# Contents

**Preface** ........................................................................................................................................... 15

**Related Documents** ......................................................................................................................... 15

**Typographic and Syntax Conventions** .......................................................................................... 15
  - SKILL Syntax Examples ....................................................................................................................... 17

## 1

**Introduction to OCEAN** .................................................................................................................. 19

**Types of OCEAN Commands** ......................................................................................................... 20

**OCEAN Online Help** ....................................................................................................................... 20

**OCEAN Syntax Overview** .............................................................................................................. 21
  - Common SKILL Syntax Characters Used in OCEAN ..................................................................... 21
  - Parentheses .................................................................................................................................. 21
  - Quotation Marks ............................................................................................................................... 22
  - Single Quotation Marks .................................................................................................................... 23
  - Question Mark ................................................................................................................................. 24
  - Data Types Used in OCEAN .......................................................................................................... 24
  - OCEAN Return Values .................................................................................................................... 25
  - Design Variables in OCEAN .......................................................................................................... 26
  - outputs() in OCEAN ....................................................................................................................... 26

**Parametric Analysis** ....................................................................................................................... 27
  - Data Access Without Running a Simulation .................................................................................. 28

**Distributed Processing** .................................................................................................................. 28
  - Blocking and Nonblocking Modes ................................................................................................. 29

## 2

**Using OCEAN** ................................................................................................................................. 31

**OCEAN Use Models** ....................................................................................................................... 31

**Using OCEAN Interactively** ............................................................................................................ 32
  - Using OCEAN from a UNIX Shell ................................................................................................. 32
  - Using OCEAN from the CIW ........................................................................................................ 32
## OCEAN Reference

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interactive Session Demonstrating the OCEAN Use Model</td>
<td>33</td>
</tr>
<tr>
<td>Creating OCEAN Scripts</td>
<td>34</td>
</tr>
<tr>
<td>Creating Scripts Using Sample Script Files</td>
<td>34</td>
</tr>
<tr>
<td>Creating Scripts from the Affirma Analog Circuit Design Environment</td>
<td>34</td>
</tr>
<tr>
<td>Selectively Creating Scripts</td>
<td>34</td>
</tr>
<tr>
<td>Loading OCEAN Scripts</td>
<td>37</td>
</tr>
<tr>
<td>Running Multiple Simulators</td>
<td>38</td>
</tr>
<tr>
<td>OCEAN Tips</td>
<td>38</td>
</tr>
</tbody>
</table>

### 3

## Introduction to SKILL

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Advantages of SKILL</td>
<td>41</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>42</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>42</td>
</tr>
<tr>
<td>Scaling Factors</td>
<td>42</td>
</tr>
<tr>
<td>Relational and Logical Operators</td>
<td>43</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>44</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>44</td>
</tr>
<tr>
<td>SKILL Syntax</td>
<td>46</td>
</tr>
<tr>
<td>Special Characters</td>
<td>46</td>
</tr>
<tr>
<td>White Space</td>
<td>47</td>
</tr>
<tr>
<td>Comments</td>
<td>47</td>
</tr>
<tr>
<td>Role of Parentheses</td>
<td>48</td>
</tr>
<tr>
<td>Line Continuation</td>
<td>49</td>
</tr>
<tr>
<td>Arithmetic and Logical Expressions</td>
<td>49</td>
</tr>
<tr>
<td>Constants</td>
<td>49</td>
</tr>
<tr>
<td>Variables</td>
<td>50</td>
</tr>
</tbody>
</table>

### 4

## Working with SKILL

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skill Functions</td>
<td>53</td>
</tr>
<tr>
<td>Data Types</td>
<td>53</td>
</tr>
<tr>
<td>Numbers</td>
<td>54</td>
</tr>
<tr>
<td>Atoms</td>
<td>55</td>
</tr>
<tr>
<td>Constants and Variables</td>
<td>55</td>
</tr>
</tbody>
</table>
OCEAN Reference

Strings ................................................................. 55
Arrays ......................................................................... 56
  Allocating an Array of a Given Size ......................... 56
Concatenating Strings (Lists) ..................................... 56
  Comparing Strings .................................................. 57
Declaring a SKILL Function ...................................... 58
  Defining Function Parameters ................................. 59
  Defining Local Variables (let) .................................. 59
Skill Function Return Values ..................................... 60
Syntax Functions for Defining Functions ..................... 60
  procedure ............................................................ 60
  Terms and Definitions ............................................ 60

5
OCEAN Environment Commands .............................. 63
  setup ...................................................................... 64

6
Simulation Commands ............................................... 67
  ac ........................................................................ 69
  analysis ................................................................. 70
  appendPath ............................................................ 72
  createFinalNetlist .................................................. 73
  createNetlist .......................................................... 74
  dc ........................................................................ 75
  definitionFile ........................................................ 77
  delete .................................................................. 78
  design ................................................................. 80
  desVar ................................................................. 82
  envOption ............................................................... 83
  forcenode ............................................................. 84
  ic ........................................................................ 85
  includeFile ............................................................ 86
  modelFile ............................................................... 87
  nodeset ................................................................. 88
## 7 Data Access Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>noise</td>
<td>89</td>
</tr>
<tr>
<td>ocnDisplay</td>
<td>90</td>
</tr>
<tr>
<td>off</td>
<td>92</td>
</tr>
<tr>
<td>option</td>
<td>93</td>
</tr>
<tr>
<td>path</td>
<td>94</td>
</tr>
<tr>
<td>prependPath</td>
<td>95</td>
</tr>
<tr>
<td>restore</td>
<td>96</td>
</tr>
<tr>
<td>resultsDir</td>
<td>97</td>
</tr>
<tr>
<td>run</td>
<td>98</td>
</tr>
<tr>
<td>save</td>
<td>101</td>
</tr>
<tr>
<td>saveOption</td>
<td>103</td>
</tr>
<tr>
<td>simulator</td>
<td>105</td>
</tr>
<tr>
<td>stimulusFile</td>
<td>106</td>
</tr>
<tr>
<td>store</td>
<td>107</td>
</tr>
<tr>
<td>temp</td>
<td>108</td>
</tr>
<tr>
<td>tran</td>
<td>109</td>
</tr>
<tr>
<td>dataTypes</td>
<td>111</td>
</tr>
<tr>
<td>getData</td>
<td>113</td>
</tr>
<tr>
<td>i</td>
<td>114</td>
</tr>
<tr>
<td>noiseSummary</td>
<td>116</td>
</tr>
<tr>
<td>ocnHelp</td>
<td>118</td>
</tr>
<tr>
<td>ocnPrint</td>
<td>121</td>
</tr>
<tr>
<td>openResults</td>
<td>123</td>
</tr>
<tr>
<td>outputParams</td>
<td>125</td>
</tr>
<tr>
<td>outputs</td>
<td>127</td>
</tr>
<tr>
<td>phaseNoise</td>
<td>128</td>
</tr>
<tr>
<td>pv</td>
<td>129</td>
</tr>
<tr>
<td>report</td>
<td>130</td>
</tr>
<tr>
<td>resultParam</td>
<td>132</td>
</tr>
<tr>
<td>results</td>
<td>135</td>
</tr>
<tr>
<td>selectResult</td>
<td>137</td>
</tr>
<tr>
<td>sp</td>
<td>138</td>
</tr>
<tr>
<td>temp</td>
<td>140</td>
</tr>
</tbody>
</table>
8
Plotting Commands

sweepNames ............................................. 141
sweepValues ............................................ 142
v ........................................................... 143
vswr ....................................................... 145
zm .......................................................... 146
zref ........................................................ 147

addSubwindow ........................................... 151
addSubwindowTitle ...................................... 152
addTitle ................................................... 153
addWaveLabel ............................................ 154
addWindowLabel ......................................... 156
clearAll .................................................... 157
clearSubwindow .......................................... 158
currentSubwindow ....................................... 159
currentWindow ........................................... 160
dbCompressionPlot .................................... 161
deleteSubwindow ........................................ 162
deleteWaveform ......................................... 163
displayMode ............................................... 164
graphicsOff .............................................. 165
graphicsOn ................................................. 166
hardCopy .................................................. 167
hardCopyOptions ......................................... 168
ip3Plot ..................................................... 170
newWindow ............................................... 171
plot ......................................................... 172
plotStyle ................................................. 174
removeLabel ............................................. 175
xLimit ...................................................... 176
yLimit ...................................................... 177
# OCEAN Aliases

Predefined Functions and Waveform (Calculator) Functions

### Predefined Arithmetic Functions
- abs
- acos
- add1
- asin
- atan
- cos
- exp
- linRg
- log
- logRg
- max
- min
- mod
- random
- round
- sin
- sqrt
- srandom
- sub1
- tan

### Waveform (Calculator) Functions
- average
- b1f
- bandwidth
- clip
- compression
- compressionVRI
compressionVRICurves .................................................. 217
conjuate ................................................................. 219
complex ................................................................. 220
complexp ................................................................. 221
convolve ................................................................. 222
cPwrContour ............................................................ 224
cReflContour ............................................................ 226
cross ..................................................................... 228
db10 ...................................................................... 230
db20 ...................................................................... 231
dbm ....................................................................... 232
delay ..................................................................... 233
deriv ..................................................................... 236
dft ........................................................................ 237
flip ...................................................................... 239
tourEval ................................................................. 240
frequency ................................................................. 241
ga ....................................................................... 242
gac ....................................................................... 243
gainBwProd .............................................................. 245
gainMargin ............................................................... 247
gmax ..................................................................... 248
gmin ..................................................................... 249
gmsg ..................................................................... 250
gmux ..................................................................... 251
gp ....................................................................... 252
gpc ....................................................................... 253
groupDelay .............................................................. 255
gt ......................................................................... 256
harmonic ................................................................. 257
harmonicFreqList ....................................................... 259
harmonicList ............................................................ 261
iinteg ................................................................. 263
imag ................................................................. 264
integ ................................................................. 265
ipn ................................................................. 266
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipnVR1</td>
<td>269</td>
</tr>
<tr>
<td>ipnVR1Curves</td>
<td>272</td>
</tr>
<tr>
<td>kf</td>
<td>275</td>
</tr>
<tr>
<td>ln</td>
<td>276</td>
</tr>
<tr>
<td>log10</td>
<td>277</td>
</tr>
<tr>
<td>lsb</td>
<td>278</td>
</tr>
<tr>
<td>lshift</td>
<td>280</td>
</tr>
<tr>
<td>mag</td>
<td>281</td>
</tr>
<tr>
<td>nc</td>
<td>282</td>
</tr>
<tr>
<td>overshoot</td>
<td>284</td>
</tr>
<tr>
<td>peakToPeak</td>
<td>286</td>
</tr>
<tr>
<td>phase</td>
<td>287</td>
</tr>
<tr>
<td>phaseDeg</td>
<td>289</td>
</tr>
<tr>
<td>phaseDegUnwrapped</td>
<td>290</td>
</tr>
<tr>
<td>phaseMargin</td>
<td>291</td>
</tr>
<tr>
<td>phaseRad</td>
<td>293</td>
</tr>
<tr>
<td>phaseRadUnwrapped</td>
<td>294</td>
</tr>
<tr>
<td>pow</td>
<td>295</td>
</tr>
<tr>
<td>psd</td>
<td>297</td>
</tr>
<tr>
<td>psdb</td>
<td>301</td>
</tr>
<tr>
<td>real</td>
<td>305</td>
</tr>
<tr>
<td>riseTime</td>
<td>306</td>
</tr>
<tr>
<td>rms</td>
<td>308</td>
</tr>
<tr>
<td>rmsNoise</td>
<td>309</td>
</tr>
<tr>
<td>root</td>
<td>310</td>
</tr>
<tr>
<td>rshift</td>
<td>312</td>
</tr>
<tr>
<td>sample</td>
<td>313</td>
</tr>
<tr>
<td>settlingTime</td>
<td>315</td>
</tr>
<tr>
<td>slewRate</td>
<td>317</td>
</tr>
<tr>
<td>spectralPower</td>
<td>319</td>
</tr>
<tr>
<td>ssb</td>
<td>320</td>
</tr>
<tr>
<td>tangent</td>
<td>322</td>
</tr>
<tr>
<td>thd</td>
<td>323</td>
</tr>
<tr>
<td>value</td>
<td>325</td>
</tr>
<tr>
<td>xmax</td>
<td>327</td>
</tr>
<tr>
<td>xmin</td>
<td>329</td>
</tr>
</tbody>
</table>
### Advanced Analysis

**Parametric Analysis Commands**
- `paramAnalysis` .................................................. 336
- `paramRun` ......................................................... 340

**Corners Analysis Commands**
- `cornerDesVar` ..................................................... 342
- `cornerMeas` ......................................................... 343
- `cornerRun` .......................................................... 344
- `cornerRunTemp` .................................................... 346
- `residual` ............................................................ 347
- `selectProcess` ..................................................... 348

**Monte Carlo Analysis Commands**
- `correlationTable` .................................................. 349
- `dataFilter` .......................................................... 350
- `histogram` .......................................................... 352
- `iterVsValue` ......................................................... 354
- `monteCarlo` .......................................................... 355
- `monteCorrelate` ..................................................... 357
- `monteDisplay` ....................................................... 358
- `monteExpr` ........................................................... 359
- `monteOutputs` ....................................................... 360
- `monteResults` ....................................................... 361
- `monteRun` ............................................................ 362
- `monteSelectResults` ............................................... 364
- `scatterplot` ........................................................ 365
- `specLimits` ........................................................ 366
- `yield` ................................................................. 367

**Optimization Commands**
- `optimizeAlgoControl` .............................................. 369
- `optimizeGoal` ....................................................... 370
### OCEAN Distributed Processing Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleteJob</td>
<td>376</td>
</tr>
<tr>
<td>digitalHostMode</td>
<td>377</td>
</tr>
<tr>
<td>digitalHostName</td>
<td>378</td>
</tr>
<tr>
<td>hostMode</td>
<td>379</td>
</tr>
<tr>
<td>hostName</td>
<td>380</td>
</tr>
<tr>
<td>killJob</td>
<td>381</td>
</tr>
<tr>
<td>monitor</td>
<td>382</td>
</tr>
<tr>
<td>remoteDir</td>
<td>383</td>
</tr>
<tr>
<td>resumeJob</td>
<td>384</td>
</tr>
<tr>
<td>suspendJob</td>
<td>385</td>
</tr>
<tr>
<td>wait</td>
<td>386</td>
</tr>
</tbody>
</table>

### Language Constructs

<table>
<thead>
<tr>
<th>Construct</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>388</td>
</tr>
<tr>
<td>unless</td>
<td>390</td>
</tr>
<tr>
<td>when</td>
<td>391</td>
</tr>
<tr>
<td>for</td>
<td>392</td>
</tr>
<tr>
<td>foreach</td>
<td>394</td>
</tr>
<tr>
<td>while</td>
<td>396</td>
</tr>
<tr>
<td>case</td>
<td>397</td>
</tr>
<tr>
<td>cond</td>
<td>399</td>
</tr>
</tbody>
</table>

### File Commands and Functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>402</td>
</tr>
<tr>
<td>fscanf</td>
<td>403</td>
</tr>
<tr>
<td>gets</td>
<td>405</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
</tr>
<tr>
<td>infile</td>
<td>406</td>
</tr>
<tr>
<td>load</td>
<td>407</td>
</tr>
<tr>
<td>newline</td>
<td>409</td>
</tr>
<tr>
<td>outfile</td>
<td>410</td>
</tr>
<tr>
<td>printf</td>
<td>412</td>
</tr>
<tr>
<td>println</td>
<td>413</td>
</tr>
</tbody>
</table>

### A

**OCEAN 4.4.3 Issues** 415

- Mixed-Signal in OCEAN 4.4.3 415

**Index** 417
Preface

The preface discusses the following:

- Related Documents on page 15
- Typographic and Syntax Conventions on page 15

Related Documents

The Open Command Environment for Analysis (OCEAN) is based on the Cadence® SKILL programming language. The following manuals give you more information about the SKILL language and other related products.

- The *SKILL Language User Guide* describes how to use the SKILL language functions, the SKILL++ functions, and the SKILL++ object system (for object-oriented programming).
- The *SKILL Language Reference* provides descriptions, syntax, and examples for the SKILL and SKILL++ functions.
- The *SKILL++ Object System Functions Reference* provides descriptions, syntax, and examples for the object system functions.
- The *Affirma Analog Circuit Design Environment User Guide* explains how to design and simulate analog circuits.
- The *Affirma Mixed-Signal Circuit Design Environment User Guide* explains how to design and simulate mixed-signal circuits.
- The *Affirma AMS Distributed Processing Option User Guide* explains how to set up and run distributed processing for OCEAN and other Affirma™ analog circuit design environment applications.

Typographic and Syntax Conventions

This list describes the syntax conventions used for the Affirma analog circuit design environment SKILL functions.
literal

Nonitalic words indicate keywords that you must type literally. These keywords represent command (function, routine) or option names.

argument(z_argument)

Words in italics indicate user-defined arguments for which you must substitute a name or a value. (The characters before the underscore (_) in the word indicate the data types that this argument can take. Names are case sensitive. Do not type the underscore (z_) before your arguments.) For a listing of data types, see “Data Types Used in OCEAN” on page 24.

| Vertical bars (OR-bars) separate possible choices for a single argument. They take precedence over any other character.

[ ] Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices. You can choose one argument from the list.

{ } Braces are used with OR-bars and enclose a list of choices. You must choose one argument from the list.

... Three dots (...) indicate that you can repeat the previous argument. If you use them with brackets, you can specify zero or more arguments. If they are used without brackets, you must specify at least one argument, but you can specify more.

argument... Specify at least one, but more are possible.

[argument]... Specify zero or more.

,... A comma and three dots together indicate that if you specify more than one argument, you must separate those arguments by commas.

=> A right arrow precedes the possible values that a SKILL function can return. This character is represented by an equal sign and a greater than sign.

/ A slash separates the possible values that can be returned by a SKILL function.
<yourSimulator>

Angle brackets indicate places where you need to insert the name of your simulator. Do not include the angle brackets when you insert the simulator name.

Important

The characters included in the list above are the only characters that are not typed literally. All other characters in the SKILL language are required and must be typed literally.

SKILL Syntax Examples

The following examples show typical syntax characters used in the SKILL language.

Example 1

```
list( g_arg1 [g_arg2] ... )
=> l_result
```

Example 1 illustrates the following syntax characters.

- `list` Plain type indicates words that you must type literally.
- `g_arg1` Words in italics indicate arguments for which you must substitute a name or a value.
- `()` Parentheses separate names of functions from their arguments.
- `_` An underscore separates an argument type (left) from an argument name (right).
- `[ ]` Brackets indicate that the enclosed argument is optional.
- `=>` A right arrow points to the return values of the function. Also used in code examples in SKILL manuals.
- `...` Three dots indicate that the preceding item can appear any number of times.
Example 2

needNCells(
    s_cellType | st_userType
    x_cellCount
) => t/nil

Example 2 illustrates two additional syntax characters.

| Vertical bars separate a choice of required options.

/ Slashes separate possible return values.
Introduction to OCEAN

This chapter provides an introduction to Open Command Environment for Analysis (OCEAN). In this chapter, you can find information about

- **Types of OCEAN Commands** on page 20
- **OCEAN Online Help** on page 20
- **OCEAN Syntax Overview** on page 21
- **Parametric Analysis** on page 27
- **Distributed Processing** on page 28

OCEAN lets you set up, simulate, and analyze circuit data. OCEAN is a text-based process that you can run from a UNIX shell or from the Command Interpreter Window (CIW). You can type OCEAN commands in an interactive session, or you can create scripts containing your commands, then load those scripts into OCEAN. OCEAN can be used with any simulator integrated into the Affirma™ analog circuit design environment.

Typically, you use the Affirma analog circuit design environment when creating your circuit (in Composer) and when interactively debugging the circuit. After the circuit has the performance you want, you can use OCEAN to run your scripts and test the circuit under a variety of conditions. After making changes to your circuit, you can easily rerun your scripts. OCEAN lets you

- Create scripts that you can run repeatedly to verify circuit performance
- Run longer analyses such as parametric analyses, Corners Analyses, and statistical analyses more effectively
- Run long simulations in OCEAN without starting the Affirma analog circuit design environment graphical user interface
- Run simulations from a nongraphic, remote terminal
Types of OCEAN Commands

You can create OCEAN scripts to accomplish the full suite of simulation and data access tasks that you can perform in the Affirma analog circuit design environment. An OCEAN script can contain three types of commands, as shown in the following figure.

![Diagram showing OCEAN Commands and Purpose]

- **Simulation Set-up Commands**
  - Specify the analyses to be run
  - Specify the nets and currents to save
  - Specify the simulator option values
  - Specify the circuit stimulus

- **Simulator Run Command**
  - Run the simulator

- **Data Access Commands**
  - Perform calculations on the results
  - Print information
  - Plot waveforms

All the parameter storage format (PSF) information created by the simulator is accessible through the OCEAN data access commands. (The data access commands include all of the Affirma analog circuit design environment calculator functions.)

**OCEAN Online Help**

Online help is available for all the OCEAN commands when you are in an OCEAN session. To get help for a specific OCEAN command, type the following:

```
ocnHelp('commandName')
```

This command returns an explanation of the command and examples of how the command can be used.

To get a listing of all the different types of commands in OCEAN, type the following:
ocnHelp()

For more information, see “ocnHelp” on page 121.

**OCEAN Syntax Overview**

OCEAN is based on the Cadence® SKILL programming language and uses SKILL syntax. All the SKILL language commands can be used in OCEAN. This includes if statements, case statements, for loops, while loops, read commands, print commands, and so on.

The most commonly used SKILL commands are documented in this manual. However, you are not limited to these commands. You can use any SKILL routine from any SKILL manual.

**Common SKILL Syntax Characters Used in OCEAN**

This section provides an overview of some basic SKILL syntax concepts that you need to understand to use OCEAN. For more information about SKILL syntax, see Chapter 3, “Introduction to SKILL.”

**Parentheses**

Parentheses surround the arguments to the command. The command name is followed immediately by the left parenthesis, with no intervening space.

**Examples**

The following example shows parentheses correctly enclosing two arguments to the path command.

```plaintext
path( "~/simulation1/schematic/psf" "~/simulation2/schematic/psf" )
```

In the next example, the space after the command name causes a syntax error.

```
Syntax error.
path ( "~/simulation1/schematic/psf" "~/simulation2/schematic/psf" )
```
Quotation Marks

Quotation Marks are used to surround string values. A string value is a sequence of characters, such as "abc".

In the following example, the directory names provided to the path command are strings, which must be surrounded by quotation marks.

```
path( "~/simulation1/schematic/psf" "~/simulation2/schematic/psf" )
```

Convention

In this manual, a SKILL convention is used to let you know when an argument must be a string. When you see the prefix t_, you must substitute a string value (surrounded by quotation marks) for the argument. Consider the following syntax statement:

```
desVar( t_desVar1 g_value1 t_desVar2 g_value2 )
```

In this case, there are two string values that must be supplied: t_desVar1 and t_desVar2. (The g_ prefix indicates a different type of argument. For more information about prefixes, see Chapter 4, "Working with SKILL."

Recovering from an Omitted Quotation Mark

Accidentally omitting a closing quotation mark from an OCEAN command can cause great confusion. For example, typing the incorrect command

```
strcat( "rain" "bow " )
```

appears to hang OCEAN. In an attempt to recover, you type a Control-c. That gives you a prompt but it does not fix the problem, as you discover when you then type the correct command.

```
strcat( "rain" "bow" )
```

Again, you have to type a Control-c and OCEAN responds with another message.

^C*Error* parser: interrupted while reading input

If you find yourself in this situation, do not press a Control-c. Instead, recover by entering a quotation mark followed by a right square bracket ( ] ). This procedure reestablishes a normal OCEAN environment and you can then reenter the correct command.

```
ocean> strcat( "rain" "bow " ] "rainbow ) "
```
ocean> strcat( "rain" "bow" )
"rainbow"
ocean>

**Single Quotation Marks**

The single quotation mark indicates that an item is a symbol. Symbols in SKILL correspond to constant enums in C. In the context of OCEAN, there are predefined symbols. The simulator that you use also has predefined symbols. When using symbols in OCEAN, you must use these predefined symbols.

