log1V=log(-100) returns 2.421/34.301
logCV=log(1+j*0.9) returns 0.343/67.961
```

See Also

ln() (expsim), *log10()* (expsim)

Note
The function name log() is used for more than one type of expression. For comparison, see the Measurement Expression *log() Measurement* (expmeas) and the AEL Function *log() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

mag()

Returns the magnitude of a complex number

Syntax

mag(x)

Arguments

Name	Description	Default	Range	Type	Required
x	number to find the magnitude	None	(-∞:∞)	Complex, or an array of complex	Yes

Examples

```
magRV=mag(-0.3) returns 0.3
magCV=mag(0.3-j*0.4) returns 0.5
magRVA=mag(list(-0.3,-0.4)) returns list(0.3,0.4)
magCVA=mag(list(-0.3-j*0.4,-0.3+j*0.4)) returns list(0.5,0.5)
```

See Also

abs() (expsim), *conj()* (expsim)

Note
The function name mag() is used for more than one type of expression. For comparison, see the Measurement Expression *mag() Measurement* (expmeas) and the AEL Function *mag() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

max()

Returns the maximum of two numbers as a real number

Syntax

max(x, y)

Arguments

Name	Description	Default	Range	Type	Required
x	first number	None	(-∞:∞)	Integer, real	Yes
y	second number	None	(-∞:∞)	Integer, real	Yes

Examples

```
maxV=max(1,2) returns 2.0
```

See Also

min() (expsim)

Note
The function name max() is used for more than one type of expression. For comparison, see the Measurement Expression *max() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

min()

Returns the minimum of two numbers as a real number

Syntax

min(x, y)

Arguments

Name	Description	Default	Range	Type	Required
x	first number	None	(-∞:∞)	Integer, real	Yes
y	second number	None	(-∞:∞)	Integer, real	Yes

Examples

```
minV=min(1,2) returns 1.0
```

See Also

max() (expsim)

Note
The function name min() is used for more than one type of expression. For comparison, see the Measurement Expression *min() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

phase()

Returns the phase in degrees as a real number

Syntax

phase(x)

Arguments

Name	Description	Default	Range	Type	Required
x	number to find the phase	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
phaseV=phase(1+j*2) returns 64.435
```

See Also

phasedeg() (expsim), *phaserad()* (expsim)

Note
The function name phase() is used for more than one type of expression. For comparison, see the Measurement Expression *phase() Measurement* (expmeas) and the AEL Function *phase() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

phasedeg()

Returns the phase in degrees as a real number

Syntax

phasedeg(x)

Arguments

Name	Description	Default	Range	Type	Required
x	number to find the phase	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
phasedegV=phasedeg(1+j*2) returns 64.435
```

See Also

phase() (expsim), *phaserad()* (expsim)

Note
The function name phasedeg() is used for more than one type of expression. For comparison, see the Measurement Expression *phasedeg() Measurement* (expmeas) and the AEL Function *phasedeg() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

phaserad()

Returns phase to radians as a real number

Syntax

phaserad(x)

Arguments

Name	Description	Default	Range	Type	Required
x	number to find the phase	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
phaseradV=phaserad(1+j*2) returns 1.107
```

See Also

phase() (expsim), *phasedeg()* (expsim)

Note
The function name phaserad() is used for more than one type of expression. For comparison, see the Measurement Expression *phaserad() Measurement* (expmeas) and the AEL Function *phaserad() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

phasewrap()

Returns wrapped phase value in range [0,360) or [0,2*PI) when supplied raw phase value.

Syntax

wrappedPhase = phasewrap(rawPhase, unitString) where unitString={"deg","rad"} is optional.

Arguments

Name	Description	Default	Range	Type	Required
rawPhase	user supplied phase value	NONE	(-∞:∞)	Integer, Real	Yes
unitString	optional user-identified phase unit	"deg"	{"deg", "rad"}	String	No

Examples

```
phasewrap( -95 ) evaluates to phasewrap( 360-95 ) = 265 degrees
phasewrap( 390 ) evaluates to phasewrap( 30 ) = 30 degrees
phasewrap( x, "deg" ) = phasewrap( x )
phasewrap( -0.5*pi, "rad" ) evaluates to 1.5*pi in radians
phasewrap( 3.5*pi, "rad" ) evaluates to 1.5*pi in radians
```

Notes / Equations

1. This function can handle only scalar phase value *rawPhase* .
2. The closest reverse function is measurement expression *unwrap* () which unravels a vector of wrapped phases relative to a reference phase value.

polar()

Builds a complex number from magnitude and angle (in degrees)

Syntax

polar(x, y)

Arguments

Name	Description	Default	Range	Type	Required
x	magnitude part of the complex number	None	(-∞:∞)	Integer, real	Yes
y	phase part of the complex number in degrees	None	(-∞:∞)	Integer, real	Yes

Examples

```
polarV=polar(1, 90) returns 1/90
```

See Also

complex() (expsim)

Note
The function name polar() is used for more than one type of expression. For comparison, see the Measurement Expression *polar() Measurement* (expmeas) and the AEL Function *polar() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

polarcpx()

Polar to rectangular conversion function

Syntax

polarcpx(Number, Convert)

Arguments

Name	Description	Default	Range	Type	Required
Number	Number or array to convert	None	(-∞:∞)	Integer, real, complex, array	Yes
Convert	Converts result to real (for integer, real)	0	[0:1]	Integer	No

Examples

```
y=polarcpx(polar(1,90)) returns 3.308e-17 + j*5.153e-17
y=polarcpx(makearray(2,polar(1,90),polar(2,90))) returns (3.308e-17, 5.153e-17), (-5.096e-17, 1.113e-16)
```

See Also

polar() (expsim)

pow()

Calculates the power and returns x**y as an integer, real or complex number.

Syntax

pow(x, y)

Arguments

Name	Description	Default	Range	Type	Required
x	Integer or real number	None	(-∞:∞)	Integer, real, complex	Yes
y	exponent of the number	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
powI=pow(10, 3) returns 1000.0
powC=pow(1+j*2,3+j*3) returns 0.404/-31.374
```

See Also

abs() (expsim), *exp()* (expsim), *int()* (expsim), *log10()* (expsim), *sgn()* (expsim), *sqrt()* (expsim)

Notes

1. For very large parameter values, pow() converts integers to real numbers, and returns a real number.

Note
The function name pow() is used for more than one type of expression. For comparison, see the Measurement Expression *pow() Measurement* (expmeas) and the AEL Function *pow() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

rad()

Converts degree to radian and returns integer or real number.

Syntax

rad(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer or real number to convert in degrees	None	(-∞:∞)	Integer, real	Yes

Examples

```
radV=rad(180) returns 3.142
```

See Also

deg() (expsim)

Note
The function name rad() is used for more than one type of expression. For comparison, see the Measurement Expression *rad() Measurement* (expmeas) and the AEL Function *rad() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

real()

Returns the real part as a real number

Syntax

real(x)

Arguments

Name	Description	Default	Range	Type	Required
x	number to find the real part	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
realV=real(1+j*2) returns 1.0
```

See Also

complex() (expsim), *imag()* (expsim)

Note
The function name real() is used for more than one type of expression. For comparison, see the Measurement Expression *real() Measurement* (expmeas) and the AEL Function *real() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

rem()

Returns the remainder of the division of 2 numbers as a real number

Syntax

rem(x1, x2)

Arguments

Name	Description	Default	Range	Type	Required
x1	dividend	None	(-∞:∞)	Integer, real, complex	Yes
x2	divisor	1	(-∞:∞)	Integer, real, complex	No

Examples

```
remV=rem(100, 2.3) returns 1.1
remV=rem(1) returns 1
remCV=rem(1+j*0.2, 10) returns 1.02
```

See Also

fmod() (expsim)

sgn()

Returns the signum value as an integer, real or complex number

Syntax

sgn(x)

Arguments

Name	Description	Default	Range	Type	Required
x	Integer, real, complex value	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
sgnV=sgn(100) returns 1.0
sgnV=sgn(-0.3) returns -1.0
```

See Also

abs() (expsim), *exp()* (expsim), *int()* (expsim), *log10()* (expsim), *pow()* (expsim), *sqrt()* (expsim)

Note
The function name sgn() is used for more than one type of expression. For comparison, see the Measurement Expression *sgn() Measurement* (expmeas) and the AEL Function *sgn() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

sin()

Returns the sine as an integer, real or complex number.

Syntax

sin(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real or complex number	None	(-∞:∞)	Integer, real or complex	Yes

Examples

```
sinRV=sin(pi) returns 1.225e-16
sinRV=sin(-pi/2) returns -1
sinCV=sin(1+j*0.4) returns 0.936/13.71
```

See Also

cos() (expsim), *tan()* (expsim)

Note
The function name sin() is used for more than one type of expression. For comparison, see the Measurement Expression *sin() Measurement* (expmeas) and the AEL Function *sin() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

sinc()

Returns the sinc - sin(x)/x value of an integer or real number

Syntax

sinc(x)

Arguments

Name	Description	Default	Range	Type	Required
x	value to find sinc	None	(-∞:∞)	Integer, real	Yes

Examples

```
y=sinc(pi/2) returns 0.637
```

See Also

sin() (expsim)

Note
The function name sinc() is used for more than one type of expression. For comparison, see the Measurement Expression *sinc() Measurement* (expmeas) and the AEL Function *sinc() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

sinh()

Returns the hyperbolic sine as an integer, real or complex number

Syntax

sinh(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real, or complex number	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
sinhV=sinh(0.8) returns 0.888
sinhV=sinh(-0.5) returns -0.521
sinhCV=sinh(0.8+j*0.5) returns 1.238/29.037
```