**Examples**

In the following example, `tran` is a symbol and must be preceded by a single quotation mark. The symbol `tran` is predefined. You can determine what the valid symbols for a command are by checking the valid values for the command's arguments. For example, if you refer to "analysis" on page 70, you see that the valid values for the first argument include `'tran`.

```c
analysis( 'tran ... )
```

The list of items you can save with the `save` command is also predefined. You must choose from this predefined list. See "save" on page 101 and refer to the valid values for the `s_saveType` argument. The `v` symbol indicates that the item to be saved is the voltage on a net.

```c
save( 'v "net1" )
```

**Convention**

In this manual, a SKILL convention is used to let you know when an argument must be a symbol. When you see the prefix `s_`, you must substitute a symbol (preceded by a single quotation mark) for the argument. Consider the following syntax statement:

```c
selectResults( s_resultsName ) => t/nil
```

In this case, there is one symbol that must be supplied: `s_resultsName`. For the `selectResults` command, there is a different mechanism that lets you know the list of predefined symbols. If you type the following command, with no arguments, the list of predefined symbols is returned:

```c
results() => ( dc tran ac )
```

**Note:** Depending on which results are selected, the values returned by the `results` command vary.
Question Mark

The question mark indicates an optional keyword argument, which is the first part of a keyword parameter. A keyword parameter has two components:

- The first component is the keyword, which has a question mark in front of it.
- The second component is the value being passed, which immediately follows the keyword.

Keyword parameters, composed of these keyword/value pairs, are always optional.

Examples

In the following example, all the arguments to the `analysis` command except `’tran` are keyword/value pairs and are optional.

```
Keyword   Value passed

analysis( ’tran ?start 0 ?stop 1u ?step 1n )
```

For example, you can use `?center` and `?span` instead of `?start` and `?stop`. You also can omit `?start` altogether because it is an optional argument.

Convention

In this manual, a SKILL convention is used to let you know when arguments are optional. Optional arguments are surrounded by square brackets `[ ]`. In the following example, all of the keyword/value pairs are surrounded by square brackets, indicating that they are optional.

```
report( [?output t_filename | p_port] [?type t_type] [?name t_name]
        [?param t_param] [?format s_reportStyle] ) => t/nil
```

Data Types Used in OCEAN

The following table shows the internal names and prefixes for the SKILL data types that are used in OCEAN commands.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Internal Name</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating-point number</td>
<td>flonum</td>
<td>f</td>
</tr>
</tbody>
</table>
OCEAN Reference
Introduction to OCEAN

Data Type | Internal Name | Prefix
---|---|---
any data type | general | g
linked list | list | l
integer, floating-point number, or complex number | n
user-defined type | o
I/O port | port | p
symbol | symbol | s
symbol or character string | S
character string (text) | string | t
window type | w
integer number | fixnum | x

For more information about SKILL datatypes, see Chapter 4, “Working with SKILL.”

OCEAN Return Values

You get return values from most OCEAN commands and can use these values in other OCEAN commands.

The following table shows some examples in which the return value from a command is assigned to a variable.

<table>
<thead>
<tr>
<th>Assigning a Return Value to a Variable</th>
<th>Resulting Value for the Variable</th>
</tr>
</thead>
</table>
a=desVar("r1" 1k) | a=1k |
a=desVar("r1" 1k "r2" 2k) | a=2k |
a=desVar("r1") | a=1k, assuming r1 was set in a desVar command |
a=desVar("r2") | a=2k, assuming r2 was set in a desVar command |
Design Variables in OCEAN

Design variables in OCEAN function as they do in the Affirma analog circuit design environment. Design variables are not assigned in the order specified. Rather, they are reordered and then assigned. Consider the following example:

\[
\text{desVar( "a" "b+1" )}
\]
\[
\text{desVar( "b" 1 )}
\]

You might expect an error because \(a\) is assigned the value \(b+1\) before \(b\) is assigned a value. However, OCEAN reorders the statements and sends them as follows:

\[
\text{desVar( "b" 1 )}
\]
\[
\text{desVar( "a" "b+1" )}
\]

After the reordering, there is no error. (\(b\) is equal to 1 and \(a\) is equal to 2.)

Suppose you run a simulation, then specify the following:

\[
\text{desVar("b" 2)}
\]

You might expect \(a\) to be equal to 2, which was the last value specified. Instead, \(a\) is reevaluated to \(b+1\) or 3.

This approach is similar to how the design variables are used in simulation. For example, consider a circuit that has the following resistor:

\[
\text{R1 1 0 resistor r=b}
\]

If you change the value of \(b\), you expect the value of \(\text{R1}\) to change. You do not expect to have to netlist again or retype the \(\text{R1}\) instantiation.

This approach is used in the Affirma analog circuit design environment. Users are not expected to enter design variables in a particular order. Rather, the design variables are gathered during the design variable search then reordered before they are used.

outputs() in OCEAN

Throughout this manual are examples of nets and instances preceded by a “/” as well as examples without the “/”. There is a significant difference between the two.

If you create a design in the Affirma analog circuit design environment and save the OCEAN file, all net and instance names will be preceded with a “/”, indicating they are schematic names. The netlist/amap directory must be available to map these schematic names to names the simulator understands. (If your design command points to the raw netlist in the netlist directory, the amap directory is there.)
If you create a design or an OCEAN script by hand, or move the raw netlist from the netlist directory, the net and instance names might not be preceded with “/”. This indicates that simulator names are used, and mapping is not necessary.

If you are unsure whether schematic names or simulator names are used, after selectResult( S_resultsName ), type outputs() to see the list of net and instance names.

**Note:** Although you can move the raw netlist file from the netlist directory, it is not advised. There are other files in the netlist directory that are now required to run OCEAN.

### Parametric Analysis

There are two ways you can run parametric analyses in OCEAN:

- You can use the `paramAnalysis` command (recommended approach).
- You can use a SKILL for loop.

Using the `paramAnalysis` command is an easier approach. With this command, you can set up any number of nested parametric analyses in an OCEAN script. The `paramRun` command runs all the parametric analyses. When the analysis is complete, the data can be plotted as a family of curves. The following example shows how you might use nested parametric analyses:

```plaintext
paramAnalysis( "rl" ?start 200 ?stop 600 ?step 200
               paramAnalysis( 'rs ?start 300 ?stop 700 ?step 200
               )

) paramRun ()
```

In this example, the outer loop cycles through `rl`, and the inner loop cycles through `rs` as follows:

- Loop through `rl` from 200 to 600 by 200.
- Loop through `rs` from 300 to 700 by 200.
- Run.
- End the first loop.
- End the second loop.
So, for \( r_1=200 \), \( r_s \) equals 300, 500 and then 700. Then, for \( r_1=400 \), \( r_s \) equals 300, 500, and then 700. Finally, for \( r_1=600 \), \( r_s \) equals 300, 500, and then 700.

Use a SKILL for loop only if the `paramAnalysis` command is not adequate. You can use the SKILL for loop to set up any number of variable-switching runs. After all the simulations are complete, you have to work with the results directories individually. The following example shows how you might use SKILL loops for parametric analyses.

```plaintext
Cload = list( 2p 4p 6p 8p )
foreach( val Cload
desVar( "Cload" val )
a = resultsDir( sprintf( nil "./demo/Cload=%g" val ) )
printf( "%L", a )
run()
)
foreach( val Cload
openResults( sprintf( nil "./<dir>/Cload=%g" val ) )
selectResults( 'ac')
plot( vdb( "vout" ) )
)
```

### Data Access Without Running a Simulation

You can retrieve and use data from previous simulations at any time by opening the data with the `openResults` command. After opening the data, you can use any data access commands on this data. For more information, see Chapter 7, “Data Access Commands.”

You can use query commands such as `results`, `outputs`, and `dataTypes` to see what data is available to be opened.

### Distributed Processing

You can use OCEAN distributed processing commands to run simulations across a collection of computer systems. The distributed processing commands allow you to specify where and when jobs are run and allow you to monitor and control jobs in a variety of ways. Using distributed commands, you can

- Submit one or more jobs to a distributed processing queue
- Specify a host or group of hosts on which to distribute jobs
- View the status of jobs
- Specify when a job will run or in what sequence a group of jobs will run
Suspend and resume jobs

Cancel jobs

For you to be able to use the distributed processing commands, your site administrator needs to set up the lists of machines to which jobs are submitted. Each list of machines is a group of hosts and is called a queue. Consult the Affirma AMS Distributed Processing Option User Guide for more information on how to configure systems for distributed processing. For information on the distributed processing commands for OCEAN, see Chapter 12, “OCEAN Distributed Processing Commands.”

Blocking and Nonblocking Modes

You can configure jobs to run in blocking or nonblocking mode. In blocking mode, execution of subsequent OCEAN commands is halted until the job completes. In nonblocking mode, the system does not wait for the first job to finish before starting subsequent jobs.

Blocking Mode

You must run jobs in blocking mode to be able to use the data resulting from a job in a subsequent command in an OCEAN script or batch run.

For example, if you want to run a simulation, select the tran results from that simulation, and then plot them, you

1. Configure the simulation with setup commands
2. Run the simulation with the run() command
3. Select the desired results with the selectResults( ‘tran) command
4. Plot the results with the plot() command

A job like the one in the example above must run in blocking mode so that the commands are processed sequentially. If the jobs in the example above are run in nonblocking mode, the selectResult command starts before the run command can return any data, and the selectResult command and the plot command cannot complete successfully.

Nonblocking Mode

If you are submitting several jobs that have no interdependencies, you can run them concurrently when hostmode is set to distributed.
For example, if you want to run two separate simulations as jobs, but do not want to wait until the first is complete before starting the second, you

1. Configure the first simulation with setup commands
2. Configure a second simulation with setup commands

In the example above, the script starts the first job and then starts the second job without waiting for the first job to finish.

If you are running several commands, and some of them are data access commands, you can use the `wait` command to block a single job. The `wait` command is needed between the simulation and the data access commands to ensure the desired simulation is complete before the data is accessed.

For example, if you want to run two separate simulations as jobs (`sim1` and `sim2`), and want to select and plot the results of the second simulation run, you

1. Configure the first simulation with setup commands
2. Run the simulation with a `run(?jobPrefix "sim1")` command
3. Configure a second simulation with setup commands
4. Run the second simulation with the `run(?jobPrefix "sim2")` command
5. Cause the script to wait until the second simulation finishes before starting the `selectResults` command with the `wait(sim2)` command
6. Select the desired results with the `selectResults(’tran)` command
7. Plot the results with the `plot( )` command

In the example above, the script starts the first job and then starts the second job without waiting for the first job to finish. When the script reaches the `wait` command, it pauses until the second simulation runs and then selects the results to plot.
Using OCEAN

This chapter explains the different ways you can use OCEAN to perform simulation tasks. In this chapter, you can find information about

- **OCEAN Use Models** on page 31
- **Using OCEAN Interactively** on page 32
- **Creating OCEAN Scripts** on page 34
- **Running Multiple Simulators** on page 38
- **OCEAN Tips** on page 38

OCEAN Use Models

There are two ways you can use OCEAN:

- You can use OCEAN interactively to perform simple tasks.
- You can use OCEAN in batch mode and provide the name of an existing (or parameterized) script as a command line argument. OCEAN scripts can be created
  - From the Affirma™ analog circuit design environment window with the command `Session – Save Script`
  - By hand (by you or someone else in your organization) with a text editor

  For information about creating scripts, see “Creating OCEAN Scripts” on page 34.

All the OCEAN commands are described in this manual, and online help is available for all these commands. For information about using the OCEAN online help, see “OCEAN Online Help” on page 20.

**Note**: The current version of OCEAN has some specific issues that are addressed in Appendix A, “OCEAN 4.4.3 Issues.” Please refer to this appendix before using OCEAN.
Using OCEAN Interactively

You can run OCEAN from a UNIX prompt or from the Cadence® design framework II (DFII) Command Interpreter Window (CIW).

**Note:** The primary use model is to use OCEAN in a UNIX shell. Unless otherwise indicated, the rest of this chapter assumes that you are working from OCEAN in a UNIX shell.

Using OCEAN from a UNIX Shell

To start OCEAN from a UNIX prompt, type the following command:

```
ocean
```

This loads and reads your `.oceanrc` file. (You can place OCEAN commands in your `.oceanrc` file.)

**Note:** OCEAN does not read your `.cdsinit` file. If you want your `.cdsinit` file read, you must load it in your `.oceanrc` file.

The OCEAN prompt appears:

```
ocean>
```

If you do not see this prompt after starting OCEAN, press Return. If you still do not see this prompt, you may have redefined the prompt with the `setPrompt` command. (This does not affect OCEAN; the prompt just will not indicate OCEAN is running.)

Now you can start typing OCEAN commands interactively. For an example of interactive use, see “Interactive Session Demonstrating the OCEAN Use Model” on page 33.

To quit the OCEAN executable from UNIX, type the following command:

```
exit
```

Using OCEAN from the CIW

You can type OCEAN commands in the CIW after you bring up the Affirma analog circuit design environment. (Starting the design environment loads the required OCEAN files.)

Your `.oceanrc` file is not automatically read when you start the DFII software (using the `icms` command). Therefore, you might want to load your `.oceanrc` file manually in the CIW if you need information that it contains.
Interactive Session Demonstrating the OCEAN Use Model

The following figure shows a typical set of simulation tasks that you might perform interactively in OCEAN with the corresponding commands.

```
Start OCEAN and specify your simulator.
```

```
ocean
simulator( 'spectre )
```

```
Specify an AC analysis.
```

```
analysis( 'ac ?start 1 ?stop 1000 ?lin 100 )
or
ac( 1 1000 "linear" 100 )
```

```
Set a design variable.
```

```
desVar( "rs" 1k )
```

```
Perform the first simulation run.
```

```
run( )
```

```
Change a design variable.
```

```
desVar( "rs" 2k )
```

```
Perform the second simulation run
```

```
run( )
```

```
Specify a transient analysis.
```

```
analysis( 'tran ?stop 1u ) (Spectre only)
or
tran( 0 1u 1n )
```

```
Perform the third simulation run.
```

```
run( )
```

On the second and third run, the AC analysis runs because it is still active. If you do not want it to run, you must disable it with the following command:

```
delete( 'analysis 'ac )
```

The simulator is not called and run until the `run()` command is entered.
The commands can be given in any order, as long as they are all defined before the `run()` command.

## Creating OCEAN Scripts

You can modify the included sample script files or create script files interactively from the Affirma analog circuit design environment.

### Creating Scripts Using Sample Script Files

You can create your own script files with a text editor using the sample scripts as examples, or you can make copies of the sample scripts and modify them as needed using a text editor. The scripts can be found in the following directory:

`your_install_dir/tools/dfII/samples/artist/OCEAN`

Refer to the `README` file in this directory for information about the scripts.

### Creating Scripts from the Affirma Analog Circuit Design Environment

When you perform tasks in the design environment, the associated OCEAN commands are automatically stored in the `simulatorx.ocn` file in your `netlist` directory. For example, if you start the Cadence software, open the Affirma analog circuit design environment window, and run a simulation using the Cadence SPICE simulator, a `cdsSpice0.ocn` file is created in your `netlist` directory. You can load this `cdsSpice0.ocn` script as described in “Loading OCEAN Scripts” on page 37.

### Selectively Creating Scripts

You can be selective about the information that is created in your `.ocn` script. The Affirma analog circuit design environment has a feature that lets you create an OCEAN script based on the state of your current session. The following example illustrates how using this feature is different than using the automatic script generation feature.

Consider the following task flow:

1. Start the Affirma analog circuit design environment.
2. Specify a DC analysis.
3. Select nets on the schematic to save.
4. Run the simulation.

5. Turn off the DC analysis.

6. Select a transient analysis.

7. Run the simulation.

8. Save the script from the Affirma analog circuit design environment.

The script that is created, called oceanScript.ocn by default, contains only the selected nets, the transient analysis, and the run command. The script does not contain the DC analysis because it was turned off.

In contrast, the simulator0.ocn script, which is automatically created in the netlist directory, contains all of the commands, including the DC analysis and the current state of the analysis (on or off).

Creating a Script

To selectively create a script from the Affirma analog circuit design environment,

1. Start the Cadence software with the executable you prefer; for example, icms&

   The CIW appears.

2. From the CIW, choose Tools – Analog Environment – Simulation.

   The Affirma analog circuit design environment window appears.

3. Perform all of the design environment tasks that you want to capture in the script.


   The Save Ocean Script to File form appears.

5. Click OK to accept the default file name (.oceanScript.ocn), or change the name for the file and click OK.

   A script containing the OCEAN commands for the tasks you performed is created. For information about how to load this script, see “Loading OCEAN Scripts” on page 37.

Controlling What Is Included in Scripts

You can use .cdsenv variables to alter the OCEAN script that is created when you choose Session – Create Script in the Affirma analog circuit design environment. One variable
allows you to include default environment settings in a script, two other variables allow you to run procedures before and after a script is created.

Including Default Control Statements
To save every control statement, including default statements, in your OCEAN script, add the following line to your .cdsenv file.

```
asimenv.misc saveDefaultsToOCEAN boolean t
```

Setting `saveDefaultsToOCEAN` to `t` results in a complete dump of the current circuit design environment, defaults and all. Because the created OCEAN script contains all the settings, you might use this variable when you plan to archive a script, for example.

If `saveDefaultsToOCEAN` is not set to `t`, the created OCEAN script contains only those items that you explicitly set to some value other than their default.

Running Functions Before or After Creating a Script
The information in this section describes how you can specify functions to be run before or after a script is created. You can use these functions, for example, to add information at the beginning or end of a script. To use this capability follow these steps.

1. Decide when you want the functions to run.
   - Add the following line to your .cdsenv file to run the function `preOceanFunc` before the OCEAN script is created.
     
     ```
     asimenv.misc preSaveOceanScript string "preOceanFunc"
     ```
   - Add the following line to your .cdsenv file to run the function `postOceanFunc` after the OCEAN script is created.
     
     ```
     asimenv.misc postSaveOceanScript string "postOceanFunc"
     ```

2. Use the following syntax to specify the functions.

   ```
   preOceanFunc( session fp )
   postOceanFunc( session fp )
   ```

   In this syntax, `session` is the OASIS session and `fp` is the file pointer to the OCEAN script file. For guidance on determining the `session` to use, see the Affirma AMS Circuit Design Environment SKILL Language Reference.

3. Load the functions in your .cdsinit file.

   For example, you might add the following lines to your .cdsenv file.
asimenv.misc  preSaveOceanScript string "MYfirstProc"
asimenv.misc  postSaveOceanScript string "MYlastProc"

The functions MYfirstProc and MYlastProc might be defined like this.

procedure( MYfirstProc( session fp)
   fprintf(fp "; This will be the first line in the ocean script.\n")
 )

procedure( MYlastProc( session fp)
   fprintf(fp "; This will be the last line in the ocean script.\n")
 )

If these procedures are defined in a file called myOceanProcs.il, you can load them by adding to your .cdsinit file a command like the following.

load "myOceanProcs.il"

When you choose Session – Create Script, first the preSaveOceanScript procedure is called, then the OCEAN script is created, then the postSaveOceanScript procedure is called.

**Loading OCEAN Scripts**

You can load OCEAN scripts from OCEAN (in UNIX) or from the CIW.

**From a UNIX Shell**

To load an OCEAN script,

1. Type the following command to start OCEAN:
   
   ocean
   
   The OCEAN prompt appears.

2. Use the SKILL load command to load your script:
   
   load( "script_name.ocn" )
   
   Messages about the progress of your script appear.

**From the CIW**

To load an OCEAN script,

1. Start the Cadence software with the executable you prefer, for example
icms&

The CIW appears.

2. In the text entry field, use the SKILL `load` command to load your script:

```skill
load("script_name.ocn")
```

Messages about the progress of your script appear in the CIW.

**Note:** OCEAN does not read your `.cdsinit` file. If you want your `.cdsinit` file read, you must load it in your `.oceanrc` file.

## Running Multiple Simulators

There are times when you might want to run more than one simulator. You might be benchmarking simulators or comparing results. In OCEAN, you can only use one simulator per OCEAN session. If you change simulators, you must start a new OCEAN session. This is because some OCEAN command arguments are simulator specific, and therefore change when the simulator changes. For example, the arguments to the `option` command are simulator specific. (No two simulators have the exact same options.) The analyses are typically simulator specific also.

## OCEAN Tips

The information in this section can help you solve problems that you encounter while using OCEAN.

- While working in OCEAN, you might get the following SKILL error message:

  ```skill
  *Error* eval: unbound variable - nameOfVariable
  ```

  In this case, you need to see if you have an undeclared variable or if you are missing a single quotation mark (`) or a quotation mark (""") for one of your arguments. For example, the following command returns an error message stating that `fromVal` is an unbound variable because the variable has not been declared:

  ```skill
  analysis('tran ?from fromVal)
  ```

  However, the following pair of statements work correctly because `fromVal` has a value (is bound).

  ```skill
  fromVal=0
  analysis('tran ?from fromVal)
  ```
If you get an error in an OCEAN session, you are automatically put into the SKILL debugger. In this case, you see a prompt similar to this:

```
ocean-Debug 2>
```

You can continue working. However, if you would like to get out of the debugger, you can type

```
debugQuit()
```

Now you are back to the normal prompt:

```
ocean>
```

If it appears that OCEAN does not accept your input, or OCEAN appears to hang, then you may have forgotten to enter a closing quotation mark. Type `)` to close all strings. For more information, and some examples, see “Recovering from an Omitted Quotation Mark” on page 22.

In SKILL, the following formats are equivalent: `(one two)` and `one(two)`. Results might be returned in either format. For example, OCEAN might return `ac(tran)` or `(ac tran)`, but the two forms are equivalent.

You can check your script for simple syntax errors by running SKILL lint. For example, you might use a command like

```
sklint -file myScript.ocn
```

From within OCEAN, you can run SKILL lint by typing the following at the OCEAN prompt:

```
sklint(?file "yourOceanScript.ocn")
```

Running SKILL lint helps catch basic errors, such as unmatched parentheses and strings that are not closed.
Introduction to SKILL

This chapter introduces you to the basic concepts that can help you get started with the Cadence® SKILL programming language. In this chapter, you can find information about

- The Advantages of SKILL on page 41
- Naming Conventions on page 42
- Arithmetic Operators on page 42
- Scaling Factors on page 42
- Relational and Logical Operators on page 43
- SKILL Syntax on page 46
- Arithmetic and Logical Expressions on page 49

The Advantages of SKILL

The SKILL programming language lets you customize and extend your design environment. SKILL provides a safe, high-level programming environment that automatically handles many traditional system programming operations, such as memory management. SKILL programs can be immediately run in the Cadence environment.

SKILL is ideal for rapid prototyping. You can incrementally validate the steps of your algorithm before incorporating them in a larger program.

SKILL leverages your investment in Cadence technology because you can combine existing functionality and add new capabilities.

SKILL lets you access and control all the components of your tool environment: the User Interface Management System, the Design Database, and the commands of any integrated design tool. You can even loosely couple proprietary design tools as separate processes with SKILL's interprocess communication facilities.
Naming Conventions

The recommended naming scheme is to use uppercase and lowercase characters to separate your code from code developed by Cadence.

All code developed by Cadence Design Systems typically names global variables and functions with up to three lowercase characters, that signify the code package, and the name starting with an uppercase character. For example, `dmiPurgeVersions()` or `hnlCellOutputs`. All code developed outside Cadence should name global variables by starting them with an uppercase character, such as `AcmeGlobalForm`.

Arithmetic Operators

SKILL provides many arithmetic operators. Each operator corresponds to a SKILL function, as shown in the following table.

**Sample SKILL Operators**

<table>
<thead>
<tr>
<th>Operators in Descending Precedence</th>
<th>Underlying Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>exponentiation</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
</tr>
<tr>
<td>–</td>
<td>minus</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
</tr>
<tr>
<td>!=</td>
<td>nequal</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
</tbody>
</table>

Scaling Factors

SKILL provides a set of scaling factors that you can add to the end of a decimal number (integer or floating point) to achieve the scaling you want.

- Scaling factors must appear immediately after the numbers they affect. Spaces are not allowed between a number and its scaling factor.
- Only the first nonnumeric character that appears after a number is significant. Other characters following the scaling factor are ignored. For example, for the value 2.3mvolt, the \( m \) is significant, and the \( volt \) is discarded. In this case, \( volt \) is only for your reference.
If the number being scaled is an integer, SKILL tries to keep it an integer; the scaling factor must be representable as an integer (that is, the scaling factor is an integral multiplier and the result does not exceed the maximum value that can be represented as an integer). Otherwise, a floating-point number is returned.

The scaling factors are listed in the following table.

### Scaling Factors

<table>
<thead>
<tr>
<th>Character</th>
<th>Name</th>
<th>Multiplier</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Tera</td>
<td>$10^{12}$</td>
<td>$10T$ [1.0e13]</td>
</tr>
<tr>
<td>G</td>
<td>Giga</td>
<td>$10^9$</td>
<td>$10G$ [10,000,000,000]</td>
</tr>
<tr>
<td>M</td>
<td>Mega</td>
<td>$10^6$</td>
<td>$10M$ [10,000,000]</td>
</tr>
<tr>
<td>K</td>
<td>Kilo</td>
<td>$10^3$</td>
<td>$10K$ [10,000]</td>
</tr>
<tr>
<td>%</td>
<td>percent</td>
<td>$10^{-2}$</td>
<td>$5%$ [0.05]</td>
</tr>
<tr>
<td>m</td>
<td>milli</td>
<td>$10^{-3}$</td>
<td>$5m$ [5.0e-3]</td>
</tr>
<tr>
<td>u</td>
<td>micro</td>
<td>$10^{-6}$</td>
<td>$1.2u$ [1.2e-6]</td>
</tr>
<tr>
<td>n</td>
<td>nano</td>
<td>$10^{-9}$</td>
<td>$1.2n$ [1.2e-9]</td>
</tr>
<tr>
<td>p</td>
<td>pico</td>
<td>$10^{-12}$</td>
<td>$1.2p$ [1.2e-12]</td>
</tr>
<tr>
<td>f</td>
<td>femto</td>
<td>$10^{-15}$</td>
<td>$1.2f$ [1.2e-15]</td>
</tr>
</tbody>
</table>

**Note:** The characters used for scaling factors depend on your target simulator. For example, if you are using cdsSpice, the scaling factor for $M$ is different than shown in the previous table, because cdsSpice is not case sensitive. In cdsSpice, $M$ and $m$ are both interpreted as $10^{-3}$, so $ME$ or $me$ is used to signify $10^6$.

### Relational and Logical Operators

This section introduces you to

- Relational Operators: <, <=, >, >=, ==, !=
- Logical Operators: !, &&, ||
Relational Operators

Use the following operators to compare data values. SKILL generates an error if the data types are inappropriate. These operators all return \texttt{t} or \texttt{nil}.

**Sample Relational Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arguments</th>
<th>Function</th>
<th>Example</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>numeric</td>
<td>lessp</td>
<td>3 &lt; 5</td>
<td>\texttt{t}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 &lt; 2</td>
<td>\texttt{nil}</td>
</tr>
<tr>
<td>&lt;=</td>
<td>numeric</td>
<td>leqp</td>
<td>3 &lt;= 4</td>
<td>\texttt{t}</td>
</tr>
<tr>
<td>&gt;</td>
<td>numeric</td>
<td>greaterp</td>
<td>5 &gt; 3</td>
<td>\texttt{t}</td>
</tr>
<tr>
<td>&gt;=</td>
<td>numeric</td>
<td>geqp</td>
<td>4 &gt;= 3</td>
<td>\texttt{t}</td>
</tr>
<tr>
<td>==</td>
<td>numeric</td>
<td>equal</td>
<td>3.0 == 3</td>
<td>\texttt{t}</td>
</tr>
<tr>
<td></td>
<td>string</td>
<td></td>
<td>&quot;abc&quot; == &quot;ABC&quot;</td>
<td>\texttt{nil}</td>
</tr>
<tr>
<td>!=</td>
<td>numeric</td>
<td>nequal</td>
<td>&quot;abc&quot; != &quot;ABC&quot;</td>
<td>\texttt{t}</td>
</tr>
<tr>
<td></td>
<td>string</td>
<td></td>
<td>&quot;abc&quot; != &quot;ABC&quot;</td>
<td>\texttt{t}</td>
</tr>
</tbody>
</table>

Knowing the function name is helpful because error messages mention the function (\texttt{greaterp} below) instead of the operator (\texttt{>}).

\begin{verbatim}
1 > "abc"
Message: *Error* greaterp: can't handle (1 > "abc")
\end{verbatim}

Logical Operators

SKILL considers \texttt{nil} as FALSE and any other value as TRUE. The \texttt{and (&&)} and or (\texttt{||}) operators only evaluate their second argument if it is required for determining the return result.

**Sample Logical Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arguments</th>
<th>Function</th>
<th>Example</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>general</td>
<td>and</td>
<td>3 &amp;&amp; 5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5 &amp;&amp; 3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t &amp;&amp; nil</td>
<td>\texttt{nil}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\texttt{nil} &amp;&amp; t</td>
<td>\texttt{nil}</td>
</tr>
</tbody>
</table>
The && and || operators return the value last computed. Consequently, both && and || operators can be used to avoid cumbersome if or when expressions.

The following example illustrates the difference between using && and || operators and using if or when expressions.

You do not need to use

```clojure
If (usingcolor then
currentcolor=getcolor( )
else
currentcolor=nil
)
```

Instead use

```clojure
currentcolor=usingcolor && getcolor( )
```

### Using &&

When SKILL creates a variable, it gives the variable a value of unbound to indicate that the variable has not been initialized yet. Use the boundp function to determine whether a variable is bound. The boundp function

- Returns t if the variable is bound to a value
- Returns nil if the variable is not bound to a value

Suppose you want to return the value of a variable `trMessages`. If `trMessages` is unbound, retrieving the value causes an error. Instead, use the expression

```clojure
boundp( ’trMessages ) && trMessages
```
Using ||

Suppose you have a default name, such as noName, and a variable, such as userName. To use the default name if userName is nil, use the following expression:

```
userName || "noName"
```

SKILL Syntax

This section describes SKILL syntax, which includes the use of special characters, comments, spaces, parentheses, and other notation.

Special Characters

Certain characters are special in SKILL. These include the *infix* operators such as less than (<), colon (:), and assignment (=). The following table lists these special characters and their meaning in SKILL.

**Note:** All nonalphanumeric characters (other than _ and ?) must be preceded *(escaped)* by a backslash (\) when you use them in the name of a symbol.

**Special Characters in SKILL**

<table>
<thead>
<tr>
<th>Character</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>backslash</td>
<td>Escape for special characters</td>
</tr>
<tr>
<td>( )</td>
<td>parentheses</td>
<td>Grouping of list elements, function calls</td>
</tr>
<tr>
<td>[ ]</td>
<td>brackets</td>
<td>Array index, super right bracket</td>
</tr>
<tr>
<td>'</td>
<td>single quotation mark</td>
<td>Specifies a symbol (quoting the expression prevents its evaluation)</td>
</tr>
<tr>
<td>&quot;</td>
<td>quotation mark</td>
<td>String delimiter</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td>Optional delimiter between list elements</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
<td>Line-style comment character</td>
</tr>
<tr>
<td>+, -, *, /</td>
<td>arithmetic</td>
<td>For arithmetic operators; the /* and */ combinations are also used as comment delimiters</td>
</tr>
<tr>
<td>!, ^, &amp; ,</td>
<td>logical</td>
<td>For logical operators</td>
</tr>
</tbody>
</table>
White Space

White space sometimes takes on semantic significance and a few syntactic restrictions must therefore be observed.

Write function calls so the name of a function is immediately followed by a left parenthesis; no white space is allowed between the function name and the parenthesis. For example

\( f(a \ b \ c) \) and \( g() \) are legal function calls, but \( f \ (a \ b \ c) \) and \( g \ () \) are illegal.

The unary minus operator must immediately precede the expression it applies to. No white space is allowed between the operator and its operand. For example

\( -1 \), \( -a \), and \( -(a*b) \) are legal constructs, but \( -1 \), \( -a \), and \( -(a*b) \) are illegal.

The binary minus (subtract) operator should either be surrounded by white space on both sides or be adjacent to non-white space on both sides. To avoid ambiguity, one or the other method should be used consistently. For example:

\( a - b \) and \( a-b \) are legal constructs for binary minus, but \( a -b \) is illegal.

Comments

SKILL permits two different styles of comments. One style is block oriented, where comments are delimited by /* and */. For example:

/* This is a block of (C style) comments
comment line 2
comment line 3 etc.
*/

The other style is line-oriented where the semicolon (;) indicates that the rest of the input line is a comment. For example:
x = 1 ; comment following a statement
; comment line 1
; comment line 2 and so forth

For simplicity, line-oriented comments are recommended. Block-oriented comments cannot be nested because the first */ encountered terminates the whole comment.

**Role of Parentheses**

Parentheses ( ) delimit the names of functions from their argument lists and delimit nested expressions. In general, the innermost expression of a nested expression is evaluated first, returning a value used in turn to evaluate the expression enclosing it, and so on until the expression at the top level is evaluated. There is a subtle point about SKILL syntax that C programmers, in particular, must be very careful to note.

**Parentheses in C**

In C, the relational expression given to a conditional statement such as if, while, and switch must be enclosed by an outer set of parentheses for purely syntactical reasons, even if that expression consists of only a single Boolean variable. In C, an if statement might look like

```c
if (done) i=0; else i=1;
```

**Parentheses in SKILL**

In SKILL, parentheses are used for specifying lists, calling functions, delimiting multiple expressions, and controlling the order of evaluation. You can write function calls in prefix notation

```
(fn2 arg1 arg2) or (fn0)
```

as well as in the more conventional algebraic form

```
fn2(arg1 arg2) or fn0()
```

The use of syntactically redundant parentheses causes variables, constants, or expressions to be interpreted as the names of functions that need to be further evaluated. Therefore,

- Never enclose a constant or a variable in parentheses by itself; for example, (1), (x).
- For arithmetic expressions involving *infix* operators, you can use as many parentheses as necessary to force a particular order of evaluation, but never put a pair of parentheses immediately outside another pair of parentheses; for example, ((a + b)): the
expression delimited by the inner pair of parentheses would be interpreted as the name of a function.

For example, because \textit{if} evaluates its first argument as a logical expression, a variable containing the logical condition to be tested should be written without any surrounding parentheses; the variable by itself is the logical expression. This is written in SKILL as

\begin{equation*}
\text{if}(\ \text{done} \ \text{then} \ i = 0 \ \text{else} \ i = 1)
\end{equation*}

\section*{Line Continuation}

SKILL places no restrictions on how many characters can be placed on an input line, even though SKILL does impose an 8,191 character limit on the strings being entered. The parser reads as many lines as needed from the input until it has read in a complete form (that is, expression). If there are parentheses that have not yet been closed or binary \textit{infix} operators whose right sides have not yet been given, the parser treats carriage returns (that is, newlines) just like spaces.

Because SKILL reads its input on a form-by-form basis, it is rarely necessary to “continue” an input line. There might be times, however, when you want to break up a long line for aesthetic reasons. In that case, you can tell the parser to ignore a carriage return in the input line simply by preceding it immediately with a backslash (\).

\begin{verbatim}
string = "This is \n
  a test."
=> "This is a test."
\end{verbatim}

\section*{Arithmetic and Logical Expressions}

\textit{Expressions} are SKILL objects that also evaluate to SKILL objects. SKILL performs a computation as a sequence of function evaluations. A SKILL \textit{program} is a sequence of expressions that perform a specified action when evaluated by the SKILL interpreter.

There are two types of primitive expressions in SKILL that pertain to OCEAN: constants and variables.

\section*{Constants}

A \textit{constant} is an expression that evaluates to itself. That is, evaluating a constant returns the constant itself. Examples of constants are 123, 10.5, and "abc".
Variables

A *variable* stores values used during the computation. The variable returns its value when evaluated. Examples of variables are $a$, $x$, and `init_var`.

When the interpreter evaluates a variable whose value has not been initialized, it displays an error message telling you that you have an unbound variable. For example, you get an error message when you misspell a variable because the misspelling creates a new variable.

```plaintext
myVariable
```

causes an error message because it has been referenced before being assigned, whereas

```plaintext
myVariable = 5
```

works.

When SKILL creates a variable, it gives the variable an initial value of `unbound`. It is an error to evaluate a variable with this value because the meaning of `unbound` is *that-value-which-represents-no-value*. `unbound` is not the same as `nil`.

Using Variables

You do not need to declare variables in SKILL as you do in C. SKILL creates a variable the first time it encounters the variable in a session. Variable names can contain

- Alphanumeric characters
- Underscores (_)
- Question marks
- Digits

The first character of a variable must be an alphanumeric character or an underscore. Use the assignment operator to store a value in a variable. You enter the variable name to retrieve its value.

```plaintext
lineCount = 4                      => 4
lineCount                          => 4
lineCount = "abc"                  => "abc"
lineCount                          => "abc"
```
Creating Arithmetic and Logical Expressions

Constants, variables, and function calls can be combined with the *infix* operators, such as less than (<), colon (:), and greater than (>) to form arithmetic and logical expressions. For example: \(1+2\), \(a*b+c\), \(x>y\).

You can form arbitrarily complicated expressions by combining any number of the primitive expressions described above.
Working with SKILL

This chapter provides information on using SKILL functions. It includes information on the types of SKILL functions, the types of data accepted as arguments, how data types are used, and how to declare and define functions. In this chapter, you can find information about:

- **Skill Functions** on page 53
- **Data Types** on page 53
- **Arrays** on page 56
- **Concatenating Strings (Lists)** on page 56
- **Declaring a SKILL Function** on page 58
- **Skill Function Return Values** on page 60
- **Syntax Functions for Defining Functions** on page 60

**Skill Functions**

There are two basic types of SKILL functions:

- **Functions** carry out statements and return data that can be redirected to other commands or functions.

- **Commands** are functions that carry out statements defined by the command and return `true` or `nil`. Some commands return the last argument entered, but the output cannot be redirected.

**Data Types**

SKILL supports several data types, including integer and floating-point numbers, character strings, arrays, and a highly flexible linked list structure for representing aggregates of data. The simplest SKILL expression is a single piece of data, such as an integer, a floating-point
number, or a string. SKILL data is case sensitive. You can enter data in many familiar ways, including the following:

### Sample SKILL Data Items

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Syntax Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>5</td>
</tr>
<tr>
<td>floating point number</td>
<td>5.3</td>
</tr>
<tr>
<td>text string</td>
<td>&quot;Mary had a little lamb&quot;</td>
</tr>
</tbody>
</table>

For symbolic computation, SKILL has data types for dealing with symbols and functions.

For input/output, SKILL has a data type for representing I/O ports. The table below lists the data types supported by SKILL with their internal names and prefixes.

### Data Types Supported by SKILL

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Internal Name</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>array</td>
<td>a</td>
</tr>
<tr>
<td>boolean</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>floating-point number</td>
<td>flonum</td>
<td>f</td>
</tr>
<tr>
<td>any data type</td>
<td>general</td>
<td>g</td>
</tr>
<tr>
<td>linked list</td>
<td>list</td>
<td>l</td>
</tr>
<tr>
<td>floating-point number or integer</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>user-defined type</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>I/O port</td>
<td>port</td>
<td>p</td>
</tr>
<tr>
<td>symbol</td>
<td>symbol</td>
<td>s</td>
</tr>
<tr>
<td>symbol or character string</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>character string (text)</td>
<td>string</td>
<td>t</td>
</tr>
<tr>
<td>window type</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>integer number</td>
<td>fixnum</td>
<td>x</td>
</tr>
</tbody>
</table>

### Numbers

SKILL supports the following numeric data types:
Integers
Floating-point

Both integers and floating-point numbers may use scaling factors to scale their values. For information on scaling factors, see “Scaling Factors” on page 42.

Atoms

An atom is any data object that is not a grouping or collection of other data objects. Built into SKILL are several special atoms that are fundamental to the language.

nil
The nil atom represents both a false logical condition and an empty list.

t
The symbol t represents a true logical condition.

Both nil and t always evaluate to themselves and must never be used as the name of a variable.

unbound
To make sure you do not use the value of an uninitialized variable, SKILL sets the value of all symbols and array elements initially to unbound so that such an error can be detected.

Constants and Variables

Supported constants and variables are discussed in “Arithmetic and Logical Expressions” on page 3-14.

Strings

Strings are sequences of characters; for example, "abc" or "123". A string is marked off by quotation marks, just as in the C language; the empty string is represented as "". The SKILL parser limits the length of input strings to a maximum of 8,191 characters. There is, however, no limit to the length of strings created during program execution. Strings of more than 8,191 characters can be created by applications and used in SKILL if they are not given as arguments to SKILL string manipulation functions.

When typing strings, you specify

Printable characters (except a quotation mark) as part of a string without preceding them with the backslash (\) escape character
Arrays

An array represents aggregate data objects in SKILL. Unlike simple data types, you must explicitly create arrays before using them so the necessary storage can be allocated. SKILL arrays allow efficient random indexing into a data structure using familiar syntax.

- Arrays are not typed. Elements of the same array can be different data types.
- SKILL provides run-time array bounds checking. The array bounds are checked with each array access during runtime. An error occurs if the index is outside the array bounds.
- Arrays are one dimensional. You can implement higher dimensional arrays using single dimensional arrays. You can create an array of arrays.

Allocating an Array of a Given Size

Use the declare function to allocate an array of a given size.

```skill
declare( week[7] ) => array[7]:9780700
week => array[7]:9780700
type( week ) => array
```

- The declare function returns the reference to the array storage and stores it as the value of week.
- The type function returns the symbol array.

Concatenating Strings (Lists)

Concatenating a List of Strings with Separation Characters (buildString)

buildString makes a single string from the list of strings. You specify the separation character in the third argument. A null string is permitted. If this argument is omitted, buildString provides a separating space as the default.
concatenating two or more input strings (strcat)

strcat creates a new string by concatenating two or more input strings. The input strings are left unchanged.

strcat( "l" "ab" "ef" ) => "labef"
You are responsible for any separating space.

strcat( "a" "b" "c" "d" ) => "abcd"
strcat( "a" "b" "c" "d" ) => "a b c d"

append a maximum number of characters from two input strings (strncat)

strncat is similar to strcat except that the third argument indicates the maximum number of characters from string2 to append to string1 to create a new string. string1 and string2 are left unchanged.

strncat( "abcd" "efghi" 2) => "abcdef"
strncat( "abcd" "efghijk" 5) => "abcdefghi"

comparing strings

comparing two strings or symbol names alphabetically (alphalessp)

alphalessp compares two objects, which must be either a string or a symbol, and returns true if arg1 is alphabetically less than arg2. alphalessp can be used with the sort function to sort a list of strings alphabetically. For example:

stringList = '( "xyz" "abc" "ghi" )
sort( stringList 'alphalessp ) => ("abc" "ghi" "xyz")

the next example returns a sorted list of all the files in the login directory:
sort( getDirFiles( "~" ) 'alphalessp )
Comparing Two Strings Alphabetically (strcmp)

`strcmp` compares two strings. (To simply test if two strings are equal or not, you can use the `equal` command.) The return values for `strcmp` are explained in the following table.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>string1</code> is alphabetically greater than <code>string2</code>.</td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> is alphabetically equal to <code>string2</code>.</td>
</tr>
<tr>
<td>-1</td>
<td><code>string1</code> is alphabetically less than <code>string2</code>.</td>
</tr>
</tbody>
</table>

```
strcmp( "abc" "abb" )=> 1
strcmp( "abc" "abc")=> 0
strcmp( "abc" "abd")=> -1
```

Comparing Two String or Symbol Names Alphanumerically or Numerically (alphaNumCmp)

`alphaNumCmp` compares two string or symbol names. If the third optional argument is not `nil` and the first two arguments are strings holding purely numeric values, a numeric comparison is performed on the numeric representation of the strings. The return values are explained in the following table.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>arg1</code> is alphanumerically greater than <code>arg2</code>.</td>
</tr>
<tr>
<td>0</td>
<td><code>arg1</code> is alphanumerically identical to <code>arg2</code>.</td>
</tr>
<tr>
<td>-1</td>
<td><code>arg2</code> is alphanumerically greater than <code>arg1</code>.</td>
</tr>
</tbody>
</table>

Declaring a SKILL Function

To refer to a group of statements by name, use the `procedure` declaration to associate a name with the group. The group of statements and the name make up a SKILL function.

- The name is known as the function name.
- The group of statements is the function body.

To run the group of statements, mention the function name followed immediately by `()`. 
The `clearplot` command below erases the Waveform window and then plots a net.

```skill
procedure( clearplot( netname )
  clearAll( )
  plot( v (netName))
)
```

**Defining Function Parameters**

To make your function more versatile, you can identify certain variables in the function body as formal parameters.

When you start your function, you supply a parameter value for each formal parameter.

**Defining Local Variables (let)**

Local variables can be used to establish temporary values in a function. This is done using the `let` statement. When local variables are defined, they are known only within the `let` statement and are not available outside the `let` statement.

When the function is defined, the `let` statement includes the local variables you want to define followed by one or more SKILL expressions. The variables are initialized to `nil`. When the function runs, it returns the last expression computed within its body. For example:

```skill
procedure( test ( x )
  let(( a b )
    a=1
    b=2
    x * a+b
  )
)
```

- The function name is `test`.
- The local variables are `a` and `b`.
- The local variables are initialized to `nil`.
- The return value is the value of `x * a + b`. 
Skill Function Return Values

All SKILL functions compute a data value known as the return value of the function. Throughout this document, the right arrow (=>) denotes the return value of a function call. You can

- Assign the return value to a SKILL variable
- Pass the return value to another SKILL function

Any type of data can be a return value.

Syntax Functions for Defining Functions

SKILL supports the following syntax functions for defining functions. You should use the procedure function in most cases.

procedure

The procedure function is the most general and is easiest to use and understand.

The procedure function provides the standard method of defining functions. Its return value is the symbol with the name of the function. For example:

```
procedure( trAdd( x y )
   "Display a message and return the sum of x and y"
   printf( "Adding %d and %d ... %d \n" x y x+y )
   x+y ) => trAdd
trAdd( 6 7 ) => 13
```

Terms and Definitions

function, procedure

In SKILL, the terms procedure and function are used interchangeably to refer to a parameterized body of code that can be executed with actual parameters bound to the formal parameters. SKILL can represent a function as both a hierarchical list and as a function object.

argument, parameter

The terms argument and parameter are used interchangeably.
The actual arguments in a function call correspond to the formal arguments in the declaration of the function.

<table>
<thead>
<tr>
<th>expression</th>
<th>A use of a SKILL function, often by means of an operator supplying required parameters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>function body</td>
<td>The collection of SKILL expressions that define the function’s algorithm.</td>
</tr>
</tbody>
</table>
The OCEAN environment commands let you start, control, and quit the OCEAN environment. These commands also let you specify the simulator to be used.
setup

```ruby
setup( [?numberNotation s_numberNotation] [?precision x_precision] [?reportStyle s_reportStyle] [?charsPerLine x_charsPerLine] [?messageOn g_messageOn] ) => t/nil
```

**Description**

Specifies default values for parameters.

**Arguments**

- **s_numberNotation**
  - Specifies the notation for printed information.
  - Default value: ‘suffix’
  - The format for each value is ‘suffix’: 1m, 1u, 1n, etc.; ‘engineering’: 1e-3, 1e-6, 1e-9, etc.; ‘scientific’: 1.0e-2, 1.768e-5, etc.; ‘none’.
  - The value ‘none’ is provided so that you can turn off formatting and therefore greatly speed up printing for large data files.

- **x_precision**
  - Specifies the number of significant digits that are printed.
  - Valid values: 1 through 16
  - Default value: 6

- **s_reportStyle**
  - Specifies the format of the output of the report command.
  - Valid values: spice, paramValPair
  - Default value: paramValPair

  The spice format is:

<table>
<thead>
<tr>
<th>Param1</th>
<th>Param2</th>
<th>Param3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name1</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Name2</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Name3</td>
<td>value</td>
<td>value</td>
</tr>
</tbody>
</table>
The paramValPair format is:

Name1
Param1=value Param2=value Param3=value

Name2
Param1=value Param2=value Param3=value

Name3
Param1=value Param2=value Param3=value

\textit{x_charsPerLine} \hfill Specifies the number of characters per line output to the display.
Default value: 80

\textit{g_messageOn} \hfill Specifies whether error messages are turned on.
Valid values: t, nil
Default value: t, which specifies that messages are turned on.

\textbf{Note:} The \texttt{?messageOn} feature is not implemented in this release.

\textbf{Value Returned}

\texttt{t} \hfill Returns \texttt{t} if the value is assigned to the name.

\texttt{nil} \hfill Returns \texttt{nil} if there is a problem.

\textbf{Examples}

\texttt{setup( \texttt{?numberNotation} \texttt{engineering} )} => \texttt{t}

Specifies that any printed information is to be in engineering mode by default.

\texttt{setup( \texttt{?precision} 5 )} => \texttt{t}

Specifies that 5 significant digits are to be printed.
Simulation Commands

The following OCEAN simulation commands let you set up and run your simulation.