See Also

cosh() (expsim), *tanh()* (expsim)

Note
The function name sinh() is used for more than one type of expression. For comparison, see the Measurement Expression *sinh() Measurement* (expmeas) and the AEL Function *sinh() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

spectrum()

Returns the spectrum as a complex array

Syntax

spectrum(function(0), NumPoints, Period, Delay, Window)

Arguments

Name	Description	Default	Range	Type	Required
function(0)	Function with a single argument	None	None	function	Yes
NumPoints	Number of points	None	[0:∞)	Integer, real	Yes
Period	Time step period	None	[0:∞)	Integer, real	Yes
Delay	Delay	0	[0:∞)	Integer, real	No
Window	Specifies if windowing is to be applied	0	[0:1]	Integer, Real	No

Examples

```
Creates a spectrum:
fCos(x) = cos_pulse(x, 0, 2, 0, 100ps, 100ps, 200ps, 400ps)
specV = spectrum(fCos(time), 128, 400ps)
```

sqrt()

Returns the square root as an integer, real or complex number

Syntax

sqrt(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real, complex number	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
sqrtV=sqrt(100) returns 10
sqrtCV=sqrt(0.1+j*0.3) returns 0.562/35.783
```

See Also

abs() (expsim), *exp()* (expsim), *int()* (expsim), *log10()* (expsim), *pow()* (expsim), *sgn()* (expsim)

Note
The function name sqrt() is used for more than one type of expression. For comparison, see the Measurement Expression *sqrt() Measurement* (expmeas) and the AEL Function *sqrt() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

sum()

Returns the sum of values in an array

Syntax

sum(x)

Arguments

Name	Description	Default	Range	Type	Required
x	array to sum	None	(-∞:∞)	Array of integer, real, or complex	Yes

Examples

```
sumV=sum(list(1,2,3)) returns 6
```

See Also

max() (expsim), *min()* (expsim)

Note
The function name sum() is used for more than one type of expression. For comparison, see the Simulator Expression *sum() Expression* (expsim). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

tan()

Returns the tangent as an integer, real or complex number.

Syntax

tan(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real or complex number	None	(-∞:∞)	Integer, real or complex	Yes

Examples

```
tanRV=tan(pi) returns 0
tanRV=tan(-pi/4) returns -1
tanCV=tan(0.5-j*0.5) returns 0.694/-54.396
```

See Also

cos() (expsim), *sin()* (expsim)

Note
The function name tan() is used for more than one type of expression. For comparison, see the Measurement Expression *tan() Measurement* (expmeas) and the AEL Function *tan() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

tanh()

Returns the hyperbolic tangent as an integer, real or complex number

Syntax

tanh(x)

Arguments

Name	Description	Default	Range	Type	Required
x	integer, real, or complex number	None	(-∞:∞)	Integer, real, complex	Yes

Examples

```
tanhV=tanh(0.8) returns 0.664
tanhV=tanh(-0.5) returns -0.462
tanhCV=tanh(0.8+j*0.5) returns 0.694/-35.604
```

See Also

cosh() (expsim), *sinh()* (expsim)

Note
The function name tanh() is used for more than one type of expression. For comparison, see the Measurement Expression *tanh() Measurement* (expmeas) and the AEL Function *tanh() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

wtodbm()

Converts Watts to dBm and returns a real or complex number

Syntax

dbmVal = wtodbm(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	Value in Watts	None	(-∞:∞)	Real, Complex	Yes

Examples

```
wtodbm01_V=wtodbm(0.01) returns 10
wtodbm1_V=wtodbm(1) returns 30
wtodbmC_V=wtodbm(complex(10,2)) returns 40.094/1.225
```

See Also

db() Expression (expsim), *dbm() Expression* (expsim), *dbmtoa()* (expsim), *dbmtov()* (expsim), *dbmtow() Expression* (expsim), *dbpolar()* (expsim), *dbwtow()* (expsim)

Notes/Equations

This function converts Watts to dBm using the formula below:

$$dbmVal = 30 + 10\log(\text{Value})$$

Note
The function name wtodbm() is used for more than one type of expression. For comparison, see the Measurement Expression *wtodbm() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

S-Parameter Analysis Functions for Simulator Expressions

This section describes the S-parameter functions in detail.

- *ripple() Expression* (expsim)
- *vswrpolar()* (expsim)

ripple()

Calculates the ripple as amplitude*sin(2.**PI**(variable-intercept)/period)

Syntax

y = ripple(amplitude, intercept, period, variable)

Arguments

Name	Description	Default	Range	Type	Required
amplitude	Amplitude of the ripple	None	(-∞:∞)	Integer, real	Yes
intercept	intercept of the ripple	None	(-∞:∞)	Integer, real	Yes
period	Period of the waveform	None	(0:∞)	Integer, real	Yes
variable	ripple variable	None	(-∞:∞)	Integer, real	Yes

Examples

```
y = ripple(0.1, 0, 10 MHz, freq)
yS21 = dbpolar(10 + ripple(0.1, 0, 10 MHz, freq), 0)
```

Notes/Equations

This function calculates the ripple as:

$$ripple = amplitude \times \sin\left(2\pi \times \frac{variable - intercept}{period}\right)$$

Note
The function name ripple() is used for more than one type of expression. For comparison, see the Measurement Expression *ripple() Measurement* (expmeas). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

vswrpolar()

(VSWR,angle)-to-rectangular conversion function

Syntax

vswrpolar(VSWR, Angle)

Arguments

Name	Description	Default	Range	Type	Required
VSWR	VSWR value	None	(-∞:∞)	Real	Yes
Angle	Angle to convert	None	[-360:360]	Real	Yes

Examples

```
vswr = 1.99
value = vswrpolar(vswr, 1.72) returns 0.333+j0.01
```

See Also

dbpolar() (expsim)

Notes/Equations

The vswrpolar() function converts the VSWR, angle to rectangular format. vswrpolar is calculated as follows:

$$\text{value} = \frac{|VSWR| - 1}{|VSWR| + 1}\cos\left(\frac{\theta\pi}{180}\right) + j\left(\frac{|VSWR| - 1}{|VSWR| + 1}\right)\sin\left(\frac{\theta\pi}{180}\right)$$

Transient Source Functions

There are several built-in functions that mimic SPICE transient sources. These functions are typically used with the *vt* parameter of the voltage source, and the *it* parameter of the current source. This section describes the transient source functions in detail. The functions are listed in alphabetical order.

- *bitseq()* (expsim)
- *cos pulse()* (expsim)
- *damped sin()* (expsim)
- *erf pulse()* (expsim)
- *exp pulse()* (expsim)
- *impulse()* (expsim)
- *lfsr()* (expsim)
- *pulse()* (expsim)
- *pwl()* (expsim)
- *pwlr()* (expsim)
- *ramp()* (expsim)
- *rect()* (expsim)
- *sffm()* (expsim)
- *step() Expression* (expsim)

bitseq()

Returns the bit sequence at specified time point as a real number

Syntax

bitseq(time, ClockFreq, Rise, Fall, Vlow, Vhigh, BitSeq)

Arguments

Name	Description	Default	Range	Type	Required
time	program variable time	None	[0:∞)	Real	Yes
ClockFreq	Clock frequency of the signal	Fstop †	(0:∞)	Real	No
Rise	Rise time of pulse	Tstep † †	[0:∞)	Real	No
Fall	Fall time of pulse	Tstep † †	[0:∞)	Real	No
Vlow	Minimum voltage level	0 V	(-∞:∞)	Real	No
Vhigh	Maximum voltage level	1 V	(-∞:∞)	Real	No
BitSeq	Bit sequence	"1101010100101"	[0:1] † † †	String	No

† Fstop is 1/Transient StopTime or 1/Envelope Stop
† † Tstep is Transient MaxTimeStep or Envelope Step
† † † sequence of 0s and 1s

Examples

This example assumes that a transient simulation is performed using:

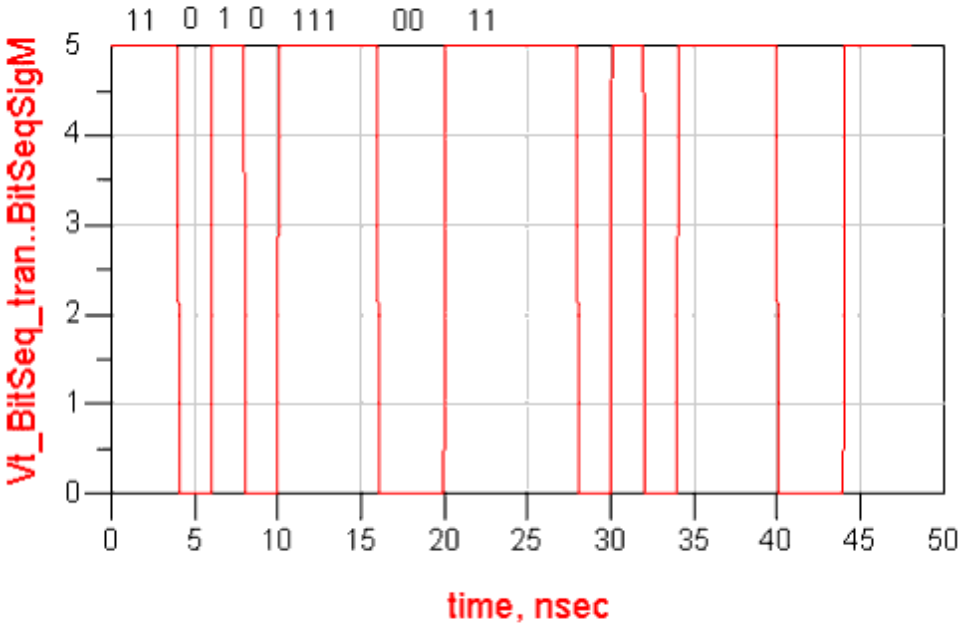
```
StartTime = 0, StopTime = 2*BitPeriod*NumBits,
MaxTimeStep = BitPeriod*NumBits/pow(2,NumBits)
where: BitRate = 500MHz, BitPeriod = 1/BitRate, NumBits = length(BitSeq)-1,
BitSeq = "110101110011"
This expression creates a bit sequence which repeats every 24 nsec:
value = bitseq(time, BitRate, 0.1nsec, 0.1nsec, 0, 5, BitSeq)
```

Notes/Equations

The bitseq() function can be used to vary the waveform of a pulse, an arbitrary bit pattern such a 110101110011. When the end of the sequence is reached, the sequence is repeated.

The transient stop time (Tstop) should be exactly one bit cycle for good results. For the example given above, `BitPeriod=1/BitRate = 1/500MHz = 2nsec` . For `BitSeq="110101110011", Tstop=NumBits * BitRate = 12bits * 2nsec = 24 nsec` .

Figure: The bitseq() Function



cos_pulse()

Returns the periodic cosine shaped pulse value at time, as a real number

Syntax

cos_pulse(time, Low, High, Delay, Rise, Fall, Width, Period)

Arguments

Name	Description	Default	Range	Type	Required
time	program time variable	None	[0:∞)	Real	Yes
Low	initial value	0	(-∞:∞)	Real	No
High	peak value	1	(-∞:∞)	Real	No
Delay	delay time	0	[0:∞)	Real	No
Rise	rise time	Tstep †	[0:∞)	Real	No
Fall	fall time	Tstep †	[0:∞)	Real	No
Width	pulse width	Tstop † †	(0:∞)	Real	No
Period	pulse period	Tstop † †	[Width+Rise+Fall:∞)	Real	No

† Where Tstep is Transient MaxTimeStep or Envelope Step
† † Where Tstop is StopTime or Envelope Stop

Examples

```
This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 3ns, MaxTimeStep = 50ps
```

1. This expression creates cosine-shaped pulses, repeating periodically, at every 400ps up to 3ns:
   ```
   Vcos = cos_pulse(time, 0, 2, 0, 100ps, 100ps, 200ps, 400ps)
   ```
2. This expression creates a single cosine-shaped pulse using default values for the arguments not listed in the function:
   ```
   Vcos_default = cos_pulse(time)
   Low=0, High=1, Delay=0, Rise=50ps, Fall=50ps, Width=3ns, Period=3ns
   ```

See Also

pulse() (expsim), *exp_pulse()* (expsim), *erf_pulse()* (expsim), *damped_sin()* (expsim), *pwl()* (expsim), *pwlr()* (expsim)

Notes/Equations

This expression can be used to create a current or voltage cosine-shaped pulse using the *ItUserDef* or *VtUserDef* time domain sources. Alternately, the built-in time domain current or voltage source component, *ItPulse* or *VtPulse* can be used and setting Edge=cosine.

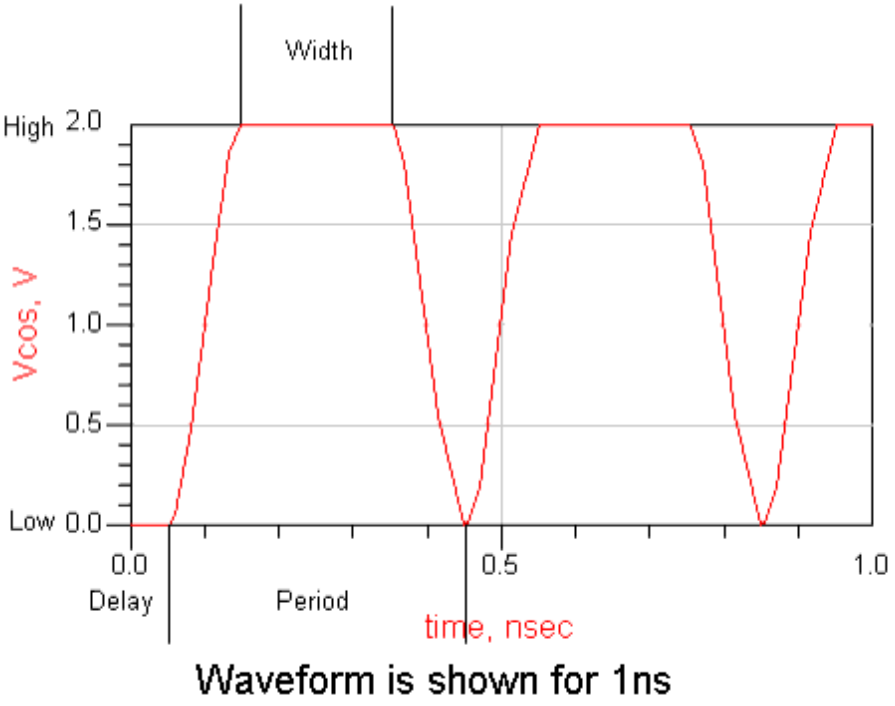
The slope change is not abrupt and its frequency spectrum decreases more rapidly. The rise and fall time define the total transition period and the maximum slope is greater than

(High-Low)/Rise.
The output value is calculated as follows:

Figure: The cos_pulse() Function

$$t = (t - Delay) - floor\left(\frac{(t - Delay)}{Period}\right) Period \qquad ((t - Delay) > Period)$$

$$value = \begin{cases} value = Low & (t < Rise) \\ 0.5\left(Low + High + \cos\left(\frac{\pi t}{Rise}\right)(Low - High)\right) & (t < Rise) \\ High & (t < Width) \\ 0.5\left(High + Low + \cos\left(\frac{\pi t}{Fall}\right)(High - Low)\right) & (t < Fall) \end{cases}$$



Waveform is shown for 1ns

damped_sin()

Returns the damped sin value at time point as a real number

Syntax

damped_sin(time, Offset, Amplitude, Freq, Delay, Damping, Phase)

Arguments

Name	Description	Default	Range	Type	Required
time	program time variable	None	(0:∞)	Real	Yes
Offset	initial offset	0.0	(-∞:∞)	Real	No
Amplitude	amplitude of sinusoidal wave	1.0	(-∞:∞)	Real	No
Freq	frequency of sinusoidal wave	Fstop †	(0:∞)	Real	No
Delay	time delay	0.0	[0:∞)	Real	No
Damping	damping factor in Hertz	0.0	(-∞:∞)	Real	No
Phase	initial phase value in degrees	0.0	(-∞:∞)	Real	No

† Fstop is 1/(Transient StopTime or Envelope Stop)

Examples

This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 3ns, MaxTimeStep = 50ps

1. This expression creates a damped sine pulse repeating periodically every 1ns:
   DsineValue = damped_sin(time, 0, 2, 1e9, 0.05, 0.5, 0)
2. This expression creates one damped sine pulse using default values for the arguments not listed in the function:
   DsineValue = damped_sin(time)
   Offset=0, Amplitude=1, Freq=1/3ns, Delay=0.0, Damping=0.0, Phase=0
   This example assumes that a transient simulation is performed using:
   StartTime = 0, StopTime = 10ns, MaxTimeStep = 10ps
   DsineValue=damped_sin(time, 0.5 V, 2 V, 1.2 GHz, 0.8 ns, 0.2 GHz, 15 _deg)
   The output of this example is shown in Example damped_sin() Output below.

See Also

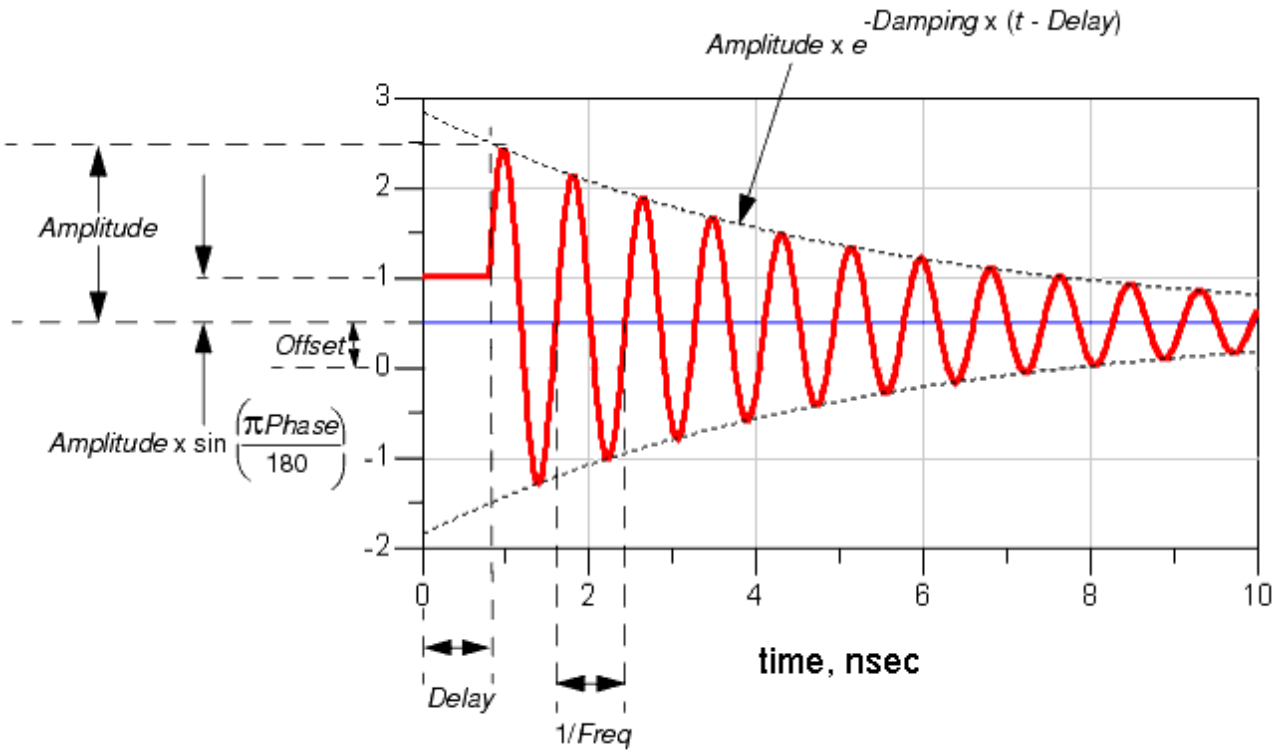
cos_pulse() (expsim), *erf_pulse()* (expsim), *exp_pulse()* (expsim), *pulse()* (expsim), *pwl()* (expsim), *pwlr()* (expsim), *ramp()* (expsim), *step()* (expsim)

Notes/Equations

This expression creates a time-periodic sinusoidal waveform at a specified frequency and phase, including turn-on characteristics. It can be used to create a current or voltage damped sinusoidal wave using the *ItUserDef* or *VtUserDef* time domain sources. Alternately, the built-in time domain current or voltage source component, *ItSine* or *VtSine* can be used.
The output is calculated as follows:

Figure: Example damped_sin() Output

$$Output = \begin{cases} Offset + Amplitude \times \sin\left(\frac{\pi Phase}{180}\right) & \text{, for (t-Delay)} < 0\,\text{sec} \\ Offset + Amplitude \times \sin\left(2\pi Freq \times (t - Delay) + \left(\frac{\pi Phase}{180}\right)\right) \times e^{-Damping \times (t - Delay)} & \text{, for (t-Delay)} >= 0\,\text{sec} \end{cases}$$



erf_pulse()

Returns the periodic error function shaped pulse at specified time point as a real number

Syntax

erf_pulse(time, Low, High, Delay, Rise, Fall, Width, Period)

Arguments

Name	Description	Default	Range	Type	Required
time	program time variable	None	[0:∞)	Real	Yes
Low	initial value	0	(-∞:∞)	Real	No
High	peak value	1	(-∞:∞)	Real	No
Delay	delay time	0	[0:∞)	Real	No
Rise	rise time	Tstep †	[0:∞)	Real	No
Fall	fall time	Tstep †	[0:∞)	Real	No
Width	pulse width	Tstop † †	(0:∞)	Real	No
Period	pulse period	Tstop † †	[Width+Rise+Fall:∞)	Real	No

† Where Tstep is Transient MaxTimeStep or Envelope Step
† † Where Tstop is StopTime or Envelope Stop

Examples