- ac on page 69
- analysis on page 70
- appendPath on page 72
- createFinalNetlist on page 73
- createNetlist on page 74
- dc on page 75
- definitionFile on page 77
- delete on page 78
- design on page 80
- desVar on page 82
- envOption on page 83
- forcenode on page 84
- ic on page 85
- includeFile on page 86
- modelFile on page 87
- nodeset on page 88
- noise on page 89
- ocnDisplay on page 90
- off on page 92
- option on page 93
- path on page 94
- prependPath on page 95
- restore on page 96
resultsDir on page 97
run on page 98
save on page 101
saveOption on page 103
simulator on page 105
stimulusFile on page 106
store on page 107
temp on page 108
tran on page 109
ac

ac( g_fromValue g_toValue g_ptsPerDec ) => undefined/nil
ac( g_fromValue g_toValue t_incType g_points ) => undefined/nil

Description
Specifies an AC analysis.

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_fromValue</td>
<td>Starting value for the AC analysis.</td>
</tr>
<tr>
<td>g_toValue</td>
<td>Ending value.</td>
</tr>
<tr>
<td>g_ptsPerDec</td>
<td>Points per decade.</td>
</tr>
<tr>
<td>t_incType</td>
<td>Increment type.</td>
</tr>
<tr>
<td></td>
<td>Valid values: For the Affirma™ Spectre® analog circuit simulator, &quot;Linear&quot;, &quot;Logarithmic&quot;, or &quot;Automatic&quot;. For other simulators, &quot;Linear&quot; or &quot;Logarithmic&quot;.</td>
</tr>
<tr>
<td>g_points</td>
<td>Either the linear or the logarithmic value, which depends on t_incType.</td>
</tr>
</tbody>
</table>

Value Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined</td>
<td>The return value for this command/function is undefined.</td>
</tr>
<tr>
<td>nil</td>
<td>Returns nil and prints an error message if the analysis is not specified.</td>
</tr>
</tbody>
</table>

Examples

ac(1 10000 2)

Specifies an AC analysis from 1 to 10,000 with 2 points per decade.
ac(1 10000 "Linear" 100)

Specifies an AC analysis from 1 to 10,000 by 100.
**analysis**

`analysis(s_analysisType [?analysisOption1
g_analysisOptionValue1]... [?analysisOptionN
g_analysisOptionValueN]) => undefined/nil`

**Description**

Specifies the analysis to be simulated.

You can include as many analysis options as you want. Analysis options vary, depending on the simulator you are using. To include an analysis option, replace `analysisOption1` with the name of the desired analysis option and include another argument to specify the value for the option. If you have an AC analysis, the first option/value pair might be `[?from 0]`.

**Note:** Some simplified commands are available for basic SPICE analyses. See the `ac`, `dc`, `tran`, and `noise` commands. Use the `ocnHelp('analysis')` command for more information on the analysis types for the simulator you choose.

**Arguments**

- `s_analysisType`  
Type of the analysis. The valid values for this argument depend on the analyses that the simulator contains.  
The basic SPICE2G-like choices: `tran`, `dc`, `ac`, and `noise`.

- `?analysisOption1`  
Analysis option. The analysis options available depend on which simulator you use. (See the documentation for your simulator.)

  **Note:** If you are using the Affirma™ analog circuit simulator, see the information about analysis statements in the *Affirma Spectre Circuit Simulator Reference* manual for analysis options you can use.

- `g_analysisOptionValue1`  
Value for the analysis option.

- `?analysisOptionN`  
Any subsequent analysis option. The analysis options that are available depend on which simulator you use. (See the documentation for your simulator.)
g_analysisOptionValueN

Value for the analysis option.

Value Returned

undefined
The return value for this command/function is undefined.

nil
Returns nil and prints an error message if there is a problem specifying the analysis.

Examples

analysis( ‘ac ?start 1 ?stop 10000 ?lin 100 )

For the Affirma analog circuit simulator, specifies that an AC analysis be performed.

analysis( ‘tran ?start 0 ?stop 1u ?step 10n )

Specifies that a transient analysis be performed.

analysis(’dc ?oppoint "rawfile" ?save "allpub"
        ?param "temp" ?start -50 ?stop 100 )

Sweeps temperature for the Affirma analog circuit simulator.

analysis(‘dc ?saveOppoint t )

Saves the DC operating point information for the Affirma analog circuit simulator.
appendPath

$\text{appendPath}( t\_dirName1 \ldots [t\_dirNameN]) \Rightarrow t\_dirNameN/\text{nil}$

Description

Appends a new path to the end of the search path list. You can append as many paths as you want with this command.

For the Affirma analog circuit simulator, use of the modelFile command is preferred over the use of this command.

Arguments

- $t\_dirName1$ Directory path.
- $t\_dirNameN$ Additional directory path.

Value Returned

- $t\_dirNameN$ Returns the last path specified.
- nil Returns nil and prints an error message if the paths cannot be appended.

Example

$\text{appendPath}( "/usr/mnt/user/processA/models" ) \Rightarrow "/usr/mnt/user/processA/models"

Adds /usr/mnt/user/processA/models to the end of the current search path.
createFinalNetlist

createFinalNetlist() => t/nil

Description

Creates the final netlist for viewing purposes. The netlist also can be saved but is not required to run the simulator.

Note: This command is not supported for the Affirma analog circuit simulator. Use createNetlist instead.

Arguments

None.

Value Returned

t Returns t if the final netlist is created.

nil Returns nil and prints an error message otherwise.

Example

createFinalNetlist()

Creates the final netlist for the current simulation run.
createNetlist

createNetlist( [?recreateAll b_recreateAll] ) => t_filename/nil

Description

Creates the simulator input file.

If the design is specified as lib/cell/view, this command netlists the design, if required, and creates the simulator input file. When the \texttt{b\_recreateAll} argument is set to \texttt{t} and the design is specified as lib/cell/view, all the cells in the design hierarchy are renetlisted, before creating the simulator input file. If the design is specified as netlist file, that netlist is included in the simulator input file.

\textbf{Note:} This command does not work with socket simulators.

Arguments

\texttt{b\_recreateAll}

If set and the design is specified as lib/cell/view, the entire netlist is recreated.

Value Returned

\texttt{t\_fileName}

Returns the name of the simulator input file on success.

\texttt{nil}

otherwise \texttt{nil} is returned

Examples

createNetlist() => "/usr/foo/netlist/input.scs"

Creates simulator input file for the current simulation run.

design( ?lib "test" ?cell "mytest" ?view "spectre")
createNetlist( ?recreateAll t ) => "/usr/foo/netlist/input.scs"

Netlists and creates simulator input file for the current simulation run.
dc

dc( t_compName [ t_compParam ] g_fromValue g_toValue g_byValue )
   => undefined/nil

Description

Specifies a DC sweep analysis with limited options. If other analysis options are needed, use the analysis command.

Note: t_compParam is valid only for the Affirma analog circuit simulator.

Arguments

- **t_compName**: Name of the source (or component, for the Affirma analog circuit simulator) to sweep.
- **t_compParam**: For the Affirma analog circuit simulator, the component parameter to be swept.
- **g_fromValue**: Starting value for the DC analysis.
- **g_toValue**: Ending value.
- **g_byValue**: The increment at which to step through the analysis.

Value Returned

- **undefined**: The return value for this command/function is undefined.
- **nil**: Returns nil and prints an error message if the analysis is not specified.

Examples

```
dc("v1" "dc" 0 5 1)
dc("r1" "r" 0 5 1)
```

Specifies two DC sweep analyses for the Affirma analog circuit simulator.

```
dc("v1" 0 5 1)
```
Specifies one DC sweep analysis for a simulator other than the Affirma analog circuit simulator.
**definitionFile**

**definitionFile( t_fileName [t_fileName2 .. t_fileNameN ] ) => l_fileNames**

**Description**

Specifies definition files to be included in the simulator input file.

**Note:** This command does not work with socket simulators.

**Arguments**

- **t_fileName**
  
  The name of the definition file that would typically contain functions or parameter statements.

**Value Returned**

- **l_fileNames**
  
  A list of the file names specified; returned on success.

- **nil**
  
  Otherwise nil is returned.

**Example**

```plaintext
definitionFile( "functions.def" "constants.def" ) => 
("functions.def" "constants.def")

Includes functions.def and constants.def files in the simulator input file.
```
delete

delete( s_command [g_commandArg1] [g_commandArg2] … ) => t/nil

Description

Deletes all the information specified.

The $s_{command}$ argument specifies the command whose information you want to delete. If you include only this argument, all the information for the command is deleted. If you supply subsequent arguments, only those particular pieces of information are deleted as opposed to deleting all the information for that command.

Arguments

$s_{command}$ Command that was initially used to add the items that are now being deleted.
Valid values: analysis, desVar, path, save, ic, forcenode, monteCarlo, monteExpr, nodeset, optimizeGoal, optimizeVar, optimizeAlgoControl, optimizePlotOption

Using `delete(monteCarlo)` turns off the monteCarlo command and sets everything back to the defaults.

$g_{commandArg1}$ Argument corresponding to the specified command.

$g_{commandArg2}$ Additional argument corresponding to the specified command.

Value Returned

t Returns `t` if the information is deleted.

nil Returns `nil` if there is an error.

Examples

dele$te( ‘save ’ ) => t$

Deletes all the saves.

dele$te( ‘save ’v ’ ) => t$
Deletes *only* the nets. The rest of the information can be saved in subsequent simulations.

delete( 'save "net23"' ) => t

Deletes only *net23*. The rest of the information can be saved in subsequent simulations.
**design**

design( t_cktFile ) => t_cktFile/nil  
design( t_lib t_cell t_view ) => (t_lib t_cell t_view)/nil

**Description**

Specifies the name of the design to be simulated.

**Note:** You can use the lib, cell, view version of the design command only if you are running OCEAN within icms, msfb, or icfb. You cannot use this version of the command within the OCEAN process itself.

**Arguments**

- **t_cktFile**
  
  For the Affirma analog circuit simulator, the name of the netlist. The name must end in netlist. Note that the netlistHeader and netlistFooter files are also needed in the same directory.

  For socket simulators (cdsSpice, spectreS) this is the name of the raw circuit file. If generated in the Affirma analog circuit design environment, the file is named design.c and is found in the netlist directory.

  Otherwise, cktFile is a preexisting netlist file from another source. In this case, you might need to remove the .cards from the netlist because the OCEAN commands are converted to .cards and appended to the final netlist. The simulator might give an error or warning if the .cards are read twice.

- **t_lib**
  
  Name of the Affirma analog circuit design environment library that contains the design.

- **t_cell**
  
  Name of the design.

- **t_view**
  
  View of the design (typically schematic).

**Value Returned**

- **t_cktFile**
  
  Returns the name of the design if successful.
l_(  lib cell view ) Returns the name of the view for an Affirma analog circuit design environment design if successful.

nil Returns nil and prints an error message if there is a problem using the specified design.

Examples

For the Affirma analog circuit simulator,

design( "netlist" ) => simple.c

specifies that netlist, a netlist file, be used in the simulation.

For the spectreS simulator,

design( "simple.c" ) => simple.c

specifies that simple.c, a raw circuit file, be used in the simulation.

design( "tests" "simple" "schematic" ) => (tests simple schematic)

Specifies that the schematic view of the simple design from your tests library be used in the simulation.
desVar

\[
desVar( \; t_{\text{desVar1}} \; f_{\text{value1}} \; \ldots \; [t_{\text{desVarN}} \; f_{\text{valueN}}] \) \Rightarrow \text{undefined/nil}
\]

Description
Sets the values of design variables used in your design. You can set the values for as many design variables as you want.

Arguments
\[
\begin{align*}
t_{\text{desVar1}} & \quad \text{Name of the design variable.} \\
f_{\text{value1}} & \quad \text{Value for the design variable.} \\
t_{\text{desVarN}} & \quad \text{Name of an additional design variable.} \\
f_{\text{valueN}} & \quad \text{Value for the additional design variable.}
\end{align*}
\]

Value Returned
\[
\begin{align*}
\text{undefined} & \quad \text{The return value for this command/function is undefined.} \\
nil & \quad \text{Returns nil and prints an error message if the assignments fail.}
\end{align*}
\]

Examples
\[
\begin{align*}
desVar( \; "rs" \; 1k )
\end{align*}
\]
Sets the \text{rs} design variable to \text{1k}.

\[
\begin{align*}
desVar( \; "r1" \; "rs" \; "r2" \; "rs*2" )
\end{align*}
\]
Sets the \text{r1} design variable to \text{rs}, or \text{1k}, and sets the \text{r2} design variable to \text{rs*2}, or \text{2k}.

\[
\begin{align*}
a = \text{evalstring}( \; \text{desVar}( \; "rs" ) \; ) \; / \; 2
\end{align*}
\]
Sets \text{a} to \text{2k/2}, or \text{1k}.

Note: \text{evalstring} is necessary because \text{desVar} returns a string.
envOption

envOption( s_envOption1 g_value1 ... [ s_envOptionN g_valueN ] ) => undefined/nil

Description

Sets environment options.

Use the OCEAN online help command ocnHelp('envOption) to get the list of environment options. To specify an include file, use the includeFile command, not the envOption command. To set a model path, use the path command, not the envOption command.

Arguments

s_envOption1 Name of the first environment option to set.
g_value1 Value for the option.

s_envOptionN Name of an additional environment option to set.
g_valueN Value for the option.

Value Returned

undefined The return value for this command/function is undefined.
nil Returns nil if there are problems setting the option.

Examples

eenvOption('paramRangeCheckFile "./myDir/range.check"
Sets the paramRangeCheckFile environment option.
eenvOption('initFile "./myDotSFiles/init"
Sets the initFile environment option.
eenvOption('updateFile "./myDotSFiles/update"
Sets the updateFile environment option.
forcenode

forcenode( t_netName1 f_value1 ... [t_netNameN f_valueN] ) => undefined/nil

Description

Holds a node at a specified value.

Note: This is not available for the Affirma analog circuit simulator. Refer to the documentation for your simulator to see if this feature is available for your simulator.

Arguments

- **t_netName1**: Name of the net.
- **f_value1**: Voltage value for the net.
- **t_netNameN**: Name of an additional net.
- **f_valueN**: Voltage value for the net.

Value Returned

- **undefined**: The return value for this command/function is undefined.
- **nil**: Returns nil and prints an error message.

Example

forcenode( "net1" 5 "net34" 2 )

Sets the force nodes of "net1" to 5 and "net34" to 2.
ic

ic( t_netName1 f_value1 ... [t_netNameN f_valueN] ) => undefined/
nil

Description
Sets initial conditions on nets in a transient analysis.

Arguments

\textit{t_netName1} \quad \text{Name of the net.}

\textit{f_value1} \quad \text{Voltage value for the net.}

\textit{t_netNameN} \quad \text{Name of an additional net.}

\textit{f_valueN} \quad \text{Voltage value for the net.}

Value Returned

\textbf{undefined} \quad \text{The return value for this command/function is undefined.}

\textbf{nil} \quad \text{Returns nil and prints an error message.}

Example

\texttt{ic( "/net1" 5 "/net34" 2 )}

Holds the nodes of "/net1" at 5 and "/net34" at 2.
**includeFile**

`includeFile( t_fileName ) => t_fileName`

**Description**

Includes the specified file in the final netlist of the simulator for the current session.

**Note:** This command is not available for the Affirma analog circuit simulator. Use the `modelFile` or `stimulusFile` command instead.

**Note:** Using this command is comparable to using the Environment Options form of the Affirma analog circuit design environment to name an include file and specify that the syntax for the file be that of the target simulator. If you want the include file to be in Affirma Cadence-SPICE circuit simulator syntax, you must edit the raw netlist file (which has a `.c` or `.C` suffix), and manually add the include file.

**Arguments**

`t_fileName`  
Name of the file to include in the final netlist.

**Value Returned**

`t_fileName`  
Returns the name of the file if successful.

nil  
Returns nil and prints an error message otherwise.

**Example**

`includeFile( "~/projects/nmos" ) => "~/projects/nmos"`

Includes the `nmos` file in the final netlist of the simulator for the current session.
modelFile

modelFile( [g_modelFile1 [g_modelFile2 ...]] ) => l_modelFile

Description

Specifies model files to be included in the simulator input file. This command returns the model files used. When model files are specified through the arguments, the model files are set accordingly. Use of full paths for the model file is recommended.

Arguments

- g_modelFile1
  - This argument can be a string to specify the name of the model file.

- g_modelFile2
  - This argument can be a list of two strings to specify the name of the model file and the name of the section.

Value Returned

- l_modelFile
  - A list of all the model file/section pairs.

- nil
  - Returned when no file section pairs have been specified with the current call or a previous call of this command. The nil value is also returned when an error has been encountered.

Example

modelFile( "bjt.scs" "nmos.scs" ) => ( ("bjt.scs" "") ("nmos.scs" "") )
modelFile( "bjt.scs" '("nmos.scs" "typ") 'my_models ) => ( ("bjt.scs" "") ("nmos.scs" "typ") ("my_models" "") )
nodeset

nodeset( t_netName1 f_value1 ... [t_netNameN f_valueN]) => undefined/nil

Description
Sets the initial estimate for nets in a DC analysis, or sets the initial condition calculation for a transient analysis.

Arguments

- **t_netName1**: Name of the net.
- **f_value1**: Voltage value for the net.
- **t_netNameN**: Name of an additional net.
- **f_valueN**: Voltage value for the net.

Value Returned

- **undefined**: The return value for this command/function is undefined.
- **nil**: Returns nil and prints an error message otherwise.

Example

nodeset( "net1" 5 "net34" 2 )

Sets the initial estimates of "net1" to 5 and "net34" to 2.
noise

noise( t_output t_source )=> undefined/nil

Description
Specifies a noise analysis.

Note: This command cannot be used with the Affirma analog circuit simulator.

Arguments

  t_output      Output node.
  t_source      Input source.

Value Returned

  undefined    The return value for this command/function is undefined.
  nil          Returns nil and prints an error message if there is a problem specifying the analysis.

Example

noise( "n1" "v1" )

Specifies a noise analysis.
ocnDisplay

\[
\text{ocnDisplay}([\text{?output}\ t\_filename\ |\ p\_port]\ s\_command}
  \quad [g\_commandArg1]\ [g\_commandArg2] \ ... \ ) \Rightarrow t/\text{nil}
\]

Description

Displays all the information specified.

The \textit{s\_command} argument specifies the command whose information you want to display. If you include only this argument, all the information for the command displays. If you supply subsequent arguments, only those particular pieces of information display as opposed to displaying all the information for that command. If you provide a filename as the \textit{?output} argument, the \textit{ocnDisplay} command opens the file and writes the information to it. If you provide a port (the return value of the SKILL \textit{outfile} command), the \textit{ocnDisplay} command appends the information to the file that is represented by the port.

Arguments

- \textit{t\_filename}:
  File in which to write the information. The \textit{ocnDisplay} command opens the file, writes to the file, then closes the file. If you specify the filename without a path, the \textit{ocnDisplay} command creates the file in the directory pointed to by your Skill Path. To find out what your Skill path is, type \texttt{getSkillPath()} at the OCEAN prompt.

- \textit{p\_port}:
  Port (previously opened with \textit{outfile}) through which to append the information to a file. You are responsible for closing the port. See the \textit{outfile} command for more information.

- \textit{s\_command}:
  Command that was initially used to add the items that are now being displayed.
  Valid values: Most simulation setup commands

- \textit{g\_commandArg1}:
  Argument corresponding to the specified command.

- \textit{g\_commandArg2}:
  Additional argument corresponding to the specified command.

Value Returned

- \textit{t}:
  Displays the information and returns \textit{t}.
nil          Returns nil and prints an error message if there are problems displaying the information.

Examples

ocnDisplay( 'optimizeGoal ) => t
Displays all the optimizeGoal information.

ocnDisplay( 'analysis 'tran ) => t
Displays only transient analyses.

ocnDisplay( 'save ) => t
Displays all the keeps.

ocnDisplay( ?output myPort 'analyis ) => t
Displays and writes all the analyses to the port named myPort.
off

off( s_command [g_commandArg1] [g_commandArg2] ... ) => t/nil

Description

Turns off the specified information.

The first argument specifies the command whose information you want to turn off. If you include only this first argument, all the information for the command is turned off. If you supply subsequent arguments, only those particular pieces of information are turned off as opposed to turning off all the information for that command. The information is not deleted and can be used again.

Arguments

s_command Command that was initially used to add the items that are now being turned off.
Valid value: restore

g_commandArg1 Argument corresponding to the specified command.

g_commandArg2 Additional argument corresponding to the specified command.

Value Returned

t Returns t if the information is turned off.

nil Returns nil and prints an error message if there are problems turning off the information.

Examples

off( 'restore ) => t

Turns off the restore command.

off( restore 'tran ) => t

Turns off the transient restore command.
option

option( [?categ s_categ] s_option1 g_value1 [s_option2 g_value2] … ) => undefined/nil

Description
Specifies the values for built-in simulator options. You can specify values for as many options as you want.

Arguments

s_categ
Type of simulator to be used.
Valid values: analog if the options are for an analog simulator, digital for a digital simulator, or mixed for a mixed-signal simulator
Default value: analog

s_option1
Name of the simulator option.

g_value1
Value for the option.

s_option2
Name of an additional simulator option.

g_value2
Value for the option.

Value Returned

undefined
The return value for this command/function is undefined.

nil
Returns nil and prints an error message if there are problems setting the option.

Examples

option( 'abstol 1e-10 )
Sets the abstol option to 1e-10.

option( 'delmax 50n )
Sets the delmax option to 50n.
path

path( t_dirName1 ... [t_dirNameN]) => l_pathList/nil

Description
Sets the search path for included files.

Using this command is comparable to setting the Include Path for the Affirma analog circuit simulator, or the modelPath for socket simulators in the Affirma analog circuit design environment user interface. You can add as many paths as you want with this command.

Note: For the Affirma analog circuit simulator, the modelFile command is preferred over this command.

Arguments

t_dirName1 Directory path.

t_dirNameN Additional directory path.

Value Returned

l_pathList Returns the entire list of search paths specified.

nil Returns nil and prints an error message if the paths cannot be set.

Examples

path( "~/models" "/tmp/models" ) => "~/models" "/tmp/models"

Specifies that the search path includes /models followed by /tmp/models.
prependPath

prependPath( t_dirName1 ... [t_dirNameN]) => undefined/nil

Description

Adds a new path to the beginning of the search path list. You can add as many paths as you want with this command.

Note: For the Affirma analog circuit simulator, use of the modelFile command is preferred over the use of this command.

Arguments

<table>
<thead>
<tr>
<th>t_dirName1</th>
<th>Directory path.</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_dirNameN</td>
<td>Additional directory path.</td>
</tr>
</tbody>
</table>

Value Returned

<table>
<thead>
<tr>
<th>undefined</th>
<th>The return value for this command/function is undefined.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>Returns nil and prints an error message if the paths cannot be added.</td>
</tr>
</tbody>
</table>

Examples

prependPath( "/usr/mnt/user/processB/models" ) => "/usr/mnt/user/processB/models"

Adds /usr/mnt/user/processB/models to the beginning of the search path list.
restore

restore( s_analysisType t_filename ) => undefined/nil

Description

Tells the simulator to restore the state previously saved to a file with a store command.

This command is not available for the Affirma analog circuit simulator. Use the powerful store/restore options (readns, readforce, write, or writefinal) on all analyses.

Note: Restore is not available for all simulators.

Arguments

s_analysisType Type of the analysis.
Valid values: dc or tran

t_filename Name of the file containing the saved state.

Value Returned

undefined The return value for this command/function is undefined.

nil Returns nil and prints an error message if there are problems restoring the information.

Examples

restore( 'dc "./storeFile" ) => ./storeFile

Initializes the simulator to the state saved in the storeFile file.

restore( 'tran "./tranStoreFile" ) => ./tranStoreFile

Initializes the simulator to the state of a transient analysis saved in the tranStoreFile file.
resultsDir

resultsDir( t_dirName ) => undefined/nil

Description

Specifies the directory where the PSF files (results) are stored.

If you do not specify a directory with this command, the PSF files are placed in ../psf to the netlist directory.

Note: The directory you specify with resultsDir is also where the simulator.out file is created.

Note: Some simulators are designed to always put their results in a specific location. For these simulators, resultsDir has no effect. You might use this command when you want to run several simulations using the same design and want to store each set of results in a different location. If this command is not used, the results of an analysis are overwritten with each simulation run.