```
This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 3ns, MaxTimeStep = 50ps
```

1. ```
   This expression creates an error function repeating periodically at every 400ps:
   value = erf_pulse(time, 0, 2, 0, 100ps, 100ps, 200ps, 400ps)
   ```
2. ```
   This expression creates an error function using default values for the arguments not listed in the function:
   value = erf_pulse(time)
   Low=0, High=1, Delay=0, Rise=50ps, Fall=50ps, Width=3ns, Period=3ns
   ```

See Also

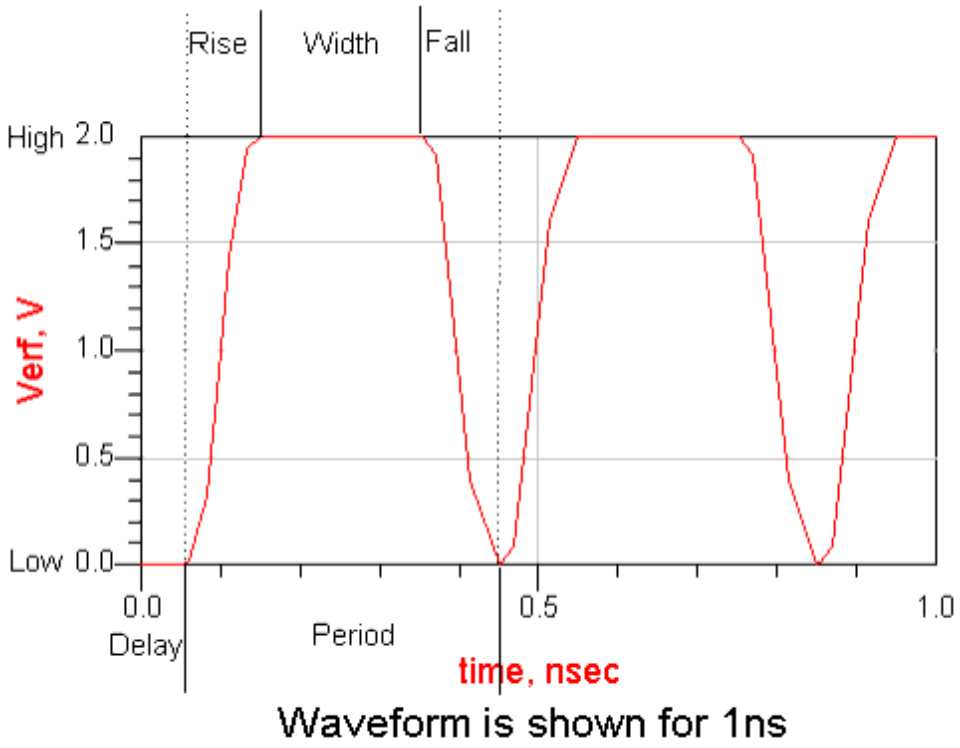
cos_pulse() (expsim), *damped_sin()* (expsim), *exp_pulse()* (expsim), *pulse()* (expsim), *pwl()* (expsim), *pwlr()* (expsim), *ramp()* (expsim), *step()* (expsim)

Notes/Equations

This function creates a time-periodic, error function shaped, rising and falling edged pulse train. For example it can be used to create a current or voltage error function shaped pulse using the *ItUserDef* or *VtUserDef* time domain sources. Alternately, the built-in time domain current or voltage source component, *ItPulse* or *VtPulse* can be used and setting Edge=erf. The slope change is not abrupt, and its frequency spectrum decreases more rapidly. The rise and fall time define the total transition period and the maximum slope is greater than (High-Low)/Rise.

Figure: The erf_pulse() Function

Figure: The erf_pulse() Function



exp_pulse()

Exponential pulse function

Syntax

exp_pulse(time, Low, High, Delay1, Tau1, Delay2, Tau2)

Arguments

Name	Description	Default	Range	Type	Required
time	program time variable	None	[0:∞)	Real	Yes
Low	initial value	0	(-∞:∞)	Real	No
High	peak value	1	(-∞:∞)	Real	No
Delay1	rise time delay	0	[0:∞)	Real	No
Tau1	rise time constant	Tstep †	[0:∞)	Real	No
Delay2	fall time delay	delay1 + Tstep †	[0:∞)	Real	No
Tau2	rise time constant	Tstep †	[0:∞)	Real	No

† Where Tstep is Transient MaxTimeStep or Envelope Step

Examples

```
This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 3ns, MaxTimeStep = 50ps
```

1. This expression creates an exponential pulse:
   `value = exp_pulse(time, 0, 2, 50ps, 50ps, 100ps, 50ps)`
2. This expression creates an exponential pulse using default values for the arguments not listed in the function:
   `value = exp_pulse(time)`
   `Low=0, High=1, Delay1=0, Tau1=50ps, Delay2=50ps, Tau2=50ps`

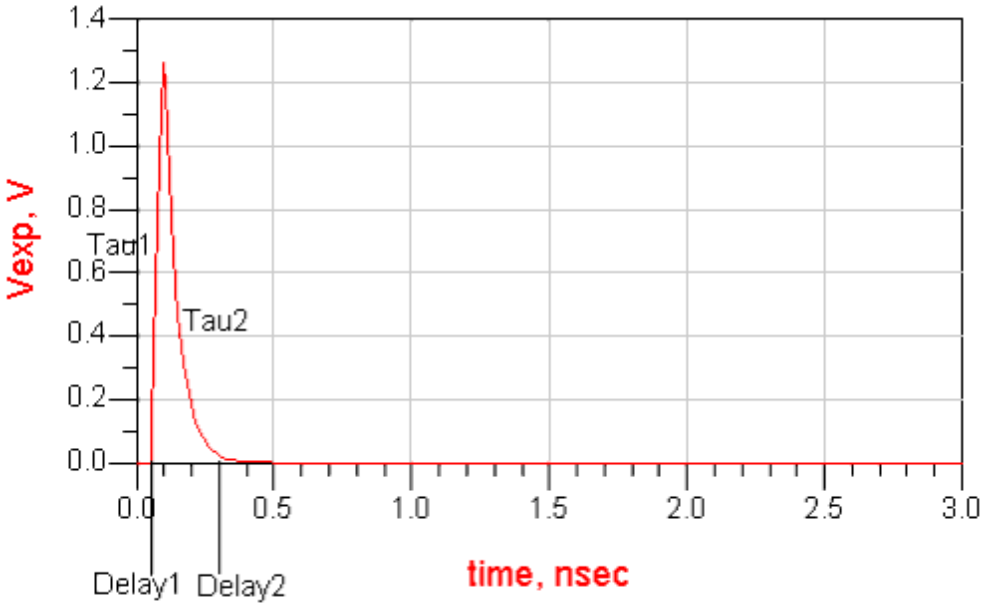
See Also

cos_pulse() (expsim), *damped_sin()* (expsim), *erf_pulse()* (expsim), *pulse()* (expsim), *pwl()* (expsim), *pwlr()* (expsim), *ramp()* (expsim), *step()* (expsim)

Notes/Equations

This expression can be used to create an exponential pulse. For example it can be used to create a current or voltage exponential decay signal using the *ItUserDef* or *VtUserDef* time domain sources. Alternately, the built-in time domain current or voltage source component, *ItExp* or *VtExp* can be used.
If Tau1 or Tau2 = 0, it is replaced by MaxTimeStep in transient simulation or Step in envelope simulation.

Figure: The exp_pulse() Function



impulse()

Computes the impulse response and returns a complex array

Syntax

impulse(CpxFunction(0), NumPoints, TimeStep, CnterFreq, Window)

Arguments

Name	Description	Default	Range	Type	Required
CpxFunction(0)	Function with a single argument	None	None	Function	Yes
NumPoints	number of points	None	[0:∞)	Integer, Real	Yes
TimeStep	Time Step	0.0	(0:∞)	Integer, Real	Yes
CenterFreq	Center frequency	0.0 †	[0:∞)	Integer, Real	No
Window	Specifies if window is to be applied	0 †	[0:1]	Integer, Real	No

† by default response is assumed to be baseband and no window applied.

Examples

```
Create a baseband response:
f(x) = exp(-j*2*pi*x*2ns)/2 + exp(-j*2*pi*x*5ns)/(1+j*2*pi*x*20ns)
imp = impulse(f(0), 128, 1ns)
Creates a RF response:
frect(x) = rect(x, 1GHz, 0.02/1ns) * exp(-j*2*pi*x*128ns)
imp = impulse(frect(0), 256, 1ns, 1GHz)
```

Notes/Equations

1. The impulse() function does nothing about potential non-causal responses. The function simply returns the entire inverse transformed waveform as being a long positive time only impulse.
2. The number of points is rounded up to the next power of 2, in order to apply an FFT.

lfsr()

Returns a string containing the complete sequence

Syntax

lfsr(Taps, Seed)

Arguments

Name	Description	Default	Range	Type	Required
Taps	Used to generate feedback.	None	[0:LARGEST_LONG_INTEGER] †	Integer, real	Yes
Seed	Initial value loaded into the shift regsiter.	None	[0:LARGEST_LONG_INTEGER] †	Integer, real	Yes

† LARGEST_LONG_INTEGER is 2147483647

Notes/Equations

For more information on lfsr, refer to the *VtLFSR_DT (Voltage Source, Pseudo-Random Pulse Train Defined at Discrete Time Steps)* in *Sources* found in the Components section of your documentation.

pulse()

Periodic pulse function

Syntax

pulse(time, Low, High, Delay, Rise, Fall, Width, Period)

Arguments

Name	Description	Default	Range	Type	Required
time	program time variable	None	[0:∞)	Real	Yes
Low	initial value	0	(-∞:∞)	Real	No
High	peak value	1	(-∞:∞)	Real	No
Delay	delay time	0	[0:∞)	Real	No
Rise	rise time	Tstep †	[0:∞)	Real	No
Fall	fall time	Tstep †	[0:∞)	Real	No
Width	pulse width	Tstop † †	(0:∞)	Real	No
Period	pulse period	Tstop † †	[Width+Rise+Fall:∞)	Real	No

† Where Tstep is Transient MaxTimeStep or Envelope Step
† † Where Tstop is StopTime or Envelope Stop

Examples

```
This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 3ns, MaxTimeStep = 50ps
```

1. This expression creates a pulse repeating periodically at every 400ps:
2. value = pulse(time, 0, 2, 50ps, 100ps, 100ps, 200ps, 400ps)
3. This expression creates a pulse using default values for the arguments not listed in the function:
   value = pulse(time)
   Low=0, High=1, Delay=0, Rise=50ps, Fall=50ps, Width=3ns, Period=3ns

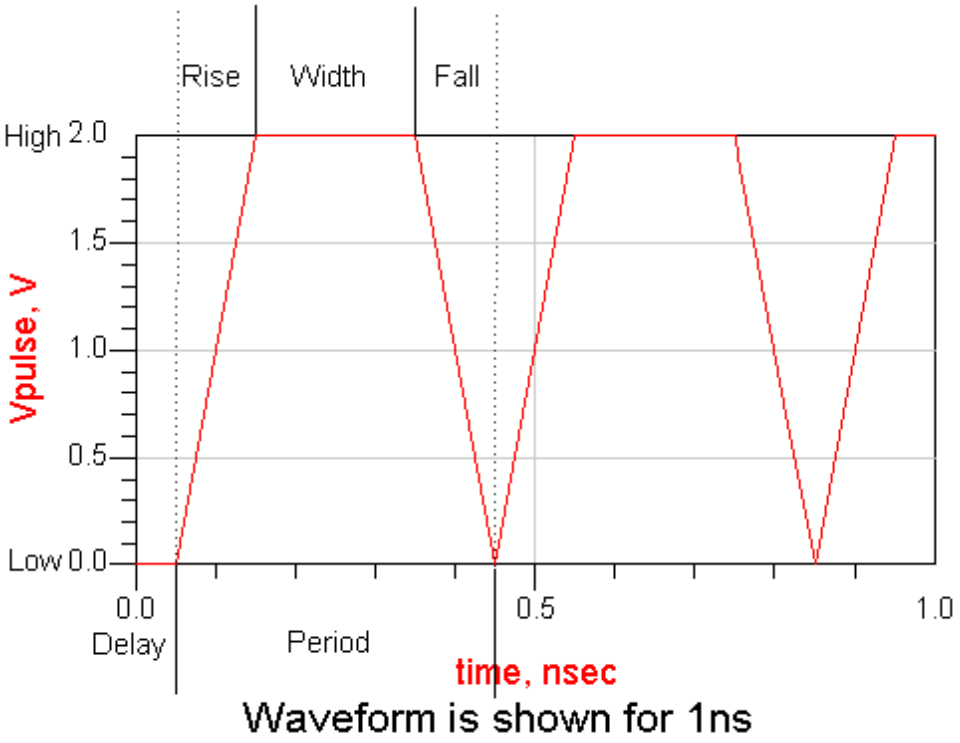
See Also

cos_pulse() (expsim), *damped_sin()* (expsim), *erf_pulse()* (expsim), *exp_pulse()* (expsim), *pwl()* (expsim), *pwlr()* (expsim), *ramp()* (expsim), *step()* (expsim)

Notes/Equations

This function creates a time-periodic linear ramp-shaped rising and falling edged pulse train. For example it can be used to create a current or voltage ramp-shaped pulse using the *ItUserDef* or *VtUserDef* time domain sources. Alternately, the built-in time domain current or voltage source component, *ItPulse* or *VtPulse* can be used and setting Edge=linear.

Figure: The pulse() function



pwl()

Piecewise-linear function

Syntax

pwl(time, T1, V1, T2, V2, ..., TN, VN)

Arguments

Name	Description	Default	Range	Type	Required
time	program time variable	None	(0:∞)	Real	Yes
T1, T2, ..., TN	time points	None	(0:∞)	Real	Yes
V1, V2, ..., VN	value at time points T1, T2, ..., TN	None	(-∞:∞)	Real	Yes

Examples