Arguments

 t_dirName Directory where the PSF files are to be stored.

Value Returned

 undefined The return value for this command/function is undefined.

 nil Returns nil and prints an error message if there is a problem with that directory.

Example

resultsDir("~/simulation/ckt/spectreS/schematic/psf")=>
 "~/simulation/ckt/spectreS/schematic/psf"

Specifies the psf directory as the directory in which to store the PSF files.
run

run( [analysisList] [?jobName t_jobName] [?host t_hostName] [?queue t_queueName] [?startTime t_startTime] [?termTime t_termTime] [?dependentOn t_dependentOn] [?mail t_mailingList] [?block s_block] [?notify s_notifyFlag] ) => s_jobName/nil
run( )=>t_dirName/nil
run(s_analysisType1 ... s_analysisTypeN)=> t_dirName/nil

Description

Starts the simulation or specifies a time after which an analysis should start.

If distributed processing is not available on the system or is not enabled, parameters specific to distributed processing (such as host, job name, and queue) are ignored and the simulation runs locally. If distributed processing is available and is enabled, the environment default values are used if not specified in the run command arguments. The environmental default values are stored in the .cdsenv file.

Do not use the run command to start the following kinds of analyses. Instead, use the command that is specific to the analysis.

<table>
<thead>
<tr>
<th>To start</th>
<th>Use this command</th>
<th>Described in</th>
</tr>
</thead>
<tbody>
<tr>
<td>parametric analyses</td>
<td>paramRun</td>
<td>paramRun</td>
</tr>
<tr>
<td>corners analyses</td>
<td>cornerRun</td>
<td>cornerRun</td>
</tr>
<tr>
<td>Monte Carlo analyses</td>
<td>monteRun</td>
<td>monteRun</td>
</tr>
<tr>
<td>optimizations</td>
<td>optimizeRun</td>
<td>optimizeRun</td>
</tr>
</tbody>
</table>

Arguments

*analysisList* List of analyses to be run with the run command.

**Note:** The following arguments apply only when distributed mode is enabled.

*t_jobName* If the name given is not unique, an integer is appended to create a unique job name.

*t_hostName* Name of the host on which to run the analysis. If no host is specified, the system assigns the job to an available host.
### Simulation Commands

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t_queueName</code></td>
<td>Name of the queue. If no queue is defined, the analysis is placed in the default queue.</td>
</tr>
<tr>
<td><code>t_startTime</code></td>
<td>Desired start time for the job. If dependencies are specified, the job does not start until all dependencies are satisfied.</td>
</tr>
<tr>
<td><code>t_termTime</code></td>
<td>Termination time for job. If the job has not completed by the specified termination time, the job is aborted.</td>
</tr>
<tr>
<td><code>t_dependentOn</code></td>
<td>List of jobs on which the specified job is dependent. The job is not started until dependent jobs are completed.</td>
</tr>
<tr>
<td><code>t_mailingList</code></td>
<td>List of users to be notified when the analysis is complete.</td>
</tr>
<tr>
<td><code>s_block</code></td>
<td>When <code>s_block</code> is not set to nil, the OCEAN script halts until the job is complete. Default value: nil</td>
</tr>
<tr>
<td><code>s_notifyFlag</code></td>
<td>When not set to nil, the job completion message is echoed to the OCEAN interactive window. Default value: t</td>
</tr>
<tr>
<td><code>s_analysisType1</code></td>
<td>Name of a prespecified analysis to be simulated.</td>
</tr>
<tr>
<td><code>s_analysisTypeN</code></td>
<td>Name of another prespecified analysis to be simulated.</td>
</tr>
</tbody>
</table>

#### Value Returned

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s_jobName</code></td>
<td>Returns the job name of the job submitted. The job name is based on the <code>jobName</code> argument. If the job name submitted is not unique, a unique identifier is appended to the job name. This value is returned for nonblocking distributed mode.</td>
</tr>
<tr>
<td><code>t_dirName</code></td>
<td>Returns the name of the directory in which the results are stored. This value is returned for local and blocking distributed modes.</td>
</tr>
<tr>
<td><code>nil</code></td>
<td>Returns nil and prints an error message if there is an error in the simulation. In this case, look at the <code>yourSimulator.out</code> file for more information. (This file is typically located in the <code>psf</code> directory.)</td>
</tr>
</tbody>
</table>
Examples

run( ) => t

Starts the simulation.

run(’tran, ’ac)

Runs only the tran and ac analyses.

run(’dc)

Runs only the dc analysis.

run( ?job "reconFilter" ?noBlock “t”) => ’reconFilter

Returns a job name of reconFilter for the specified job and runs that job if distributed processing is enabled. The job is submitted nonblocking. The actual job name is returned.

run( ?queue "fast" )

Submits the current design and enabled analyses as a job on the fast queue, assuming that distributed processing is available and enabled.
save

save( [?categ s_categ] s_saveType [t_saveName1] … [t_saveNameN] ) => undefined/nil

Description

Specifies the outputs to be saved and printed during simulation.

When specifying particular outputs with saveName, you can include as many outputs as you want. If you want to turn off the default of save,’allv, use the delete( ’save ) command.

Arguments

s_categ                     Type of simulator to be used.
Valid values: analog, digital
Default value: analog

Note: digital is not available.

s_saveType                  Type of outputs to be saved.
Valid values:

v  Specifies that a list of subsequent net names be kept.

i  Specifies that a list of subsequent currents be kept.

all  Specifies that all nets and all currents are to be saved.

allv  Specifies that all voltages are to be saved.

alli  Specifies that all currents are to be saved.

Default value: allv

t_saveName1                  Name of the net, device, or other object.

t_saveNameN                  Name of another net, device, or object.

Value Returned

undefined                   The return value for this command/function is undefined.
nil

Returns nil and prints an error message if there is a problem keeping the outputs.

Examples

save( 'v "net34" "net45" )

Saves the outputs for net34 and net45.

save( 'i "R1" "/Q1/b" )

Saves the currents for R1 and Q1/b.

save( 'all )

Saves all the nets and currents.

save( 'i "q1:b" "r1:p" "mn1:d" )

For the Affirma analog circuit simulator, saves the current through the specified devices.

save( 'i "i(q1,b)" "i(r1)" "i(mn1,d)" )

For the Affirma Cadence-SPICE circuit simulator, saves the current through the same devices.
saveOption

saveOption([s_option1 g_optionValue1]…[s_optionN
g_optionValueN]) => undefined/nil

Description

Specifies save options to be used by the simulator.

You can include as many save options as you want. To include a save option, replace s_option1 with the name of the desired save option and include another argument to specify the value for the option.

When you use the saveOption command without specifying any arguments, the command returns a list of option and value pairs.

Save options vary, depending on the simulator and interface that you are using. If you are using the Affirma analog circuit simulator, for example, you can type the following at an OCEAN prompt to see which options you can set with the saveOption command:

simulator('spectre)
ocnHelp('saveOption)

See the Affirma Spectre Circuit Simulator User Guide for more information on these options.

Note: The saveOption command does not work with socket simulators. If you are using a socket simulator, you must instead specify save options with the save command described in “save” on page 101.

Arguments

s_option1

Save option. The save options that are available depend on which simulator you use. (See the documentation for your simulator.)

g_optionValue1

Value for the save option.

s_optionN

Any subsequent save option. The save options that are available
depend on which simulator you use. (See the documentation for your simulator.)

\texttt{g\_optionValueN}

Value for the save option.

**Value Returned**

\texttt{undefined}

The return value for this command/function is undefined.

\texttt{nil}

Returns \texttt{nil} if there are problems specifying options.

**Example**

\texttt{saveOption( \textquote{save "lvl" \textquote{nestlvl 10 \textquote{currents "selected" \textquote{useprobes "yes" \textquote{subcktprobelvl 2 ?saveahdlvars "all"}}}}})}
**simulator**

`simulator( s_simulator ) => s_simulator/nil`

**Description**

Starts an OCEAN session and sets the simulator name for that session. The previous session (if any) is closed and all session information is cleared.

**Arguments**

`s_simulator` 
Name of the simulator.

**Value Returned**

`s_simulator` 
Returns the name of the simulator.

`nil` 
Returns nil and prints an error message if the simulator is not registered with the Affirma analog circuit design environment through OASIS. If the simulator is not registered, the simulator from the preceding session is retained.

**Examples**

`simulator( 'spectre ) => spectre`

Specifies that the Affirma analog circuit simulator be used for the session.

`simulator( 'spectreverilog ) => spectreverilog`

Specifies that spectreVerilog be used for the session.
stimulusFile

stimulusFile( t_fileName [t_fileName2 ... t_fileNameN ] [?xlate
       b_xlate] ) => l_fileNames/nil

Description

Specifies stimulus files to be used by the simulator.

When the b_xlate variable is set to t, the schematic net expressions [#net] and instance
name expression [$instance] in the stimulus file are mapped into simulator names before
including. When a netlist is specified as the design, this option must be set to nil.

Note: This command does not work with socket simulators.

Arguments

  t_fileName           The name of the stimulus file to be included.

  t_fileName2...t_fileNameN
     The names of the additional stimulus files to be included.

  b_xlate
     If set to t, net and instance expressions are translated to
     simulator names. The default value of the b_xlate variable is
     t.

Value Returned

  l_fileNames
     A list of the stimulus file names is output if the command is
     successful.

  nil
     Otherwise nil is returned

Example

stimulusFile( "tran.stimulus rf.stimulus" ?xlate nil) =>
("tran.stimulus rf.stimulus")

Includes tran.stimulus and rf.stimulus in the simulator input file. No net and instance
expressions are translated.
store

store( s_analysisType t_filename ) => t_filename/nil

Description

Requests that the simulator store its node voltages to a file.

You can restore this file in a subsequent simulation to help with convergence or to specify a certain starting point. This command is not available for the Affirma analog circuit simulator. Use the powerful store/restore options (readns, readforce, write, or writefinal) on all analyses instead.

Note: store is not available for all simulators.

Arguments

s_analysisType Type of the analysis.

Valid values: dc or tran

t_filename Name of the file in which to store the simulator’s node voltages.

Value Returned

t_filename Returns the filename.

nil Returns nil and prints an error message if there are problems storing the information to a file.

Examples

store( ’dc "/storeFile" ) => ./storefile

Stores the simulator’s node voltages in a file named storeFile in the current directory.

store( ’tran "/tranStoreFile" ) => ./transtorefile

Stores the node voltages for a transient analysis in a file named tranStoreFile in the netlist (design) directory unless a full path is specified.
temp

\texttt{temp( f\textunderscore tempValue )} => \texttt{f\_tempValue/nil}

\textbf{Description}

Specifies the circuit temperature.

\textbf{Arguments}

\texttt{f\_tempValue} \hspace{1cm} Temperature for the circuit.

\textbf{Value Returned}

\texttt{f\_tempValue} \hspace{1cm} Returns the temperature specified.

\texttt{nil} \hspace{1cm} Returns \texttt{nil} and prints an error message if there are problems setting the temperature.

\textbf{Example}

\texttt{temp( 125 )} => 125

Sets the circuit temperature to 125.
tran

tran( g_fromValue g_toValue g_byValue ) => g_byValue/nil
tran( g_toValue) => undefined/nil

Note: The second instance of the tran command is valid only with the Affirma analog circuit simulator.

Description

Specifies a transient analysis with limited options. If other analysis options are needed, use the analysis command.

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_fromValue</td>
<td>Starting time for the analysis.</td>
</tr>
<tr>
<td>g_toValue</td>
<td>Ending time.</td>
</tr>
<tr>
<td>g_byValue</td>
<td>Increment at which to step through the analysis.</td>
</tr>
</tbody>
</table>

Value Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined</td>
<td>The return value for this command/function is undefined.</td>
</tr>
<tr>
<td>nil</td>
<td>Returns nil and prints an error message if the analysis is not specified.</td>
</tr>
</tbody>
</table>

Examples

tran( 1u)

Specifies a transient analysis to 1u for the Affirma analog circuit simulator

tran( 0 1u 1n ) => 1e-09

Specifies a transient analysis from 0 to 1u by increments of 1n.
Data Access Commands

The data access commands let you open results and select different types of results to analyze. You can get the names and values of signals and components in the selected results, and you can print different types of reports.

In this chapter, you can find information on the following data access commands:

- `dataType` on page 113
- `getData` on page 114
- `i` on page 116
- `noiseSummary` on page 118
- `ocnHelp` on page 121
- `ocnPrint` on page 123
- `openResults` on page 125
- `outputParams` on page 127
- `outputs` on page 128
- `phaseNoise` on page 129
- `pv` on page 130
- `report` on page 132
- `resultParam` on page 135
- `results` on page 137
- `selectResult` on page 138
sp on page 140
sweepNames on page 141
sweepValues on page 142
v on page 143
vswr on page 145
zm on page 146
zref on page 147
dataTypes

dataTypes() => l_dataTypes/nil

Description

Returns the list of data types that are used in an analysis previously specified with selectResult.

Arguments

None.

Value Returned

l_dataTypes Returns the list of data types.
nil Returns nil and an error message if the list of datatypes cannot be returned.

Example

selectResult('dcOp)
dataTypes() => ( "node" "vs" "resistor" "bjt" )

Returns the data types used in the selected file, in this case, dcOp.
getData

dataGet( t_name ) => x_number/o_waveform/nil

Description

Returns the number or waveform for the signal name specified.

The type of value returned depends on how the command is used.

Arguments

\( t\_name \)  
Name of the signal.

Value Returned

\( x\_number \)  
Returns an integer simulation result.

\( o\_waveform \)  
Returns a waveform object. A waveform object represents simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \( \text{drwave:XXXXX} \).)

\( \text{nil} \)  
Returns \( \text{nil} \) and an error message if the value cannot be returned.

Examples

getData( "/net6" ) => drwave:25178234

Returns the number or waveform for net6. In this example, the return value is equivalent to v( "/net6" ).

dataGet( "/V1" ) => drwave:96879364

Returns the number or waveform for V1. In this example, the return value is equivalent to i( "/V1" ).

selectResult( 'tran ) =>

ocnPrint( getData( "net1" ) ) =>
In this example, the `getData( "net1" )` command passes a waveform to the `ocnPrint` command. The `ocnPrint` command then prints the data for the waveform. In this example, the return value is equivalent to `(v( "net1" ))`. 
i

\[
i( \text{t\_component} \ [?\text{resultsDir} \ \text{t\_resultsDir}] [?\text{result} \ \text{s\_resultName}]) \Rightarrow \text{o\_waveform/nil}
\]

**Description**

Returns the current through the specified component.

If you specify a directory with `resultsDir`, it is equivalent to temporarily using the `openResults` command on that directory. The `i` command operates on the results in that directory, and then resets `openResults` to its previous setting. If you specify particular results with `resultName`, it is equivalent to using the `selectResult` command on the specified results. The `i` command returns the voltage of the net for those results and resets the `selectResult` command to its previous setting.

If you specify a directory using `resultsDir`, you must also specify particular results with `resultName`.

**Arguments**

- **t\_component**: Name of the component.
- **t\_resultsDir**: Directory containing the PSF files (results). If you supply this argument, you must also supply the `resultName` argument.
- **s\_resultName**: Results from an analysis.

**Value Returned**

- **o\_waveform**: Returns a waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`).
- **nil**: Returns an error message and `nil` if there is a problem.

**Examples**

```
selectResult( 'tran')
i( "/R1") =>
```

Returns the current through the `R1` component.
ocnPrint( i( "/R1" ) ) =>

Prints the current through the R1 component.

ocnPrint( i( "/R1" ?result 'dc ') ) =>

Prints the current through the R1 component with respect to the dc swept component.

ocnPrint( i( "/R1" ?resultsDir "./test2/psf" ?result 'dc ') ) =>

Prints the current through the R1 component with respect to dc for the results from a different run (stored in test2/psf).
noiseSummary

noiseSummary(s_type [resultsDir t_resultsDir] [resultName] [frequency f_frequency] [weight f_weight] [output t_fileName p_port] [noiseUnit t_noiseUnit] [truncateData x_truncateData] [truncateType s_truncateType] [digits x_digits] [percentDecimals x_percentDecimals] [from f_from] [to f_to] [deviceType ls_deviceType]) => t_fileName/p_port/nil

Description

Prints a report showing the noise contribution of each component in a circuit.

If you specify a directory with resultsDir, it is equivalent to temporarily using the openResults command. The noiseSummary command prints the results for that directory and resets the openResults command to its previous setting. If you specify particular results with resultName, it is equivalent to temporarily using the selectResult command on the specified results. The noiseSummary command prints the results and resets the selectResult command to its previous setting.

Note: If you specify a directory using resultsDir, you must also specify particular results with resultName.

Arguments

s_type Type of noise-analysis results for which to print the report. Valid values: spot, to specify noise at a particular frequency, or integrated, to specify noise integrated over a frequency range.

t_resultsDir The directory containing the noise-analysis results.

S_resultName Results from an analysis for which you want to print the noiseSummary report.

f_frequency Frequency value of interest.

f_weight Waveform representing the function with which the integral is weighted. Default value: 1.0

t_fileName File in which to write the information. The noiseSummary command opens the file, writes to the file, and closes the file. If
you specify the filename without a path, the noiseSummary command creates the file in the directory pointed to by your Skill Path. To find out what your Skill path is, type getSkillPath() at the OCEAN prompt.

**p_port**

Port (previously opened with outfile) through which to append the information to a file. You are responsible for closing the port. See the outfile command for more information.

**t_noiseUnit**

Specifies the type of noise unit to be saved.

Valid values: "V^2" for v^2/Hz or "V" for v/sqrt( Hz )

**x_truncateData**

Specifies a number that the truncateType argument uses to define the components for which information is to be printed.

**s_truncateType**

Specifies the method that is used to limit the data being included in the report.

<table>
<thead>
<tr>
<th>Valid Values</th>
<th>Description</th>
<th>Sample Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>'top'</td>
<td>Saves information for the number of components specified with truncateData. The components with the highest contributions are saved.</td>
<td>10</td>
</tr>
<tr>
<td>'level'</td>
<td>Saves information for all the components that have a lower contribution than the group of top contributors specified with truncateData.</td>
<td>10u</td>
</tr>
<tr>
<td>'relative'</td>
<td>First saves information for all the components that have a lower contribution than the group of top contributors specified with truncateData, then multiplies each of these contributions times the value of the top contributor.</td>
<td>.1</td>
</tr>
<tr>
<td>'none'</td>
<td>Saves information for all the components.</td>
<td></td>
</tr>
</tbody>
</table>
x_digits
Number of significant digits with which the contributors are printed.

x_percentDecimals
Number of decimals printed for any relative contribution.

f_from
For integrated noise, the start value for frequency.

f_to
For integrated noise, the end value for frequency.

ls_deviceType
List of device type strings to be included.
Valid values: a list of strings or ‘all

Value Returned

 t_fileName
Returns the name of the port.

 p_port
Returns the name of the file.

 nil
Returns nil and an error message if the summary cannot be printed.

Examples

noiseSummary( ‘integrated ?result ‘noiseSweep-noise )
Prints a report for an integrated noise analysis.

noiseSummary( ‘integrated ?resultsDir
  ”/usr/simulation/lowpass/spectre/schematic"
  ?result ‘noise)
Prints a report for an integrated noise analysis for the results from a different run (stored in the schematic directory).

noiseSummary( ‘spot ?resultsDir
  ”/usr/simulation/lowpass/spectre/schematic"
  ?result ‘noise ?frequency 100M )
Prints a report for a spot noise analysis at a frequency of 100M.
ocnHelp

ocnHelp( [?output t_filename | p_port][s_command] ) => t/nil

Description

Provides online help for the specified command.

If no command is specified, provides information about how to use help and provides the different categories of information contained in the help library. If you provide a filename as the ?output argument, the ocnHelp command opens the file and writes the information to it. If you provide a port (the return value of the SKILL outfile command), the ocnHelp command appends the information to the file that is represented by the port. If you do not specify ?output, the output goes to standard out (stdout).

Arguments

- **t_filename**: File in which to write the information. The ocnHelp command opens the file, writes to the file, and closes the file. If you specify the filename without a path, the ocnHelp command creates the file in the directory pointed to by your Skill Path. To find out what your Skill path is, type getSkillPath() at the OCEAN prompt.

- **p_port**: Port (previously opened with outfile) through which to append the information to a file. You are responsible for closing the port. See the outfile command for more information.

- **s_command**: Command for which you want help.

Value Returned

- **t**: Displays the online help and returns t.

- **nil**: Returns nil and an error message if help cannot be displayed.

Examples

ocnHelp() => t

Displays information about using online help.

ocnHelp( ‘analysis ) => t
Displays help for the `analysis` command.

```ocnHelp( ?output "helpInfo" ) => t```

Writes information about using online help to a file named `helpInfo`. 
ocnPrint

ocnPrint( [?output t_filename | p_port] [?precision x_precision]  
[?numberNotation s_numberNotation] [?numSpaces x_numSpaces] 
[?width x_width] o_waveform1 [o_waveform2 ...] ) => t/nil

Description

Prints the text data of the waveforms specified in the list of waveforms.

If you provide a filename as the ?output argument, the ocnPrint command opens the file and writes the information to it. If you provide a port (the return value of the SKILL outfile command), the ocnPrint command appends the information to the file that is represented by the port.

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_filename</td>
<td>File in which to write the information. The ocnPrint command opens the file, writes to the file, and closes the file. If you specify the filename without a path, the OCEAN environment creates the file in the directory pointed to by your Skill Path. To find out what your Skill path is, type getSkillPath() at the OCEAN prompt.</td>
</tr>
<tr>
<td>p_port</td>
<td>Port (previously opened with outfile) through which to append the information to a file. You are responsible for closing the port. See the outfile command for more information.</td>
</tr>
<tr>
<td>x_precision</td>
<td>The number of significant digits to print. This value overrides any global precision value set with the setup command. Valid values: 1 through 16 Default value: 6</td>
</tr>
<tr>
<td>s_numberNotation</td>
<td>The notation for printed information. This value overrides any global format value set with the setup command. Valid values: ‘suffix, ‘engineering, ‘scientific, ‘none Default value: ‘suffix</td>
</tr>
</tbody>
</table>

The format for each value is ‘suffix: 1m, 1u, 1n, etc.; ‘engineering: 1e-3, 1e-6, 1e-9, etc.; ‘scientific: 1.0e-2, 1.768e-5, etc.; ‘none.
The value ‘none is provided so that you can turn off formatting and therefore greatly speed up printing for large data files. For the fastest printing, use the ‘none value and set the ?output argument to a filename or a port, so that output does not go to the CIW.

\textit{x\_numSpaces}  

The number of spaces between columns.  
Valid values: 1 or greater  
Default value: 4

\textit{x\_width}  

The width of each column.  
Valid values: 4 or greater  
Default value: 14

\textit{o\_waveform1}  

Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

\textit{o\_waveform2}  

Additional waveform object.

\textbf{Value Returned}  

\texttt{t}  

Returns \texttt{t} if the text for the waveforms is printed.

\texttt{nil}  

Returns \texttt{nil} and an error message if the text for the waveforms cannot be printed.

\textbf{Examples}  

\begin{verbatim}
o cnPrint( v( "/net56" ) ) => t

Prints the text for the waveform for the voltage of net56.

ocnPrint( vm( "/net56" ) vp( "/net56" ) ) => t

Prints the text for the waveforms for the magnitude of the voltage of net56 and the phase of the voltage of net56.

ocnPrint( ?output "myFile" v( "net55" ) ) => t

Prints the text for the specified waveform to a file named myFile.
\end{verbatim}
openResults

openResults( s_jobName | t_dirName ) => t_dirName/nil

Description

Opens simulation results stored in PSF files or opens the results from a specified job, depending on which parameter is called.

When openResults passes a symbol, it interprets the value as a job name and opens the results for the specified job. s_jobName is a job name and is defined when a run command is issued.

When openResults passes a text string, it opens simulation results stored in PSF files in the specified directory. The results must have been created by a previous simulation run through OCEAN or the Affirma™ analog circuit design environment. The directory must contain a file called logFile and might contain a file called runObjFile. When you perform tasks in the design environment, the runObjFile is created. Otherwise, only logFile is created.

If you want to find out which results are currently open, you can use openResults with no argument. The directory for the results that are currently open is returned.

Note: If you run a successful simulation with distributed processing disabled, the results are automatically opened for you. Also, a job name is generated by every analysis, even if distributed processing is not enabled.

Arguments

s_jobName

The name of a distributed process job. s_jobName is a job name and is defined when a run command is issued.

t_dirName

The directory containing the PSF files.

Value Returned

t_dirName

The directory containing the PSF files.

nil

Returns nil and an error message if there are problems opening the results.
**Examples**

`openResults( "./simulation/opamp/spectre/schematic/psf" ) => psf`

Opens the results in the `psf` directory within the specified path.

`openResults( "./psf" ) => psf`

Opens the results in the `psf` directory in the current working directory.
outputParams

outputParams( t_compType ) => l_outputParams/nil

Description

Returns the list of output parameters for the specified component.

You can use the dataTypes() command to get the list of components for a particular set of results.

Note: You can use any of the parameters in outputParams as the second argument to the pv command.

Arguments

\[ t\_compType \]
Name of a component in results previously selected with the selectResult command.

Value Returned

\[ l\_outputParams \]
Returns the list of parameters.

\[ \text{nil} \]
Returns nil and an error message if there are no associated parameters or if the specified component (compType) does not exist.

Example

selectResult( 'dcOp )
dataTypes() => ( "node" "vs" "resistor" "bjt" )
outputParams( "bjt" )

Selects the dcOp results, returns the list of components for these results, and returns the list of output parameters for the bjt component.
**outputs**

outputs( [?result S_resultName] ) => l_outputs/nil

**Description**

Returns the names of the outputs whose results are stored for an analysis previously specified with `selectResult`. You can plot these outputs or use them in calculations.

**Arguments**

*S_resultName*  
Name of the result from which to retrieve the outputs information.  
Default value: The current result selected with the `selectResult` command.

**Value Returned**

*l_outputs*  
Returns the list of outputs.  

*nil*  
Returns *nil* and an error message if there are problems returning the names of the stored outputs.

**Example**

`outputs() => ( "net13" "net16" "net18" )`

Returns the names of the outputs for the PSF file selected with `selectResult`. 
phaseNoise

phaseNoise( g_harmonic S_signalResultName [?result
       S_noiseResultName] [?resultsDir t_resultsDir] ) =>
   o_waveform/nil

Description

Returns the phase noise waveform from the selected result data.

Arguments

g_harmonic List of harmonic frequencies.

S_signalResultName Name of the result that stores the signal waveform. Use the
       results() command to obtain the list results. For the Affirma
       analog circuit simulator, this argument is pss-fd.pss.

S_noiseResultName Name of the result that stores the noise waveforms. The default
       is the current result selected with the selectResult command.

t_resultsDir The directory that contains the results. The default is the current
       results directory opened with the openResults command.

Value Returned

o_waveform Waveform representing the phase noise.

nil Returns nil if there is an error.

Example

plot(phaseNoise(0 "pss-fd.pss"))
pv

pv( t_name t_param [?resultsDir t_resultsDir][?result S_resultName]) => g_value/nil

Description

Returns the value for the specified component parameter. You can use the outputParams command to get the list of parameters for a particular component.

If you specify a directory with ?resultsDir, it is equivalent to temporarily using the openResults command on that directory. The pv command operates on the results in that directory and then resets openResults to its previous setting. If you specify particular results with ?result, it is equivalent to temporarily using the selectResult command on the specified results. The pv command returns the voltage of the net for those results and resets the selectResult command to its previous setting.

Note: If you specify a directory using resultsDir, you must also specify particular results with resultName. However, you can specify particular results without specifying a directory. In this case, you must have previously selected results with the openResults command.

Arguments

t_name Name of the node or component.
t_param Name of the parameter.
t_resultsDir Directory containing the PSF files (results). If you supply this argument, you must also supply the resultName argument.
S_resultName Name of datatypes for the particular analysis you want.

Value Returned

g_value Returns the requested parameter value.
nil Returns nil and prints an error message.

Examples

selectResult('dcOp')
pv( "/Q19" "ib" ) =>

For the Q19 component, returns the value of the ib parameter.

pv( "/Q19" "ib" ?resultsDir ".//test2/psf" ) =>

For the Q19 component, returns the value of the ib parameter for the results from a different run (stored in test2/psf).
**report**