```
This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 100ns, MaxTimeStep = 50ps
This expression creates a single piecewise linear pulse:
value = pwl(time, 0ns, 0V, 5ns, 1V, 10ns, 2V, 20ns, 0V, 30ns, 1.5V)
```

See Also

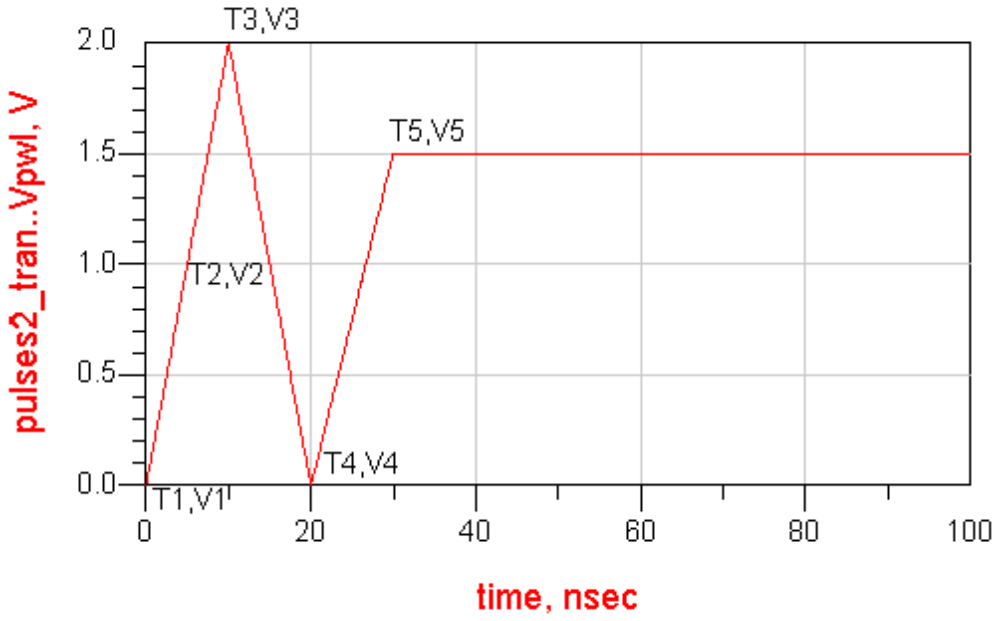
cos_pulse() (expsim), *damped_sin()* (expsim), *erf_pulse()* (expsim), *exp_pulse()* (expsim), *pulse()* (expsim), *pwlr()* (expsim), *ramp()* (expsim), *step()* (expsim)

Notes/Equations

This expression creates a time-periodic piecewise linear pulse train. It can be used to create a current or voltage pwl pulse using the *ItUserDef* or *VtUserDef* time domain sources. Alternately, the built-in time domain current or voltage source component, *ItPwl* or *VtPwl* can be used.
The piecewise linear values versus time points are specified as (Ti, Vi) pairs. A minimum of one time-value pair should be specified. Each pair specifies a value for a time=Ti. The intermediate values at time points that are not specified are interpolated.
The output is calculated as follows:

Figure: The pwl() Function

$$Value = \begin{cases} V_1 & (t = T_1) \\ V_1 + \left(\frac{t - T_1}{T_2 - T_1}\right) V_2 - V_1 & (T_1 < t < T_2) \\ V_n + \left(\frac{t - T_n}{T_{n+1} - T_n}\right) V_{n+1} - V_n & (T_n < t < T_{n+1}) \\ V_N & (t \geq T_N) \end{cases}$$



pwlr()

Piecewise-linear repeated function

Syntax

pwl(time, Ncycles, T1, V1, T2, V2, ..., Tn, Vn)

Arguments

Name	Description	Default	Range	Type	Required
time	program time variable	None	(0:∞)	Real	Yes
Ncycles	number of cycles that waveform is to be repeated	1	[1:∞)	Integer	Yes
T1, T2, ..., Tn †	time points	None	(0:∞)	Real	Yes
V1, V2, ..., Vn †	value at time points t1, t2, ..., tn	None	(-∞:∞)	Real	Yes

† a minimum of one time-value pair should be specified

Examples

```
This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 200ns, MaxTimeStep = 50ps
This expression creates a time-periodic piecewise linear pulse repeating every
30ns:
value = pwlr(time, 5, 0ns, 0V, 5ns, 1V, 10ns, 2V, 20ns, 0V, 30ns, 1.5V)
```

See Also

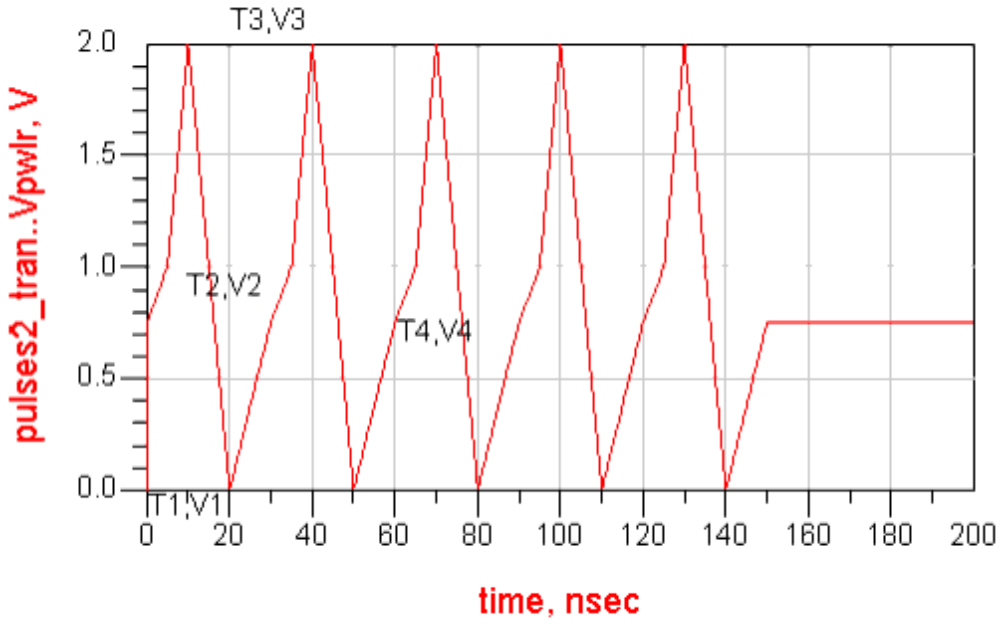
cos_pulse() (expsim), *damped_sin()* (expsim), *erf_pulse()* (expsim), *exp_pulse()* (expsim), *pulse()* (expsim), *pwl()* (expsim), *ramp()* (expsim), *step()* (expsim)

Notes/Equations

This expression creates a time-periodic piecewise linear pulse train, which is repeated for specified number of cycles. It can be used to create a current or voltage pwl pulse using the *ItUserDef* or *VtUserDef* time domain sources. Alternately, the built-in time domain current or voltage source component, *ItPwl* or *VtPwl* can be used.
The piecewise linear values versus time points are specified as (Ti, Vi) pairs. A minimum of one time-value pair should be specified. Each pair specifies a value for a time=Ti. The intermediate values at time points that are not specified are interpolated.
The output is calculated as follows:

Figure: The pwlr() Function

$$Value = \begin{cases} V_1 & (t = T_1) \\ V_1 + \left(\frac{t - T_1}{T_2 - T_1}\right) V_2 - V_1 & (T_1 < t < T_2) \\ V_n + \left(\frac{t - T_n}{T_{n+1} - T_n}\right) V_{n+1} - V_n & (T_n < t < T_{n+1}) \\ V_N & (t \geq T_N) \end{cases}$$



ramp()

Ramp function

Syntax

ramp(time)

Arguments

Name	Description	Default	Range	Type	Required
time	program variable time	0	[0:∞)	Real	Yes

Examples

```
The following example assumes that a transient simulation was done with
StartTime=0, StopTime=3ns and MaxTimeStep=50ps.
value=ramp(time)
produces a ramp pulse with value equal 0 at time equal 0, and time at time > 0
```

See Also

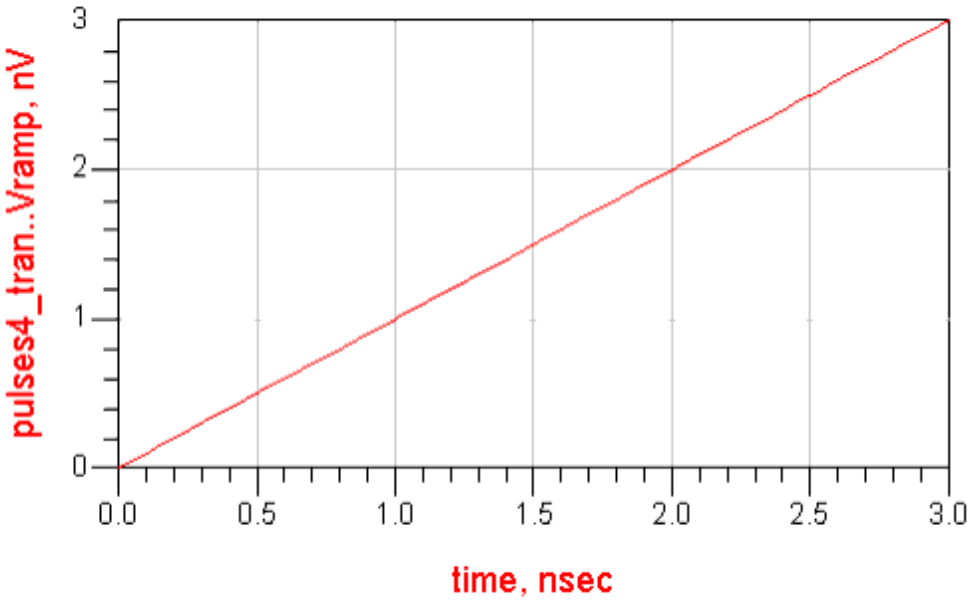
cos_pulse() (expsim), *damped_sin()* (expsim), *erf_pulse()* (expsim), *exp_pulse()* (expsim), *pulse()* (expsim), *pwl()* (expsim), *pwlr()* (expsim), *step()* (ael)

Notes/Equations

This expression creates a ramp function. It can be used to create a current or voltage ramp wave using the *ItUserDef* or *VtUserDef* time domain sources.
The output is calculated as follows:

$$\text{value} = \begin{cases} 0 & (t \le 0) \\ t & (t > 0) \end{cases}$$

Figure: The ramp() Function



rect()

Rectangular pulse function

Syntax

rect(x0, tc, td)

Arguments

Name	Description	Default	Range	Type	Required
x0	program time or freq variable	None	[0:∞)	Real	Yes
tc	center time	None	(-∞:∞)	Real	Yes
td	duration	None	(-∞:∞)	Real	Yes

Examples

```
This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 400ps, MaxTimeStep = 1ps
This expression produces a rectangular pulse:
value = rect(time, 100ps, 100ps)
```

See Also

cos_pulse() (expsim), *damped_sin()* (expsim), *erf_pulse()* (expsim), *exp_pulse()* (expsim), *pulse()* (expsim), *pwl()* (expsim), *pwlr()* (expsim), *ramp()* (expsim)

Notes/Equations

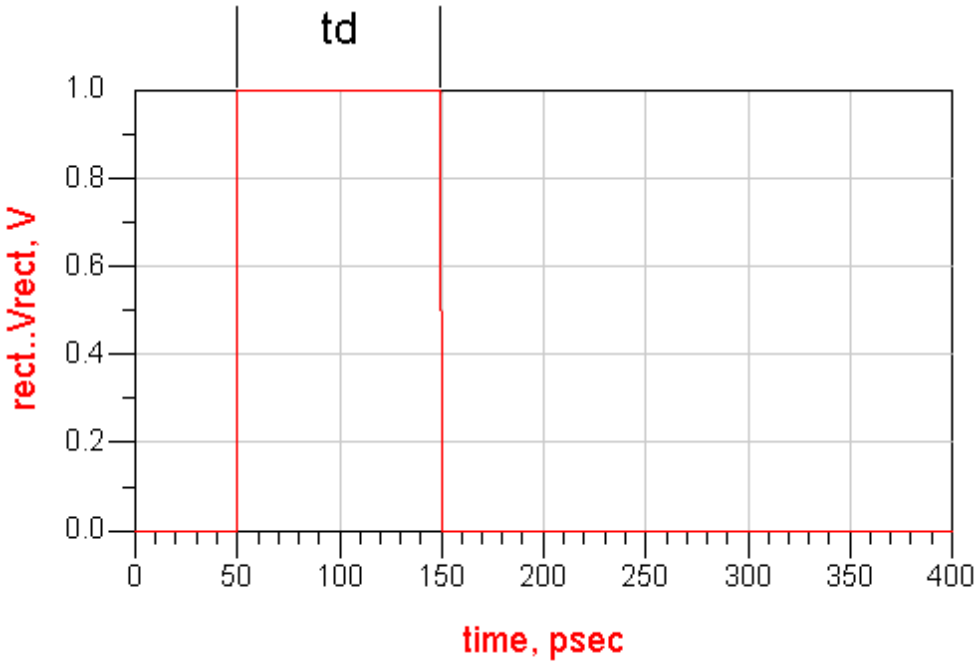
The function rect() creates a rectangular pulse of variable Time centered at time tc with duration td. It can be used to create a current or voltage rectangular pulse using the *ItUserDef* or *VtUserDef* time domain sources. Alternately it can be used in other expressions.
The output is calculated as follows:
For td > 0:

$$\text{value} = \begin{cases} 0 & (x0 > (tc + 0.5td) \ \text{OR} \ x0 < (tc - 0.5td)) \\ 0.5 & (x0 = (tc - 0.5td) \ \text{OR} \ x0 = (tc + 0.5td)) \\ 1 & Otherwise \end{cases}$$

For td< 0:

$$\text{value} = \begin{cases} 1 & (x0 < (tc + 0.5td) \ \text{OR} \ x0 > (tc - 0.5td)) \\ 0.5 & (x0 = (tc - 0.5td) \ \text{OR} \ x0 = (tc + 0.5td)) \\ 0 & Otherwise \end{cases}$$

Figure: The rect() Function



sffm()

Single frequency FM

Syntax

sffm(time, Offset, Amplitude, CarrierFreq, ModIndex, SignalFreq)

Arguments

Name	Description	Default	Range	Type	Required
time	Program variable time	None	[0:∞)	Real	Yes
Offset	Offset	0.0	[0:∞)	Real	No
Amplitude	Amplitude of signal	1.0	[0:∞)	Real	No
CarrierFreq	Carrier Frequency	1/Tstop †	[0:∞)	Real	No
ModIndex	Modulation Index	0.0	[0:∞)	Real	No
SignalFreq	Signal Frequency	1/Tstop †	[0:∞)	Real	No

† Where Tstop is StopTime or Envelope Stop

Examples

```
This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 3ns, MaxTimeStep = 50ps
```

1. This expression creates SFFM pulses repeating periodically at every 1ns until 3ns:
   ```
   Vsffm = sffm(time, 0, 2, 1GHz)
   ```
2. This expression creates a SFFM pulse using default values for the arguments not listed in the function:
   ```
   Vsffm_default = sffm(time)
   Offset=0, Amplitude=1, Carrier_freq=1/Tstop, Mod_Index=0.0,
   Signal_freq=1/Tstop
   ```

See Also

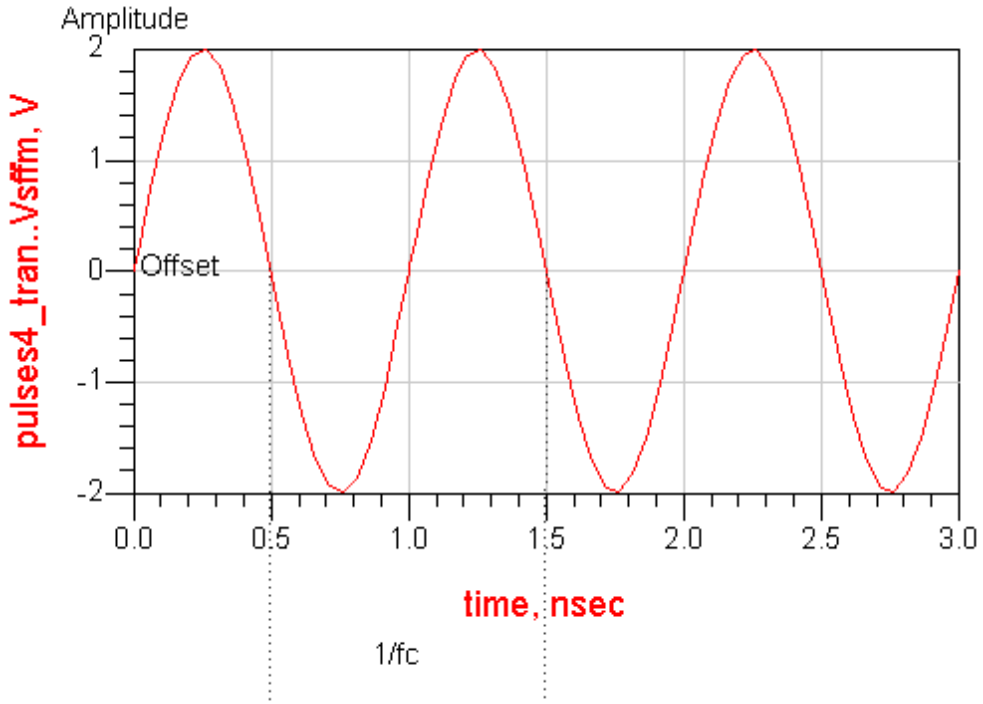
sffm() (expsim)

Notes/Equations

The sffm() function voltage value is calculated as follows:

$$\text{value} = \text{Offset} + \text{Amplitude}\sin((2\pi fc\ \text{time}) + \text{Mod_Index}\sin(2\pi fs\ \text{time}))$$

where *fc* is carrier frequency, and *fs* is signal frequency



step()

Step function

Syntax

step(time)

Arguments

Name	Description	Default	Range	Type	Required
t	program time variable	None	(-∞:∞)	Integer, real	Yes

Examples