```
report([?output t_filename | p_port] [?type t_type] [?name t_name]
       [?param t_param] [?format s_reportStyle] ) => t/nil
```

**Description**

Prints a report of the information contained in an analysis previously specified with `selectResult`.

You can use this command to print operating-point, model, or component information. If you provide a filename as the `?output` argument, the `report` command opens the file and writes the information to it. If you provide a port (the return value of the SKILL `outfile` command), the `report` command appends the information to the file that is represented by the port.

**Note:** You can use the `dataTypes` command to see what types of reports you can choose. For Affirma analog circuit simulator operating points, be sure to choose `dcOp` and `opBegin`.

**Arguments**

- **t_filename**
  - File in which to write the information. The `report` command opens the file, writes to the file, and closes the file. If you specify the filename without a path, the OCEAN environment creates the file in the directory pointed to by your Skill Path. To find out what your Skill path is, type `getSkillPath()` at the OCEAN prompt.

- **p_port**
  - Port (previously opened with `outfile`) through which to append the information to a file. You are responsible for closing the port. See the `outfile` command for more information.

- **t_type**
  - Type of information to print, such as all bjts.

- **t_name**
  - Name of the node or component.

- **t_param**
  - Name of the parameter to print.

- **s_reportStyle**
  - Specifies the format of the output.
    - **Valid values:** `spice` and `paramValPair`
    - **Default value:** `paramValPair`
The spice format looks like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>Param1</th>
<th>Param2</th>
<th>Param3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name1</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Name2</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Name3</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
</tbody>
</table>

The paramValPair format looks like this:

- Name1
  - Param1=value Param2=value Param3=value
- Name2
  - Param1=value Param2=value Param3=value
- Name3
  - Param1=value Param2=value Param3=value

Value Returned

t Returns t if the information is printed.
nil Returns nil and an error message if the information cannot be printed.

Examples

- selectResult( dcOp ) = > t
- report()
  Prints all the operating-point parameters.
- report( ?type "bjt" ) = > t
  Prints all the bjt operating-point parameters.
- report( ?type "bjt" ?param "ib" ) = > t
  Prints the ib parameter for all bjts.
- report( ?type "bjt" ?name "/Q1" ?param "ib" ) = > t
  Prints the ib parameter for the bjt named Q1.
report( ?output "myFile" ) => t

Prints all the operating-point parameters to a file named myFile.

report( ?output myAlreadyOpenedPort ) => t

Prints all the operating-point parameters to a port named myAlreadyOpenedPort.
resultParam

resultParam( S_propertyName [ ?result S_resultName ] [ ?resultsDir t_resultsDir ] ) => L_value/nil

Description
Returns the value of a header property from the selected result data.

Arguments

S_propertyName  Name of the parameter

S_resultName  Name of the result from which to retrieve the parameter value. The default is the current result selected with the selectResult command.

t_resultsDir  The directory that contains the results. The default is the current results directory opened with the openResults command.

Value Returned

L_value  Value of the parameter. The data type depends on the data type of the parameter.

nil  Returns nil and an error message if there are problems returning the results.

Examples

resultParam("positive output signal" ?result "pnoise.pss")
"pif"
resultParam("negative output signal" ?result "pnoise.pss")
"0"

Returns the name of the positive and negative output signals from PSS-noise analysis result. In this case, the data type of the returned value is a string.

resultParam("port1.r.value" ?result "sp")
40.0
resultParam("port2.r.value" ?result "sp")
40.0

Returns the reference impedance of the ports in a two-port network from the S-parameter analysis result. In this case, the data type of the returned value is a floating point number.
results

results() => l_results/nil

Description

Returns a list of the type of results that can be selected.

Arguments

None.

Value Returned

l_results Returns the list of result types.

nil Returns nil and an error message if there are problems returning the results.

Example

results() => ( dc tran ac )

Returns the list of results available.
selectResult

selectResult( S_resultsName [n_sweepValue]) => o_results/nil

Description

Selects the results from a particular analysis whose data you want to examine.

The argument that you supply to this command is a data type representing the particular type of analysis results you want. All subsequent data access commands use the information specified with selectResult.

Note: Refer to the results command to get the list of analysis results that you can select.

Arguments

s_resultsName Results from an analysis.
n_sweepValue The sweep value you wish to select for an analysis.

Value Returned

o_results Returns the object representing the selected results.
nil Returns nil and an error message if there are problems selecting the analysis.

Examples

selectResult( 'tran )

Selects the results for a transient analysis.
sweepValues(3.0 3.333333 3.666667 4.0 4.333333 4.666667 5.0 )
selectResult("tran" "3.333333")

The sweepValues command prints a list of sweep values.
The selectResult command selects a specific value for a transient analysis.
selectResult( 'tran )

Selects the results for a transient analysis.
paramAnalysis("supply" ?start 3 ?stop 5 ?step 1.0/3)
paramRun("supply")
selectResult({ 'tran car( sweepValues() )

Selects the data corresponding to the first parametric run.

**Note:** selectResult('tran) would select the entire family of parametric data.
sp

sp( x_iIndex x_jIndex [ ?result S_resultName ] [ ?resultsDir t_resultsDir ] ) => o_waveform/nil

Description

Returns S-parameters for N port networks.

Arguments

x_iIndex The i\text{th} index of the coefficient in the scattering matrix.
x_jIndex The j\text{th} index of the coefficient in the scattering matrix.
S_resultName Name of the result from which to retrieve the S-parameter. The default is the current result selected with the selectResult command.
t_resultsDir The directory that contains the results. The default is the current results directory opened with the openResults command.

Value Returned

o_waveform Waveform object representing the S-parameter.

nil Returns nil if there is an error.

Examples

s21 = sp(2 1)
s12 = sp(1 2)
plot(s21 s12)
sweepNames

sweepNames( [o_waveForm] ) => l_sweepName/nil

Description

Returns the names of all the sweep variables in the PSF file previously specified with selectResult. This command is particularly useful for parametric analyses.

Arguments

o_waveForm Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

Value Returned

l_sweepName Returns a list of the sweep names.

nil Returns nil and prints an error message if the sweep names cannot be returned.

Example

sweepNames() => ( "TEMPDC" "time" )

Returns a list of sweep variables for the selected results. In this case, the return values indicate that the data was swept over temperature and time.
sweepValues

sweepValues( [o_waveForm] ) => l_sweepValues/nil

Description

Returns the list of sweep values of the outermost sweep variable of either the selected results or the supplied waveform. This command is particularly useful for parametric analyses.

Arguments

o_waveForm

Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

Value Returned

l_sweepValues

Returns the list of sweep values.

nil

Returns nil and an error message if the list of sweep values cannot be returned.

Example

sweepValues() => ( -50 -15 20 55 90.0 )

Returns a list of sweep values for the selected results. In this case, the return values indicate the temperature over which the data was swept.
v

\[ v( t_{\text{net}} [?\text{resultsDir} t_{\text{resultsDir}}][?\text{result} S_{\text{resultName}}]) \Rightarrow \]
\[ o_{\text{waveform}}/\text{nil} \]

**Description**

Returns the voltage of the specified net.

If you specify a directory with `resultsDir`, it is equivalent to temporarily using the `openResults` command on that directory. The `v` command operates on the results in that directory and resets `openResults` to its previous setting. If you specify particular results with `resultName`, it is equivalent to using the `selectResult` command on the specified results. The `v` command returns the voltage of the net for those results and resets the `selectResult` command to its previous setting.

**Note:** If you specify a directory using `resultsDir`, you must also specify particular results with `resultName`. In this case, you must have previously selected results with the `openResults` command.

**Arguments**

- **t_net**
  - Name of the net.

- **t_resultsDir**
  - Directory containing the PSF files (results). If you supply this argument, you must also supply `S_resultName`.

- **S_resultName**
  - Results from an analysis.

**Value Returned**

- **o_waveform**
  - Returns a waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`).

- **nil**
  - Returns an error message and `nil` if there is a problem.

**Example**

```
selectResult(‘tran)
v( "/net56" )
```
Returns the voltage for net56.
\texttt{ocnPrint( v( "/net56" ) )}

Prints tabular information representing the voltage for net56.
\texttt{ocnPrint( v( "net5" ?result 'dc ) )}

Prints the voltage of net5 with respect to the dc swept component.
\texttt{ocnPrint( v( "net5" ?resultsDir ".test2/psf" ?result 'dc ) )}

Prints the voltage of net5 with respect to dc for the results from a different run (stored in test2/psf).
vswr

vswr( x_index [ ?result S_resultName ] [ ?resultsDir t_resultsDir ] ) => o_waveform/nil

Description

Computes the voltage standing wave ratio.

This function is a higher level wrapper for the OCEAN expression

\[ \frac{1 + \text{mag}( s( x\_index x\_index )))}{1 - \text{mag}( s( x\_index x\_index ))} \]

Arguments

x_index  
Index of the port.

S_resultName  
Name of the result from which to retrieve the S-parameters. The default is the current result selected with the selectResult command.

t_resultsDir  
The directory that contains the results. The default is the current results directory opened with the openResults command.

Value Returned

o_waveform  
Waveform object representing the voltage standing wave ratio.

nil  
Returns an error message and nil if there is a problem.

Example

plot( vswr(2) )
zm

zm( x_index [ ?result S_resultName ] [ ?resultsDir t_resultsDir ] )
=> o_waveform/nil

Description

Computes the port input impedance.

The zm function is computed in terms of the S-parameters and the reference impedance. This function is a higher level wrapper for the OCEAN expression

\[
(1 + s( x_index x_index )) / (1 - s( x_index x_index ))
* or( zref( x_index ) 50)
\]

Arguments

x_index
Index of the port.

S_resultName
Name of the result from which to retrieve the S-parameters and the reference impedance. The default is the current result selected with the selectResult command.

t_resultsDir
The directory that contains the results. The default is the current results directory opened with the openResults command.

Value Returned

o_waveform
Waveform object representing the port input impedance.

nil
Returns an error message and nil if there is a problem.

Example

plot(zm(2))
zref

zref( x_portIndex [ ?result S_resultName ] [ ?resultsDir t_resultsDir ] ) => f_impedance/nil

Description

Returns the reference impedance for an N-port network.

Arguments

x_portIndex
Index of the port.

S_resultName
Name of the result from which to retrieve the reference impedance. The default is the current result selected with the selectResult command.

t_resultsDir
The directory that contains the results. The default is the current results directory opened with the openResults command.

Value Returned

f_impedance
Reference impedance.

nil
Returns an error message and nil if there is a problem.

Example

Zref = zref(2)
Plotting Commands

This chapter contains information on the following plotting commands:

addSubwindow on page 151
addSubwindowTitle on page 152
addTitle on page 153
addWaveLabel on page 154
addWindowLabel on page 156
clearAll on page 157
clearSubwindow on page 158
currentSubwindow on page 159
currentWindow on page 160
dbCompressionPlot on page 161
deleteSubwindow on page 162
deleteWaveform on page 163
displayMode on page 164
graphicsOff on page 165
graphicsOn on page 166
hardCopy on page 167
hardCopyOptions on page 168
ip3Plot on page 170
newWindow on page 171
plot on page 172
plotStyle on page 174
removeLabel on page 175
xLimit on page 176
yLimit on page 177
addSubwindow

addSubwindow() => x_subwindowID/nil

Description

Adds a subwindow to the current Waveform window and returns the number for the new subwindow, which is found in the upper right corner.

Arguments

None.

Value Returned

- x_subwindowID: Returns the window ID of the new subwindow.
- nil: Returns nil and an error message if there is no current Waveform window.

Example

addSubwindow() => 3

Adds a new subwindow to the Waveform window.
addSubwindowTitle

addSubwindowTitle( x_windowtitle ) => t/nil

Description

Adds a title to the current subwindow in the active window. The current subwindow is defined using the currentSubwindow command.

Arguments

x_windowtitle  User-defined title for the subwindow.

Value Returned

t  The user-supplied name of the current subwindow.

nil  Returns nil if the title is not created.

Example

addSubwindowTitle( "waveform 2") => t

Adds the title waveform 2 to the selected subwindow.
**addTitle**

`addTitle( x_windowtitle) => t/nil`

**Description**

Adds a title to the current active OCEAN window. The current window is defined using the `currentWindow` command.

**Arguments**

`x_windowtitle`  
User-defined title for the window.

**Value Returned**

`t`  
The user-supplied name of the current window.

`nil`  
Returns `nil` if the title is not created.

**Example**

`addTitle( waveform 1) => t`

Adds the title `waveform 1` to the selected window.
addWaveLabel

addWaveLabel( x_waveIndex l_location t_label [?textOffset l_textOffset] [?Color x_color] [?justify t_justify] [?fontStyle t_fontStyle] [?height x_height] [?orient t_orient] [?drafting g_drawing] [?overBar g_overbar]) => s_labelId/nil

Description

Attaches a label to the specified waveform curve in the current subwindow.

Arguments

x_waveIndex
  Integer identifying the waveform curve.

l_location
  List of two waveform coordinates that describe the location for the label.

t_label
  Label for the waveform.

l_textOffset
  An offset of the label from l_location, in screen units of the current subwindow. If l_textOffset is not specified, it defaults to 0:0 and the label is displayed at the location. If l_textOffset is specified, the label is offset from the location and a directional arrow is drawn from the label to the location. For example, if the offset is specified as 0:20, the label is drawn 20 units above the location and a directional label is drawn from the label to the location. This feature is useful to label points on a waveform and not obstruct the waveform.

x_color
  Label color specified as an index in the technology file.
  Default value: 10

t_justify
  Justification, which is specified as "upperLeft", "centerLeft", "lowerLeft", "upperCenter", "centerCenter", "lowerCenter", "upperRight", "centerRight", or "lowerRight".
  Default value: "lowerLeft"

t_fontStyle
  Font style, which is specified as "euroStyle", "gothic", "math", "roman", "script", "stick", "fixed", "fixed"
"swedish", "raster", or "milSpec".
Default value: the font style of the current subwindow

\texttt{x\_height}\quad \text{Height of the font.}
Default value: the font height of the current subwindow

\texttt{t\_orient}\quad \text{Orientation of the text, specified as either "R0" or "R90".}
Default value: "R0"

\texttt{g\_drafting}\quad \text{Boolean that specifies whether the label stays backwards or upside-down. If set to \texttt{t}, a backwards or upside-down label is displayed in a readable form. If set to \texttt{nil}, a backwards or upside-down label stays the way it is.}
Default value: \texttt{t}

\texttt{g\_overbar}\quad \text{Boolean that specifies whether underscores in labels are displayed as overbars. If set to \texttt{t}, underscores in labels are displayed as overbars. If set to \texttt{nil}, underbars are displayed as underbars.}
Default value: \texttt{nil}

**Value Returned**

\texttt{s\_labelId}\quad \text{Returns an identification number for the waveform label.}

\texttt{nil}\quad \text{Returns \texttt{nil} if there is an error.}

**Example**

\texttt{addWaveLabel( 1 list( 0 0.5 ) "R5 = " )}

Attaches the "R5 = " label to the specified coordinates on waveform curve 1.
addWindowLabel

addWindowLabel( l_location t_label ) => s_labelId/nil

Description
Displays a label in the current subwindow. The location for the label is specified with a list of two numbers between 0 and 1.

Arguments

l_location List of two waveform coordinates that describe the location for the label.
Valid values: 0 through 1

t_label Label for the waveform.

Value Returned

s_labelId Returns an identification number for the subwindow label.

nil Returns nil if there is an error.

Example

label = addWindowLabel( list( 0.75 0.75 ) "test" )

Adds the test label to the current subwindow at the specified coordinates and stores the label identification number in label.
clearAll

clearAll() => t/nil

Description
Erases the contents of the current Waveform window and deletes the waveforms, title, date stamp, and labels stored in internal memory.

Arguments
None.

Value Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Returns t if the waveform information is deleted.</td>
</tr>
<tr>
<td>nil</td>
<td>Returns nil and an error message if there is no current Waveform window.</td>
</tr>
</tbody>
</table>

Example
clearAll() => t

Erases the contents of the current Waveform window.
clearSubwindow

clearSubwindow() => t/nil

Description
Erases the contents of the current subwindow.

Arguments
None.

Value Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Returns t if the contents of the subwindow are erased.</td>
</tr>
<tr>
<td>nil</td>
<td>Returns nil and an error message otherwise.</td>
</tr>
</tbody>
</table>

Example

clearSubwindow() => t

Erases the contents of the current subwindow.
currentSubwindow

currentSubwindow( x_subwindow ) => t/nil

Description

Specifies x_subwindow as the current subwindow.

Arguments

x_subwindow Number of the subwindow, found in the upper right corner, that is to become the current subwindow.

Value Returned

t Returns t when the subwindow is set to x_subwindow.
nil Returns nil if there is an error.

Example

currentSubwindow( 2 )

Specifies subwindow 2 as the current subwindow.
currentWindow

currentWindow( w_windowId ) => w_windowId/nil

Description

Specifies w_windowId as the current Waveform window.

Arguments

w_windowId Waveform window ID.

Value Returned

w_windowId Returns the current Waveform window ID.
nil Returns nil and an error if the current window cannot be set.

Example

currentWindow( window(2) )

Specifies window 2 as the current Waveform window.
dbCompressionPlot

dbCompressionPlot(o_wave x_harmonic x_extrapolationPoint
  [?compression x_compression] ) => t/nil

Description

Plots the nth compression point plot. The x_compression argument is optional and defaults to 1, for 1dB compression, if omitted.

Arguments

  o_wave                        The waveform for which to plot the compression.
  x_harmonic                   Harmonic frequency index.
  x_extrapolationPoint         The extrapolation point.
  x_compression

  The amount of dB compression.
  Default value: 1

Value Returned

  t                           Returns t if the point is plotted
  nil                         returns nil if there was an error

Example

dbCompressionPlot(v("/Pif") 2 -25)
deleteSubwindow

deleteSubwindow() => t/nil

Description

Deletes the current subwindow from the current Waveform window.

Arguments

None.

Value Returned

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>Returns t if the current subwindow is deleted.</td>
</tr>
<tr>
<td>nil</td>
<td>Returns nil and an error message if there is no current subwindow.</td>
</tr>
</tbody>
</table>

Example

deleteSubwindow() => t

Deletes the current subwindow from the Waveform window.
deleteWaveform

deleWaveform( {x_index  |  all_string } ) => t/nil

Description

Deletes the specified waveform curve or all the waveform curves from the current subwindow of a Waveform window.

Arguments

x_index

Integer identifying a particular waveform curve.

all_string

The string "all" specifying that all waveform curves are to be deleted.

Value Returned

t

Returns t if the curves are deleted.

nil

Returns nil and an error message if the curves are not deleted.

Examples

deleWaveform( 1 ) => t

Deletes waveform 1 from the current subwindow.

deleWaveform( "all" ) => t

Deletes all the curves from the current subwindow.
displayMode

displayMode( t_mode ) => t/nil

Description

Sets the display mode of the current subwindow.

Arguments

$\text{t\_mode}$

String representing the display mode for the subwindow.

Valid values: strip, smith, or composite

Value Returned

$\text{t}$

Returns $t$ when the display mode of the subwindow is set.

$\text{nil}$

Returns $\text{nil}$ and an error message if the display mode cannot be set.

Example

displayMode( "composite" ) => t

Sets the current subwindow to display in composite mode.
**graphicsOff**

`graphicsOff() => t/nil`

**Description**

Disables the redrawing of the current Waveform window.

You might use this command to freeze the Waveform window display, send several plots to the window, and then unfreeze the window to display all the plots at once.

**Arguments**

None.

**Value Returned**

- **t**: Returns `t` if redrawing is disabled.
- **nil**: Returns `nil` if there is an error, such as there is no current Waveform window.

**Example**

`graphicsOff() => t`

Disables the redrawing of the Waveform window.
graphicsOn

graphicsOn() => t/nil

Description

Enables the redrawing of the current Waveform window.

Arguments

None.

Value Returned

<table>
<thead>
<tr>
<th>t</th>
<th>Returns t if redrawing is enabled.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>Returns nil if there is an error, such as there is no current Waveform window.</td>
</tr>
</tbody>
</table>

Example

graphicsOn() => t

Enables the redrawing of the current Waveform window.
**hardCopy**

`hardCopy() => t/nil`

**Description**

Sends a Waveform window plot to a printer.

**Note:** You must first set any plotting options with the [hardCopyOptions](#) command.

**Arguments**

None.

**Value Returned**

- `t` Returns `t` if successful.
- `nil` Returns `nil` if there is an error.

**Example**

`hardCopy() => t`

Sends a waveform plot to the printer.
hardCopyOptions

hardCopyOptions( [?hcNumCopy x_hcNumCopy] [?hcDisplay t_hcDisplay] [?hcOrientation s_hcOrientation] [?hcOutputFile g_hcOutputFile] [?hcPaperSize t_hcPaperSize] [?hcPlotterName t_hcPlotterName] [?hcTmpDir t_hcTmpDir] ) => g_value/nil

Description

Sets Waveform window hardcopy plotting options.

The option takes effect for any Waveform window or subwindow that is opened after the option is set.

Arguments

x_hcNumCopy
The number of copies to plot.
Valid values: any positive integer
Default value: 1

t_hcDisplay
The display name.
Valid values: defined in the technology file
Default value: "display"

s_hcOrientation
The plot orientation.
Valid values: ’portrait’, ’landscape’, ’automatic
Default value: ’automatic

g_hcOutputFile
Name of the output file.
Valid values: a string or nil
Default value: nil

t_hcPaperSize
The plot paper size.
Valid values: specified in .cdsplotinit
Default value: specified in .cdsplotinit

t_hcPlotterName
The name of the plotter.
**OCEAN Reference**

**Plotting Commands**

Valid values: specified in `.cdsplotinit`
Default value: specified in `.cdsplotinit`

$t_{hcTmpDir}$

The name of a temporary directory to be used for scratch space.
Valid values: name of a temporary directory
Default value: "/usr/tmp"

**Value Returned**

$g\_value$

Returns the new value of the option.

nil

Returns nil if there is an error.

**Examples**

```ruby
hardCopyOptions( ?hcNumCopy 1 )
```

Plots one copy of the window or subwindow.

```ruby
hardCopyOptions(?hcNumCopy 3 ?hcOutputFile "myOutFile")
```

Plots three copies of the window or subwindow and sends them to the file `myOutFile`. 
ip3Plot

ip3Plot( o_wave x_sigHarmonic x_refHarmonic x_extrapolationPoint ) => t/nil

Description
Plots the IP3 curves.

Arguments

o_wave Waveform for which to plot the ip3.

x_sigHarmonic Index of the third order harmonic.

x_refHarmonic Index of the first order (fundamental) harmonic.

x_extrapolationPoint Extrapolation point.

Value Returned

t Returns t if the curves are plotted.

nil Returns nil if there is an error.

Example

ip3Plot(v("/net28") 47 45 -25)
newWindow

newWindow() => w_windowID/nil

Description

Creates a new Waveform window and returns the window ID.

Arguments

None.

Value Returned

<table>
<thead>
<tr>
<th>w_windowId</th>
<th>Returns the window ID of the new Waveform window.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>Returns nil and an error message if the new Waveform window cannot be created.</td>
</tr>
</tbody>
</table>

Example

newWindow() => window:3

Creates a new Waveform window that is numbered 3 in the upper right corner.
plot

plot( o_waveform1 [o_waveform2 ...] [?yNumber l_yNumberList] 
    [?expr l_exprList] ) => t/nil

Description
Plots waveforms in the current subwindow. If there is no Waveform window, this command opens one.

Arguments

o_waveform1  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

o_waveform2  Additional waveform object.

l_yNumberList  List that specifies the Y axes where the waveforms are to be plotted. The number of Y axes must match the number of waveform objects specified.
               Valid values: 1, 2, 3, and 4

l_exprList  List of strings used to give names to the waveform objects.

Value Returned

  t  Returns $t$ if the waveforms are plotted.

  nil  Returns $\text{nil}$ and an error message if the waveforms cannot be plotted.

Examples

plot( v( "/net56" ) )

Plots the waveform for the voltage of $\text{net56}$.

plot( vm( "/net56" ) vp( "/net56" ) )

Plots the waveforms for the magnitude of the voltage of $\text{net56}$ and the phase of the voltage of $\text{net56}$.
plot( v( "OUT" ) i( "VFB" ) ?expr list( "voltage" "current" ) )

Plots the waveforms, but changes one legend label from v("OUT") to voltage and changes the other legend label from i("VFB") to current.
plotStyle

plotStyle( S_style ) => t/nil

Description

Sets the plotting style for all the waveforms in the current subwindow.

If the plotting style is **bar** and the display mode is **smith**, the plotting style is ignored until the display mode is set to **strip** or **composite**.

Arguments

**S_style**  
Plotting style for the subwindow.  
*Valid values:* **auto**, **scatterplot**, **bar**, **joined**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>auto</strong></td>
<td>The appropriate plotting style is automatically chosen.</td>
</tr>
<tr>
<td><strong>scatterplot</strong></td>
<td>Data points are not joined.</td>
</tr>
<tr>
<td><strong>bar</strong></td>
<td>Vertical bars are drawn at each data point that extend from the point to the bottom of the graph.</td>
</tr>
<tr>
<td><strong>joined</strong></td>
<td>Each data point is joined to adjacent data points by straight-line segments.</td>
</tr>
</tbody>
</table>

Value Returned

**t**  
Returns **t** if the plotting style is set.

**nil**  
Returns **nil** and an error message if the plotting style is not set.

Example

```plaintext
plotStyle( 'auto' ) => t
```

Sets the plot style to **auto**.
removeLabel

removeLabel( l_id ) => t/nil

Description

Removes the label, or all the labels identified in a list, from the current subwindow.

Arguments

l_id
List of labels to remove.

Value Returned

t Returns t when the label or labels are removed.
nil Returns nil if there is an error.

Examples

label = addWindowLabel( list( 0.75 0.75 ) "test" )

Adds the "test" label to the current subwindow at the specified coordinates and stores the label identification number in label.

removeLabel( label )

Removes the label whose identification number is stored in label. In this case, the "test" label is removed.
xLimit

\[ \text{xLimit( l\_minMax ) } \Rightarrow t/\text{nil} \]

Description

Sets the X axis display limits for the current subwindow. This command does not take effect if the display mode is set to smith.

Arguments

\[ l\_minMax \]

List of two numbers in waveform coordinates that describe the limits for the display. The first number is the minimum and the second is the maximum. If this argument is set to nil, the limit is set to auto.

Value Returned

\[ t \]

Returns t when the X axis display limits are set.

\[ \text{nil} \]

Returns nil and an error message if the X axis display limits are not set.

Example

\[ \text{xLimit( list( 1 100 ) ) } \Rightarrow t \]

Sets the X axis to display between 1 and 100.
yLimit

yLimit( l_minMax [?yNumber x_yNumber] [?stripNumber x_stripNumber]) => t/nil

Description

Sets the Y axis display limits for the waveforms associated with a particular Y axis and strip in the current subwindow.

If you do not specify x_stripNumber, the limits are applied when the subwindow is in composite mode.

Arguments

l_minMax List of two numbers in waveform coordinates that describe the limits for the display. The first number is the minimum and the second is the maximum. If this argument is set to nil, the limit is set to auto.

x_yNumber Specifies the Y axis to have the limited display. Valid values: 1 through 4

x_stripNumber Specifies the strip in which the y display is to be limited. Valid values: 1 through 20

Value Returned

t Returns t if the Y axis display limits are set.

nil Returns nil and an error message if the Y axis display limits cannot be set.

Example

yLimit( list( 4.5 7.5 ) ?yNumber 1 ) => t

Sets Y axis 1 to display from 4.5 to 7.5.
OCEAN Aliases

The aliases in this chapter provide you with shortcuts to commonly used pairs of commands. By default, these aliases operate on results previously selected with `selectResult`. However, you can also use an alias on a different set of results. For example, to specify a different set of results for the `vm` alias, use the following syntax.

```
vm( t_net [?result s_resultName] )
```

where `s_resultName` is the name of the datatype for the particular analysis you want.

You can use the `vm` alias on results stored in a different directory as follows:

```
vm( t_net [?resultsDir t_resultsDir] [?result s_resultName] )
```

where `t_resultsDir` is the name of a different directory containing PSF results, and `s_resultName` is the name of a datatype contained in that directory. (If you specify another directory with `t_resultsDir`, you must also specify the particular results with `s_resultName`.)

List of Aliases

<table>
<thead>
<tr>
<th>Alias</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vm</code></td>
<td><code>vm( t_net [?resultsDir t_resultsDir][?result s_resultname]) =&gt; o_waveform/nil</code></td>
<td>Aliased to <code>mag(v())</code>. Gets the magnitude of the voltage of a net.</td>
</tr>
<tr>
<td><code>vdb</code></td>
<td><code>vdb( t_net [?resultsDir t_resultsDir][?result s_resultname]) =&gt; o_waveform/nil</code></td>
<td>Aliased to <code>db20(v())</code>. Gets the power gain in decibels from net in to net out.</td>
</tr>
<tr>
<td><code>vp</code></td>
<td><code>vp( t_net [?resultsDir t_resultsDir][?result s_resultname]) =&gt; o_waveform/nil</code></td>
<td>Aliased to <code>phase(v())</code>. Gets the phase of the voltage of a net.</td>
</tr>
</tbody>
</table>
**List of Aliases, continued**

<table>
<thead>
<tr>
<th>Alias (short)</th>
<th>Alias (long)</th>
<th>Description</th>
</tr>
</thead>
</table>
| vr           | vi_r(t_net [resultsDir  
|              | t_resultsDir][result  
|              | s_resultname]) => o_waveform/nil | `real(v())`. Gets the real part of a complex number representing the voltage of a net. |
| vim          | vi_im(t_net [resultsDir  
|              | t_resultsDir][result  
|              | s_resultname]) => o_waveform/nil | `imag(v())`. Gets the imaginary part of a complex number representing the voltage of a net. |
| im           | i_m(t_component [resultsDir  
|              | t_resultsDir][result  
|              | s_resultName]) => o_waveform/nil | `mag(i())`. Gets the magnitude of the AC current through a component. |
| ip           | i_p(t_component [resultsDir  
|              | t_resultsDir][result  
|              | s_resultName]) => o_waveform/nil | `phase(i())`. Gets the phase of the AC current through a component. |
| ir           | i_r(t_component [resultsDir  
|              | t_resultsDir][result  
|              | s_resultName]) => o_waveform/nil | `real(i())`. Gets the real part of a complex number representing the AC current through a component. |
| iim          | i_im(t_component [resultsDir  
|              | t_resultsDir][result  
|              | s_resultName]) => o_waveform/nil | `imag(i())`. Gets the imaginary part of a complex number representing the AC current through a component. |
Predefined Functions and Waveform (Calculator) Functions

This chapter contains information about the following functions. Some additional predefined data access commands are described in the Affirma Analog Design Environment SKILL Language Reference manual.

- **Predefined Arithmetic Functions** on page 185
  - *abs* on page 187
  - *acos* on page 188
  - *add1* on page 189
  - *asin* on page 190
  - *atan* on page 191
  - *cos* on page 192
  - *exp* on page 193
  - *linRg* on page 194
  - *log* on page 195
  - *logRg* on page 196
  - *max* on page 197
  - *min* on page 198
  - *mod* on page 199
  - *random* on page 200
  - *round* on page 201
  - *sin* on page 202
sqr

t on page 203

srandom on page 204

sub1 on page 205

tan on page 206

Waveform (Calculator) Functions on page 206

average on page 207

b1f on page 209

bandwidth on page 210

clip on page 211

compression on page 213

compressionVRI on page 215

compressionVRICurves on page 217

conjugate on page 219

convolve on page 222

cPwrContour on page 224

cReflContour on page 226

cross on page 228

db10 on page 230

db20 on page 231

dbm on page 232

delay on page 233

deriv on page 236

dft on page 237

flip on page 239

fourEval on page 240

frequency on page 241

ga on page 242
gac on page 243
gainBwProd on page 245
gainMargin on page 247
gmax on page 248
gmin on page 249
gmsg on page 250
gmux on page 251
gp on page 252
gpc on page 253
groupDelay on page 255
gt on page 256
harmonic on page 257
harmonicFreqList on page 259
harmonicList on page 261
iinteg on page 263
imag on page 264
integ on page 265
ipn on page 266
ipnVRI on page 269
ipnVRICurves on page 272
kf on page 275
In on page 276
log10 on page 277
lsb on page 278
lshift on page 280
mag on page 281
nc on page 282
overshoot on page 284
peakToPeak on page 286
phase on page 287
phaseDeg on page 289
phaseDegUnwrapped on page 290
phaseMargin on page 291
phaseRad on page 293
phaseRadUnwrapped on page 294
pow on page 295
psd on page 297
psdbb on page 301
real on page 305
riseTime on page 306
rms on page 308
rmsNoise on page 309
root on page 310
rshift on page 312
sample on page 313
settlingTime on page 315
slewRate on page 317
spectralPower on page 319
ssb on page 320
tangent on page 322
thd on page 323
value on page 325
xmax on page 327
xmin on page 329
Predefined Arithmetic Functions

Several functions are predefined in the Cadence® SKILL language. The full syntax and brief definitions for these functions follows the table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add1</strong> (n)</td>
<td>n + 1</td>
</tr>
<tr>
<td><strong>Abs</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Sub1</strong> (n)</td>
<td>n – 1</td>
</tr>
<tr>
<td><strong>Exp</strong> (n)</td>
<td>e raised to the power n</td>
</tr>
<tr>
<td><strong>LinRg</strong> (n_&lt;from&gt;, n_&lt;to&gt;, n_&lt;by&gt;)</td>
<td>Returns list of numbers in linear range from n_&lt;from&gt; to n_&lt;to&gt; in n_&lt;by&gt; steps</td>
</tr>
<tr>
<td><strong>Log</strong> (n)</td>
<td>Natural logarithm of n</td>
</tr>
<tr>
<td><strong>LogRg</strong> (n_&lt;from&gt;, n_&lt;to&gt;, n_&lt;by&gt;)</td>
<td>Returns list of numbers in log10 range from n_&lt;from&gt; to n_&lt;to&gt; in n_&lt;by&gt; steps</td>
</tr>
<tr>
<td><strong>Max</strong> (n1 n2 ...)</td>
<td>Maximum of the given arguments</td>
</tr>
<tr>
<td><strong>Min</strong> (n1 n2 ...)</td>
<td>Minimum of the given arguments</td>
</tr>
<tr>
<td><strong>Mod</strong> (x1 x2)</td>
<td>x1 modulo x2, that is, the integer remainder of dividing x1 by x2</td>
</tr>
<tr>
<td><strong>Round</strong> (n)</td>
<td>Integer whose value is closest to n</td>
</tr>
<tr>
<td><strong>Sqrt</strong> (n)</td>
<td>Square root of n</td>
</tr>
<tr>
<td><strong>Sin</strong> (n)</td>
<td>Sine, argument n is in radians</td>
</tr>
<tr>
<td><strong>Cos</strong> (n)</td>
<td>Cosine</td>
</tr>
<tr>
<td><strong>Tan</strong> (n)</td>
<td>Tangent</td>
</tr>
<tr>
<td><strong>Asin</strong> (n)</td>
<td>Arc sine, result is in radians</td>
</tr>
</tbody>
</table>
Predefined Arithmetic Functions

<table>
<thead>
<tr>
<th>Synopsis</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos((n))</td>
<td>arc cosine</td>
</tr>
<tr>
<td>atan((n))</td>
<td>arc tangent</td>
</tr>
</tbody>
</table>

Random Number Generator

| random(\(x\)) | Returns a random integer between 0 and \(x-1\). If random is called with no arguments, it returns an integer that has all of its bits randomly set. |
| srandom(\(x\)) | Sets the initial state of the random number generator to \(x\). |
abs

abs( n_number ) => n_result

Description

Returns the absolute value of a floating-point number or integer.

Arguments

n_number Floating-point number or integer.

Value Returned

n_result The absolute value of n_number.

Example

abs( -209.625) => 209.625
abs( -23) => 23
acos

acos( n_number ) => f_result

Description

Returns the arc cosine of a floating-point number or integer.

Arguments

n_number Floating-point number or integer.

Value Returned

f_result Returns the arc cosine of n_number.

Example

acos(0.3) => 1.266104
add1

add1( n_number ) => n_result

Description

Adds 1 to a floating-point number or integer.

Arguments

n_number Floating-point number or integer to increase by 1.

Value Returned

n_result n_number plus 1.

Example

add1( 59 ) => 60

Adds 1 to 59.
### asin

asin( n_number ) => f_result

#### Description

Returns the arc sine of a floating-point number or integer.

#### Arguments

- **n_number**: Floating-point number or integer.

#### Value Returned

- **f_result**: The arc sine of `n_number`.

#### Example

asin(0.3) => 0.3046927
atan

atan( n_number ) => f_result

Description

Returns the arc tangent of a floating-point number or integer.

Arguments

n_number Floating-point number or integer.

Value Returned

f_result The arc tangent of n_number.

Example

atan(0.3) => 0.2914568
COS

\[ \text{cos} \left( n_{\text{number}} \right) \Rightarrow f_{\text{result}} \]

**Description**

Returns the cosine of a floating-point number or integer.

**Arguments**

- \( n_{\text{number}} \) Floating-point number or integer.

**Value Returned**

- \( f_{\text{result}} \) The cosine of \( n_{\text{number}} \).

**Examples**

- \( \text{cos}(0.3) \Rightarrow 0.9553365 \)
- \( \text{cos}(3.14/2) \Rightarrow 0.0007963 \)
exp

exp( n_number ) => f_result

Description

Raises e to a given power.

Arguments

n_number Power to raise e to.

Value Returned

f_result The value of e raised to the power n_number.

Examples

exp( 1 ) => 2.718282
exp( 3.0 ) => 20.08554
linRg

linRg( n_from n_to n_by ) => l_range/nil

Description

Returns a list of numbers in the linear range from n_from to n_to incremented by n_by.

Arguments

n_from Smaller number in the linear range.
n_to Larger number in the linear range.
n_by Increment value when stepping through the range.

Value Returned

l_range List of numbers in the linear range.
nil Returned if error.

Example

range = linRg(-30 30 5)
(-30 -25 -20 -15 -10 -5 0 5 10 15 20 25 30)
log

\[ \text{log}( \ n_{\text{number}} \ ) \Rightarrow f_{\text{result}} \]

**Description**

Returns the natural logarithm of a floating-point number or integer.

**Arguments**

\[ n_{\text{number}} \]

Floating-point number or integer.

**Value Returned**

\[ f_{\text{result}} \]

The natural logarithm of \( n_{\text{number}} \).

**Example**

\[ \text{log}( 3.0 ) \Rightarrow 1.098612 \]
logRg

logRg( n_from n_to n_by ) => l_range/nil

Description

Returns a list of numbers in the log10 range from n_from to n_to advanced by n_by.

The list is a geometric progression where the multiplier is 10 raised to the 1/n_by power. For example if n_by is 0.5, the multiplier is 10 raised to the 2nd power or 100.

Arguments

n_from Smaller number in the linear range.

n_to Larger number in the linear range.

n_by Increment value when stepping through the range.

Value Returned

l_range List of numbers in the linear range.

nil Returned if error.

Example

logRg(1 1M 0.5)

(1.0 100.0 10000.0 1000000.0)
**max**

```max( n_num1 n_num2 [n_num3 ...] ) => n_result```

**Description**

Returns the maximum of the values passed in. Requires a minimum of two arguments.

**Arguments**

- `n_num1`  
  First value to check.
- `n_num2`  
  Next value to check.
- `[n_num3...]`  
  Additional values to check.

**Value Returned**

- `n_result`  
  The maximum of the values passed in.

**Examples**

- `max(3 2 1) => 3`
- `max(-3 -2 -1) => -1`
**min**

\[
\text{min( } n_{\text{num1}} \ n_{\text{num2}} \ \[ n_{\text{num3}} \ \text{...} \ \} \ ) \Rightarrow n_{\text{result}}
\]

**Description**

Returns the minimum of the values passed in. Requires a minimum of two arguments.

**Arguments**

- \( n_{\text{num1}} \) 
  First value to check.
- \( n_{\text{num2}} \) 
  Next value to check.
- \([ n_{\text{num3}} \ ...]\) 
  Additional values to check.

**Value Returned**

- \( n_{\text{result}} \) 
  The minimum of the values passed in.

**Examples**

\[
\text{min(1 2 3)} \Rightarrow 1
\]
\[
\text{min(}-1\ -2.0\ -3\) \Rightarrow -3.0
\]
mod

mod( x_integer1 x_integer2) => x_result

Description

Returns the integer remainder of dividing two integers. The remainder is either zero or has the sign of the dividend.

Arguments

x_integer1
Dividend.

x_integer2
Divisor.

Value Returned

x_result
The integer remainder of the division. The sign is determined by the dividend.

Example

mod(4 3) => 1
**random**

`random( [x_number] ) => x_result`

**Description**

Returns a random integer between 0 and `x_number` minus 1.

If you call `random` with no arguments, it returns an integer that has all of its bits randomly set.

**Arguments**

`x_number`  
An integer.

**Value Returned**

`x_result`  
Returns a random integer between 0 and `x_number` minus 1.

**Example**

`random( 93 ) => 26`
round

round( n_arg ) => x_result

Description
Rounds a floating-point number to its closest integer value.

Arguments

n_arg Floating-point number.

Value Returned

x_result The integer whose value is closest to n_arg.

Examples

round(1.5) => 2
round(-1.49) => -1
round(1.49) => 1
sin

\( \text{sin}( \text{n\_number} ) \Rightarrow f\_result \)

Description

Returns the sine of a floating-point number or integer.

Arguments

\( n\_number \quad \text{Floating-point number or integer.} \)

Value Returned

\( f\_result \quad \text{The sine of n\_number.} \)

Examples

\( \text{sin}(3.14/2) \Rightarrow 0.9999997 \)
\( \text{sin}(3.14159/2) \Rightarrow 1.0 \)

Floating-point results from evaluating the same expressions might be machine-dependent.
sqrt

sqrt( n_number ) => f_result

Description

Returns the square root of a floating-point number or integer.

Arguments

n_number  
Floating-point number or integer.

Value Returned

f_result  
The square root of n_number.

Examples

sqrt( 49 ) => 7.0
sqrt( 43942 ) => 209.6235
srandom

srandom( x_number ) => t

Description
Sets the seed of the random number generator to a given number.

Arguments

x_number  An integer.

Value Returned

This function always returns t.

Example

srandom( 89 ) => t
**sub1**

\[ \text{sub1( } n\_\text{number } \text{) } \Rightarrow n\_\text{result} \]

**Description**

Subtracts 1 from a floating-point number or integer.

**Arguments**

\[ n\_\text{number} \quad \text{Floating-point number or integer.} \]

**Value Returned**

\[ n\_\text{result} \quad \text{Returns } n\_\text{number} \text{ minus 1.} \]

**Example**

\[ \text{sub1( } 59 \text{ ) } \Rightarrow 58 \]

Subtracts 1 from 59.
**tan**

\[ \tan( \ n\_number \ ) \Rightarrow \ f\_result \]

**Description**

Returns the tangent of a floating-point number or integer.

**Arguments**

- \( n\_number \): Floating-point number or integer.

**Value Returned**

- \( f\_result \): The tangent of \( n\_number \).

**Example**

\[ \tan( \ 3.0 \ ) \Rightarrow -0.1425465 \]

**Waveform (Calculator) Functions**

The calculator commands are described in this section.
average

average( o_waveform ) => n_average/o_waveformAverage/nil

Description

Computes the average of a waveform over its entire range.

Average is defined as the integral of the expression $f(x)$ over the range of $x$, divided by the range of $x$.

For example, if $y=f(x)$, average(y) =

\[
\frac{\int_{from}^{to} f(x) \, dx}{to - from}
\]

where from is the initial value for x and to is the final value.

Arguments

- **o_waveform**
  
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

Value Returned

- **n_average**
  
  Returns a number representing the average value of the input waveform.

- **o_waveformAverage**
  
  Returns a waveform (or family of waveforms) representing the average value if the input is a family of waveforms.

- **nil**
  
  Returns nil and an error message otherwise.
Example

\[ \text{average( } v( \text{ "/net9" } ) ) \]

Gets the average voltage (Y-axis value) of /net9 over the entire time range specified in the simulation analysis.
b1f

b1f( o_s11 o_s12 o_s21 o_s22 ) => o_waveform/nil

Description

Returns the alternative stability factor in terms of the supplied parameters.

Arguments

o_s11 Waveform object representing s11.
o_s12 Waveform object representing s12.
o_s21 Waveform object representing s21.
o_s22 Waveform object representing s22.

Value Returned

o_waveform Waveform object representing the alternative stability factor.
nil Returns nil and an error message otherwise.

Examples

s11 = sp(1 1)
s12 = sp(1 2)
s21 = sp(2 1)
s22 = sp(2 2)
plot(b1f(s11 s12 s21 s22))


**bandwidth**

\[ \text{bandwidth}( \ o\_\text{waveform} \ n\_\text{db} \ t\_\text{type} ) \Rightarrow n\_\text{value}/o\_\text{waveform}/\text{nil} \]

**Description**

Calculates the bandwidth of a waveform.