```
This example assumes that a transient simulation is performed using:
StartTime = 0, StopTime = 2ns, MaxTimeStep = 50ps
This expression produces a step pulse with value = 0.5 at time = 0, 1 at time > 0:
value = step(time - tau)
where tau = 1.0ns
```

See Also

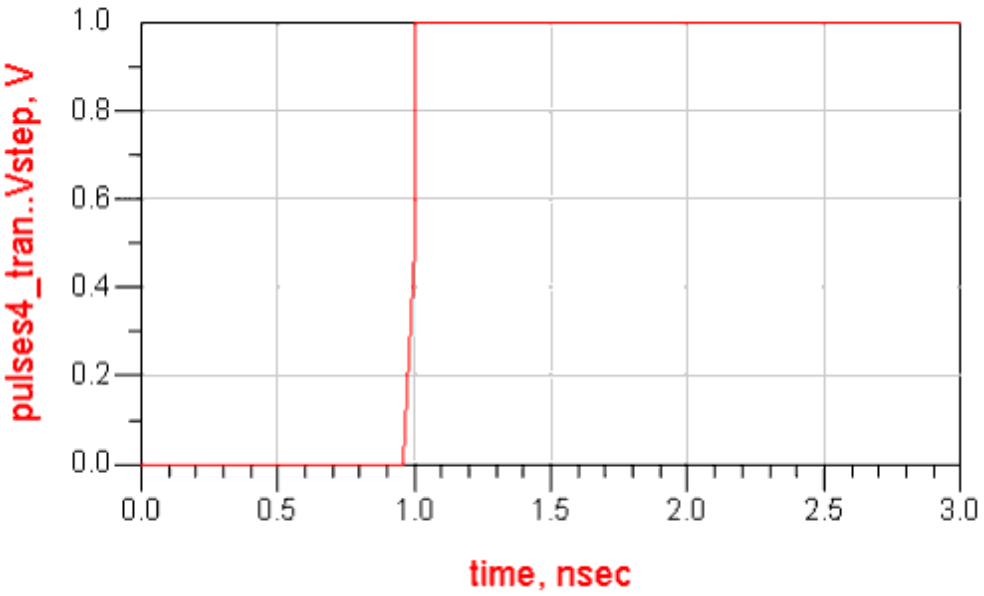
cos_pulse() (expsim), *damped_sin()* (expsim), *erf_pulse()* (expsim), *exp_pulse()* (expsim), *pulse()* (expsim), *pwl()* (expsim), *pwlr()* (expsim), *ramp()* (expsim)

Notes/Equations

This expression creates a step function. It can be used to create a current or voltage step wave using the *ItUserDef* or *VtUserDef* time domain sources.
The output is calculated as follows:

Figure: The step() Function

$$\text{value} = \begin{cases} 0 & (t < 0) \\ 1 & (t > 0) \\ 0.5 & t = 0 \end{cases}$$



Note
The function name step() is used for more than one type of expression. For comparison, see the Measurement Expression *step() Measurement* (expmeas) and the AEL Function *step() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

Utility Functions for Simulator Expressions

This section describes the utility functions in detail. Utility Functions are used for activities such as formatting, printing information to the status screen, etc. The functions are listed in alphabetical order.

- *echo()* (expsim)
- *gridSmithChart()* (expsim)
- *limit warn()* (expsim)
- *sprintf() Expression* (expsim)
- *strcat() Expression* (expsim)
- *value()* (expsim)

echo()

echo returns value of a string variable

Syntax

echo(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	Variable or value that is to be echoed	None	None	Variable or String	Yes

Examples

```
sVal=value(10.23) returns real number as a string "10.23"
echoVal=echo(sVal) returns "10.23"
installDir=echo("$HPEESOF_DIR") returns the product installation directory
```

See Also

value() (expsim), *sprintf()* (expsim)

Notes/Equations

The echo function is used to echo a string argument to the terminal and returns it as a value. It can specifically be used to echo System Environment variables such as *HOME*, *TMP*, *HPEESOF_DIR*, etc. The returned string text can then be printed in the data display. This function is similar to the Operating System echo function.in

gridSmithChart()

Generates list of points on Smith Chart based on selected grid type and additional isolated

points.

Syntax

gridPts = gridSmithChart(numGridPts, [gridCenter], [gridRadius], [gridType], [extraPts])

Arguments

Name	Description	Default	Range	Type	Required
numGridPts	Number of points covering primary grid area.	None	0:1000	Integer	Yes
gridCenter	Center of grid on Smith Chart.	0 + j*0	[0:1.0]	Complex	No
gridRadius	Radius of coverage on Smith Chart from origin.	1.0	[0:1.0]	Real	No
gridType	Type of grid coverage required.	"Uniform"	{"Uniform","Polar","Rectangular"}	String	No
extraPts	Additional points explictly required on grid.	None	[0:1.0]	Complex, Real, Integer	No

Examples

```
list1 = gridSmithChart(100) - lists 100 uniformly spaced points covering entire
Smith Chart.
list2 = gridSmithChart(100,0.20+j*0.15) - lists 100 uniformly spaced points
centered at (0.2,0.15)
list3 = gridSmithChart(100,,0.65) - lists 100 uniformly spaced points covering
the Smith Chart upto magnitude 0.65 from the origin.
list4 = gridSmithChart(100,,,"Polar") - lists 100 points in polar pattern
covering entire Smith Chart.
list5 = gridSmithChart(100,,,,makearray(2,polar(0.12,34)) - adds polar(0.12,34)
to the default grid.
```

limit_warn()

Limits the value to default, mininimum or maximum value, issues a warning and returns the limited value.

Syntax

limit_warn(Parameter, MinValue, MaxValue, Default, Name)

Arguments

Name	Description	Default	Range	Type	Required
Parameter	Parameter to be limited	None	(-∞:∞)	Integer, Real	No
MinValue	minimum value for the parameter	None	[-LargestReal:LargestReal] †	Integer, Real	No
MaxValue	maximum value for the parameter	LargestReal	[-LargestReal:LargestReal] †	Integer, Real	No
Default	default value for the parameter	-LargestReal	(-∞:∞)	Integer, Real	No
Name	name of parameter	None	None	String	No

† LargestReal = 1.79769313486231e+308

Examples

```
Let's assume that a circuit has a resistor R1. If the value of the resistor is
to be limited by a minimum value of 10 ohms, a maximum value of 49 ohms and a
default of 40 ohms, then it can be accomplished by setting the parameter R of
the resistor as:
Rval = 50
R = limit_warn(Rval, 10, 49, 30, "R1 value")
will display a warning "While evaluating expression `R1.R`: `R1 value limited
to 49`
```

Notes/Equations

This function is used to limit the value of a parameter to a default, minimum or maximum value. If the first parameter value is not set, then the value of the parameter is set to *Default* .
If *MinValue* is set, and if the value of the parameter is less than the MinValue, then the value is set to *MinValue* .
If *MaxValue* is set, and if the value of the parameter is greater than the MinValue, then the value is set to *MaxValue* .

sprintf()

Formatted print utility

Syntax

sprintf(Format, Variable)

Arguments

Name	Description	Default	Range	Type	Required
Format	Format of string in C language syntax	None	"%d,%f,%g,%e,%s,%c\\nt"	String Text	Yes
Variable	Variable that is to be formatted	None	None	Integer, Real, Complex, String or Integer/Real/Complex Array	Yes

Examples

```
iVal = 10
rVal = 10.23
cVal = 1+j*2
sVal = "one"
rA = makearray(1, 1, 2.1, 3)
cA = makearray(2, 1+j*1, 2+j*2, 3+j*3)

fmtI = sprintf("Integer value is %d", iVal) returns text "Integer value is 10"
fmtR = sprintf("Real value is %g", rVal) returns text "Real value is 10.23"
fmtC = sprintf("Complex value is (%g+j%g)", cVal) returns text "Complex value
is (1+j2)"
fmtS = sprintf("String value is %s", sVal) returns text "String value is one"
fmtr_rA = sprintf("%g", rA) returns text "1 2.1 3"
fmtr_cA = sprintf("%g+j%g", cA) returns text "1+j1 2+j2 3+j3"
```

See Also

value() (expsim)

Notes/Equations

1. The sprintf() function is used to format a Simulator Expression variable into string or text format. It can be used to format data of integer, real, complex, string, and array types. The returned string text can then be printed in the data display or output to the console using the *system* function with an *echo* command.
2. The sprintf() function is similar to the C function sprintf with certain restrictions. Only format %d, %f, %g, %e, and %s are supported. Only one variable can be formatted. For example, rVals=sprintf("%g %g", 10.1, 20.2) is not permitted.

Note
The function name sprintf() is used for more than one type of expression. For comparison, see the AEL Function *sprintf() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

strcat()

Concatenates a string or array and returns a string or an array

Syntax

strcat(A0,A1,A2, ...)

Arguments

Name	Description	Default	Range	Type	Required
A0,A1,A2,...	strings or arrays to be concatenated	None	None	String, Integer/Real/Complex array	Yes

Examples

```
iVal=10
rVal=10.23
cVal=1+j*2
sVal="one"
rA=makearray(1,1,2.1,3)
cA = makearray(2,1+j*1,2+j*2,3+j*3)
fmtI=sprintf("Integer value is %d", iVal) returns text "Integer value is 10"
fmtR=sprintf("Real value is %g", rVal) returns text "Real value is 10.23"
fmtC=sprintf("Complex value is (%g+j%g)", cVal) returns text "Complex value is
(1+j2)"
fmtS=sprintf("String value is %s", sVal) returns text "String value is one"
fmtr_rA=sprintf("%g ", rA) returns text "1 2.1 3"
fmtr_cA=sprintf("%g+j%g ", cA) returns text "1+j1 2+j2 3+j3"
```

Notes/Equations

1. The strcat() function is used to concatenate any number of strings, or arrays of the same type. The arguments must all be of the same type, i.e. if the first argument is a real array, the rest of the arguments to be concatenated must be real arrays as well.
2. The strcat() function cannot be used to concatenate string arrays.

Note
The function name strcat() is used for more than one type of expression. For comparison, see the the AEL Function *strcat() Function* (ael). Also, for more information on the different expression types and the contexts in which they are used, see *Duplicated Expression Names* (expmeas).

value()

Prints the value of a Simulation Expression variable representing an integer, real, complex as a string

Syntax

value(Value)

Arguments

Name	Description	Default	Range	Type	Required
Value	Value that is to be printed	None	None	Integer, Real, Complex, String, or Array †	Yes

† Supports integer, real and complex arrays. For string arrays use the sprintf() function.

Examples

```
rA = makearray(1, 1, 2, 3) returns an array (1, 2, 3)
value_rA = value(rA) returns "123"
cA = makearray(2, 1+j*1, 2+j*2, 3+j*3) returns a complex array (1+j*1, 2+j*2,
3+j*3)
value_cA = value(cA) returns (1,1)(2,2)(3,3)
```

See Also

echo() (expsim), *sprintf()* (expsim)

Notes/Equations

The value() function is used to format a Simulator Expression variable in text format. The returned string text can then be printed in the data display.