**Arguments**

- **o\_waveform**
  
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{d\_wave:XXX\_XX}.)

- **n\_db**
  
  Positive number that defines the bandwidth.

- **t\_type**
  
  Type of input filter.
  
  Valid values: "low", "high" or "band".

**Value Returned**

- **n\_value**
  
  Returns a number representing the value of the bandwidth if the input argument is a single waveform.

- **o\_waveform**
  
  Returns a waveform (or family of waveforms) representing the bandwidth if the input argument is a family of waveforms.

- **nil**
  
  Returns \text{nil} and an error message otherwise.

**Examples**

- \text{bandwidth( v( "/OUT" ) 3 "low")}

  Gets the 3 dB bandwidth of a low-pass filter.

- \text{bandwidth( v( "/OUT" ) 4 "band" )}

  Gets the 4 dB bandwidth of a band-pass filter.


clip

clip( o_waveform n_from n_to ) => o_waveform/nil

Description

Restricts the waveform to the range defined by n_from and n_to.

You can use the clip function to restrict the range of action of other commands. If n_from is nil, n_from is taken to be the first X value of the waveform, and if n_to is nil, n_to is taken to be the last X value of the waveform. If both n_to and n_from are nil, the original waveform is returned.

Arguments

o_waveform            Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)
n_from                Starting value for the range on the X axis.
n_to                  Ending value for the range on the X axis.

Value Returned

o_waveform            Returns a waveform object if the input argument is a waveform object or returns a family of waveforms if the input argument is a family of waveforms.
nil                    Returns nil and an error message otherwise.

Examples

x = clip( v( "/net9" ) 2m 4m )
plot( x )

Plots the portion of a waveform that ranges from 2 ms to 4 ms.

plot( clip( v( "/net9" ) nil nil ) )

Plots the original waveform.
plot( clip( v( "/net9" ) nil 3m ) )

Plots the portion of a waveform that ranges from 0 to 3 ms.
**compression**

```
compression( o_waveform  [ ?x f_x ]  [ ?y f_y ]  [ ?compression f_compression ]  [ ?io s_measure ] ) => f_compPoint/nil
```

**Description**

Performs an $n$th compression point measurement on a power waveform.

The `compression` function uses the power waveform to extrapolate a line of constant slope (dB/dB) according to a specified input or output power level. This line represents constant small-signal power gain (ideal gain). The function finds the point where the power waveform drops $n$ dB from the constant slope line and returns either the X coordinate (input referred) value or the Y coordinate (output referred) value.

**Arguments**

- **o_waveform**
  Waveform object representing output power (in dBm) versus input power (in dBm).

- **f_x**
  The X coordinate value (in dBm) used to indicate the point on the output power waveform where the constant-slope power line begins. This point should be in the linear region of operation. Default value: Unless $f_y$ is specified, defaults to the X coordinate of the first point of the $o\_waveform$ wave.

- **f_y**
  The Y coordinate value (in dBm) used to indicate the point on the output power waveform where the constant-slope power line begins. This point should be in the linear region of operation. Default value: Unless $f_x$ is specified, defaults to the Y coordinate of the first point of the $o\_waveform$ wave.

- **f_compression**
  The delta (in dB) between the power waveform and the ideal gain line that marks the compression point. Default value: 1

- **s_measure**
  Symbol indicating whether the measurement is to be input referred (`'input`) or output referred (`'output`). Default value: `'input`
Value Returned

\( f_{\text{compPoint}} \)    Depending on the setting of \( s_{\text{measure}} \), returns either input referred or output referred compression point.

\( \text{nil} \)    Returns \( \text{nil} \) and an error message otherwise.

Examples

\[
xloc = \text{compression}\left( \text{wave} \ ?x \ -25 \ ?\text{compress} \ 1 \right)
yloc = \text{compression}\left( \text{wave} \ ?x \ -25 \ ?\text{measure} \ "\text{Output}" \right)
\]

; Each of following returns a compression measurement:

\[
\text{compression}\left( \text{dB20\left( \text{harmonic}\left( \text{v}\left( \"/\text{Pif}\" \ ?\text{result} \ "\text{pss_fd}\" \right) \right) \right) \right)} \right)
\]

\[
\text{compression}\left( \text{dbm}\left( \text{harmonic}\left( \text{spectralPower}\left( \text{v}\left( \"/\text{Pif}\" \ ?\text{result} \ "\text{pss_fd}\" \ / \ 50.0 \ \text{v}\left( \"/\text{Pif}\" \ ?\text{result} \ "\text{pss_fd}\" \right) \right) \right) \right) \right) \right)
\]

\[
\text{compression}\left( \text{dbm}\left( \text{harmonic}\left( \text{spectralPower}\left( \text{v}\left( \"/\text{Pif}\" \ ?\text{result} \ "\text{pss_fd}\" \ / \ \text{resultParam}\left( \"\text{rif:r}\" \ ?\text{result} \ "\text{pss_td}\" \ \text{v}\left( \"/\text{Pif}\" \ ?\text{result} \ "\text{pss_fd}\" \right) \right) \right) \right) \right) \right) \right)
\]

\[
\text{compression}\left( \text{dbm}\left( \text{harmonic}\left( \text{spectralPower}\left( \text{i}\left( \"/\text{rif/PLUS}\" \ ?\text{result} \ "\text{pss_fd}\" \ \text{v}\left( \"/\text{Pif}\" \ ?\text{result} \ "\text{pss_fd}\" \right) \right) \right) \right) \right) \right)
\]

\[
x -25 \ ?\text{compress} \ 0.1 \ ?\text{measure} \ "\text{Output}"
\]
compressionVRI

compressionVRI( o_vport x_harm [?iport o_iport] [?rport f_rport] [?epoint f_epoint] [?gcomp f_gcomp] [?measure s_measure] ) => o_waveform/n_number/nil

Description

Performs an nth compression point measurement on a power waveform.

Use this function to simplify the declaration of a compression measurement. This function extracts the specified harmonic from the input waveform(s), and uses dBm(spectralPower((i or v/r),v)) to calculate a power waveform. The function passes this power curve and the remaining arguments to the compression function to complete the measurement.

The compression function uses the power waveform to extrapolate a line of constant slope (dB/dB) according to a specified input or output power level. This line represents constant small-signal power gain (ideal gain). The function finds the point where the power waveform drops n dB from the constant slope line and returns either the X coordinate (input referred) value or the Y coordinate (output referred) value.

Arguments

- **o_vport** Voltage across the output port. This argument must be a family of spectrum waveforms (1 point per harmonic) created by parametrically sweeping an input power (in dBm) of the circuit.

- **x_harm** Harmonic index of the voltage wave contained in o_vport. When o_iport is specified, also applies to a current waveform contained in o_iport.

- **o_iport** Current into the output port. This argument must be a family of spectrum waveforms (1 point per harmonic) created by parametrically sweeping an input power (in dBm) of the circuit. When specified, the output power is calculated using voltage and current.
  Default value: nil

- **f_rport** Resistance into the output port. When specified and o_iport is nil, the output power is calculated using voltage and resistance.
  Default value: 50
OCEAN Reference
Predefined Functions and Waveform (Calculator) Functions

\[ f_{epoint} \]
The X coordinate value (in dBm) used to indicate the point on the output power waveform where the constant-slope power line begins. This point should be in the linear region of operation. Default value: the X coordinate of the first point of the \( o_{\text{waveform}} \) wave

\[ f_{gcomp} \]
The delta (in dB) between the power waveform and the ideal gain line that marks the compression point. Default value: 1

\[ s_{measure} \]
Symbol indicating if measurement is to be input referred (’input) or output referred (’output). Default value: ’input

Value Returned

\[ o_{\text{waveform}} \]
Returns a waveform when \( o_{\text{waveform1}} \) is a family of waveforms.

\[ f_{number} \]
Returns a number when \( o_{\text{waveform1}} \) is a waveform.

\[ \text{nil} \]
Returns nil and an error message otherwise.

Examples

Each of the following returns a compression measurement:

\[
\text{compressionVRI}(v("/Pif" \ ?result "pss_fd")) 2
\]

\[
\text{compressionVRI}(v("/Pif" \ ?result "pss_fd")) 2
\quad\text{rport resultParam("rif:r" ?result "pss_td")}
\]

\[
\text{compressionVRI}(v("/Pif" \ ?result "pss_fd")) 2
\quad\text{iport i("/rif/PLUS" \ ?result "pss_fd") \ ?epoint -25}
\quad\text{?gcomp 0.1 \ ?measure "Output"}
\]
compressionVRICurves

compressionVRICurves( o_vport x_harm [?iport o_iport] [?rport f_rport] [?epoint f_epoint] [?gcomp f_gcomp] ) =>
o_waveform/nil

Description

Constructs the waveforms associated with an \( n \)th compression measurement.

Use this function to simplify the creation of waveforms associated with a compression measurement. This function extracts the specified harmonic from the input waveform(s), and uses \( \text{dBm}(\text{spectralPower}((i \text{ or } v/r),v)) \) to calculate a power waveform.

The \text{compressionVRICurves} function uses the power waveform to extrapolate a line of constant slope (1dB/1dB) according to a specified input or output power level. This line represents constant small-signal power gain (ideal gain). The function shifts the line down by \( n \) dB and returns it, along with the power waveform, as a family of waveforms.

This function only creates waveforms and neither performs a compression measurement nor includes labels with the waveforms. Use the \text{compression} or \text{compressionVRI} function for making measurements.

Arguments

\text{o_vport}

Voltage across the output port. This argument must be a family of spectrum waveforms (1 point per harmonic) created by parametrically sweeping an input power (in dBm) of the circuit.

\text{x_harm}

Harmonic index of the wave contained in \text{o_vport}. When \text{o_iport} is specified, also applies to a current waveform contained in \text{o_iport}.

\text{o_iport}

Current into the output port. This argument must be a family of spectrum waveforms (1 point per harmonic) created by parametrically sweeping an input power (in dBm) of the circuit. When specified, the output power is calculated using voltage and current. Default value: nil

\text{f_rport}

Resistance into the output port. When specified and \text{o_iport} is nil, the output power is calculated using voltage and resistance. Default value: 50
**f_epoint**
The X coordinate value (in dBm) used to indicate the point on the output power waveform where the constant-slope power line begins. This point should be in the linear region of operation. Default value: the X coordinate of the first point of the o_waveform wave

**f_gcomp**
The delta (in dB) between the power waveform and the ideal gain line that marks the compression point. Default value: 1

**Value Returned**

**o_waveform**
Returns a family of waveforms containing the output power and tangent line.

**nil**
Returns nil and an error message otherwise.

**Examples**

Each of following examples returns curves related to a compression measurement:

```plaintext
compressionVRICurves(v("/Pif" ?result "pss_fd") 2)
compressionVRICurves(v("/Pif" ?result "pss_fd") 2
  ?rport resultParam("rif:r" ?result "pss_td"))
compressionVRICurves(v("/Pif" ?result "pss_fd") 2
  ?iport i("/rif/PLUS" ?result "pss_fd") epoint -25
  ?gcomp 0.1)
```
**conjugate**

conjugate( {o_waveform | n_x} ) => o_waveform/n_y/nil

**Description**

Returns the conjugate of a waveform or number.

**Arguments**

- **o_waveform**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

- **n_x**
  Complex or imaginary number.

**Value Returned**

- **o_waveform**
  Returns the conjugate of a waveform if the input argument is a waveform.

- **n_y**
  Returns the result of `n_x` being mirrored against the real axis (X axis) if the input argument is a number.

- **nil**
  Returns `nil` and an error message otherwise.

**Example**

For this example, assume that the first three statements are true for the `conjugate` function that follows them.

```plaintext
x=complex(-1 -2)
real(x) = -1.0
imag(x) = -2.0
conjugate(x) = complex(-1, 2)
```

Returns the conjugate of the input complex number.
complex

complex( f_real f_imaginary ) => o_complex

Description

Creates a complex number of which the real part is equal to the real argument, and the imaginary part is equal to the imaginary argument.

Arguments

f_real          The real part of the complex number.

f_imaginary     The imaginary part of the complex number.

Value Returned

o_complex      Returns the complex number.

Example

complex( 1.0 2.0 ) => complex( 1, 2 )
complexp

complexp( g_value ) => t/nil

Description

Checks if an object is a complex number. The suffix p is added to the name of a function to indicate that it is a predicate function.

Arguments

$g\_value$ A skill object.

Values Returned

$\text{t}$ Returns t when $g\_value$ is a complex number.

$\text{nil}$ Returns nil if there is an error.

Example

complexp( (complex 0 1) ) => t
complexp( 1.0 ) => nil
convolve

\[
\text{convolve}( \ o\_\text{waveform1} \ o\_\text{waveform2} \ n\_\text{from} \ n\_\text{to} \ t\_\text{type} \ n\_\text{by} \ ) \Rightarrow \\
\ o\_\text{waveform}/n\_\text{number}/\text{nil}
\]

**Description**

Computes the convolution of two waveforms.

Convolution is defined as

\[
\int_{\text{from}}^{\text{to}} f_1(s)f_2(t-s)\,ds
\]

\(f_1\) and \(f_2\) are the functions defined by the first and second waveforms.

**Note:** The `convolve` function is numerically intensive and might take longer than the other functions to compute.

**Arguments**

- `o_waveform1` Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)
- `o_waveform2` Additional waveform object.
- `n_from` Starting point (X-axis value) of the integration range.
- `n_to` Ending point (X-axis value) of the integration range.
- `t_type` Type of interpolation. Valid values: "linear" or "log".
- `n_by` Increment.
OCEAN Reference
Predefined Functions and Waveform (Calculator) Functions

Value Returned

\textbf{o\_waveform} \quad \text{Returns a waveform object representing the convolution if one of the input arguments is a waveform. Returns a family of waveforms if either of the input arguments is a family of waveforms.}

\textbf{n\_number} \quad \text{Returns a value representing the convolution if both of the input arguments are numbers.}

\textbf{nil} \quad \text{Returns nil and an error message otherwise.}

Example

\begin{verbatim}
sinWave = expr( n sin( n ) linRg( 0 20 0.01 ) )
triWave = artListToWaveform( '\{ (-4, 0) (-3, 1) (-2, 0) (-1, -1) ( 0, 0) ( 1, 1) ( 2, 0) ( 3, -1) ( 4, 0) \} )
plot( convolve( sinWave triWave 0 10 "linear" 1 ) )
\end{verbatim}

Gets the waveform from the convolution of the sine waveform and triangle waveform within the range of 0 to 10.
**cPwrContour**

cPwrContour o_iwave o_vwave x_harm [?iwaveLoad o_iwaveLoad] [?vwaveLoad o_vwaveLoad] [?maxPower f_maxPower] [?minPower f_minPower] [?refImp f_refImp] [?numCont x_numCont] [?closeCont b_closeCont] [?modifier s_modifier] ) => o_waveform/nil

**Description**

Constructs constant power contours for Z-Smith plotting. The trace of each contour correlates to reference reflection coefficients that all result in the same power level.

The $x_harm$ harmonic is extracted from all the input waveforms. Power is calculated using the `spectralPower` function. The reference reflection coefficients are calculated using voltage, current, and a reference resistance.

**Arguments**

- **o_iwave**
  Current used to calculate power, expected to be a two-dimensional family of harmonic waveforms.

- **o_vwave**
  Voltage used to calculate power, expected to be a two-dimensional family of harmonic waveforms.

- **x_harm**
  Harmonic index of the waves contained in o_iwave and o_vwave.

- **o_iwaveLoad**
  Current used to calculate reflection coefficient, expected to be a two-dimensional family of harmonic waveforms.
  Default value: o_iwave

- **o_vwaveLoad**
  Voltage used to calculate reflection coefficient, expected to be a two-dimensional family of harmonic waveforms.
  Default value: o_vwave

- **f_maxPower**
  Maximum power magnitude value for contours.
  Default value: automatic

- **f_minPower**
  Minimum power magnitude value for contours.
  Default value: automatic

- **x_numCont**
  Total number of contours returned.
  Default value: 8
### OCEAN Reference
Predefined Functions and Waveform (Calculator) Functions

<table>
<thead>
<tr>
<th><strong>f_refImp</strong></th>
<th>Reference resistance used to calculate reflection coefficients. Default value: 50</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>b_closeCont</strong></td>
<td>Boolean indicating when to close the contours. When nil, largest segment of each contour is left open. Default value: nil</td>
</tr>
<tr>
<td><strong>s_modifier</strong></td>
<td>Symbol indicating the modifier function to apply to the calculated power. The modifier function can be any single argument OCEAN function such as ’db10 or ’dBm. Default value: ’mag</td>
</tr>
</tbody>
</table>

#### Value Returned

| **o_waveform** | Returns a family of waveforms (contours) for Z-Smith plotting. |
| **nil** | Returns nil and an error message otherwise. |

#### Examples

The following example plots constant output power contours according to output:

```plaintext
cPwrContour(i("/I8/out" ?result "pss_fd") v("/net28" ?result "pss_fd")1)
```

The following example plots constant output power contours according to output reflection coefficients:

```plaintext
cPwrContour(i("/I8/out" ?result "pss_fd") v("/net28" ?result "pss_fd")1 ?maxPower 0.002 ?minPower 0.001 ?numCont 9)
```

The following example plots constant input power contours according to output reflection coefficients:

```plaintext
cPwrContour(i("/C25/PLUS" ?result "pss_fd") v("/net30" ?result "pss_fd")1 ?iwaveLoad i("/I8/out" ?result "pss_fd") ?vwaveLoad v("/net28" ?result "pss_fd") ?refImp 50.0 ?numCont 9 ?modifier "mag")
```
cReflContour

cReflContour( o_iwave o_vwave x_harm [?iwaveLoad o_iwaveLoad] 
   [?vwaveLoad o_vwaveLoad] [?maxRefl f_maxRefl] [?minRefl 
   f_minRefl] [?numCont x_numCont] [?refImp f_refImp] [?closeCont 
   b_closeCont] ) => o_waveform/nil

Description

Constructs constant reflection coefficient magnitude contours for Z-Smith plotting. The trace 
of each contour correlates to reference reflection coefficients that all result in the same 
reflection coefficient magnitude.

The $x_{\text{harm}}$ harmonic is extracted from all the input waveforms. Reflection coefficient 
magnitude is calculated using voltage, current, reference resistance, and the $\text{mag}$ function. 
The reference reflection coefficients are calculated separately by using voltage, current, and 
a reference resistance.

Arguments

- **o_iwave**
  Current used to calculate reflection coefficient magnitude, 
  expected to be a two-dimensional family of spectrum waveforms.

- **o_vwave**
  Voltage used to calculate reflection coefficient magnitude, 
  expected to be a two-dimensional family of spectrum waveforms.

- **x_harm**
  Harmonic index of the waves contained in $o_{\text{iwave}}$ and 
  $o_{\text{vwave}}$.

- **o_iwaveLoad**
  Current used to calculate reference reflection coefficient, 
  expected to be a two-dimensional family of harmonic waveforms. 
  Default value: $o_{\text{iwave}}$

- **o_vwaveLoad**
  Voltage used to calculate reference reflection coefficient, 
  expected to be a two-dimensional family of spectrum waveforms. 
  Default value: $o_{\text{vwave}}$

- **f_maxRefl**
  Maximum reflection coefficient magnitude value for contours. 
  Default value: automatic

- **f_minRefl**
  Minimum reflection coefficient magnitude value for contours. 
  Default value: automatic
Predefined Functions and Waveform (Calculator) Functions

Value Returned

\( x_{\text{numCont}} \)  
Total number of contours returned.  
Default value: 8

\( f_{\text{refImp}} \)  
Reference resistance used to calculate reflection coefficients.  
Default value: 50

\( b_{\text{closeCont}} \)  
Boolean indicating when to close the contours. When \( \text{nil} \), the largest segment of each contour is left open.  
Default value: \( \text{nil} \)

Examples

The following example plots constant output reflection coefficient contours according to output reflection coefficients:

```ocean
CReflContour(i("/I8/out" ?result "pss_fd") v("/net28"  
  ?result "pss_fd") 1)
```

The following example plots constant output reflection coefficient contours according to output reflection coefficients:

```ocean
CReflContour(i("/I8/out" ?result "pss_fd") v("/net28"  
  ?result "pss_fd") 1 ?maxRefl 0.7 ?minRefl 0.1 ?numCont 7)
```

The following example plots constant output reflection coefficient contours according to output reflection coefficients:

```ocean
CReflContour(i("/C25/PLUS" ?result "pss_fd")  
  v("/net30" ?result "pss_fd") 1  
  ?iwaveLoad i("/I8/out" ?result "pss_fd")  
  ?vwaveLoad v("/net28" ?result "pss_fd") ?refImp 50.0  
  ?numCont 9)
```
cross

cross( o_waveform n_crossVal x_n s_crossType ) =>
   o_waveform/g_value/nil

Description
Computes the X-axis value at which a particular crossing of the specified edge type of the
threshold value occurs.

Arguments

  o_waveform          Waveform object representing simulation results that can be
displayed as a series of points on a grid. (A waveform object
                      identifier looks like this: drwave:XXXXX.)

  n_crossVal          Y-axis value at which the corresponding values of X are
                      calculated.

  x_n                 Number that specifies which X value to return. If x_n equals 1,
                      the first X value with a crossing is returned. If x_n equals 2, the
                      second X value with a crossing is returned, and so on. If you
                      specify a negative integer for x_n, the X values with crossings
                      are counted from right to left (from maximum to minimum).

  s_crossType         Type of the crossing.
                      Valid values: ‘rising,’ ‘falling,’ ‘either.’

Value Returned

  o_waveform          Returns a waveform if the input argument is a family of
                      waveforms.

  g_value             Returns the X-axis value of the crossing point if the input
                      argument is a single waveform.

  nil                 Returns nil and an error message otherwise.

Examples

cross( v( "/net9" ) 2.5 2 ‘rising )
Gets the time value (X axis) corresponding to specified voltage "/net9"=2.5V (Y axis) for the second rising edge.

cross( v( "/net9" ) 1.2 1 'either ')

Gets the time value (X axis) corresponding to specified voltage "/net9"=1.2V (Y axis) for the first edge, which can be a rising or falling edge.
**db10**

\( \text{db10}(\{\text{o\_waveform} \mid \text{n\_number}\}) \Rightarrow \text{o\_waveform}/\text{n\_number}/\text{nil} \)

**Description**

Returns 10 times the log10 of the specified waveform object or number.

**Arguments**

- **o\_waveform**: Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)
- **n\_number**: Number.

**Value Returned**

- **o\_waveform**: Returns a waveform object if the input argument is a waveform object or returns a family of waveforms if the input argument is a family of waveforms.
- **n\_number**: Returns a number if the input argument is a number.
- **nil**: Returns nil and an error message otherwise.

**Examples**

\( \text{db10}(\ \text{ymax}(\ v( \ "/\text{net9}\" )\ )\ ) \)

*Returns a waveform representing* \( \log_{10}(\text{ymax}(\ v("/\text{net9}\")) \) multiplied by 10.*

\( \text{db10}(\ 1000\ ) \Rightarrow 30.0 \)

*Gets the value* \( \log_{10}(1000) \) *multiplied by 10, or 30.*
db20

\[ \text{db20}( \{ \text{o\_waveform} \mid \text{n\_number} \} ) \Rightarrow \text{o\_waveform/\text{n\_number/nil} } \]

**Description**

Returns 20 times the log10 of the specified waveform object or number.

**Arguments**

- **o\_waveform**: Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)
- **n\_number**: Number.

**Value Returned**

- **o\_waveform**: Returns a waveform object if the input argument is a waveform object or returns a family of waveforms if the input argument is a family of waveforms.
- **n\_number**: Returns a number if the input argument is a number.
- **nil**: Returns nil and an error message otherwise.

**Examples**

\[ \text{db20}( \text{ymax}( \text{v( } \text{"/net9" } ) ) ) \]

Returns a waveform representing 20 times \[ \log_{10}(\text{ymax(\text{v("/net9")})}) \].

\[ \text{db20}( 1000 ) \Rightarrow 60.0 \]

Returns the value of 20 times \[ \log_{10}(1000) \], or 60.
dbm

\[ dbm( \{ o\_waveform | n\_number \} ) \Rightarrow o\_waveform/n\_number/nil \]

**Description**

Returns 10 times the log10 of the specified waveform object plus 30.

**Arguments**

- **o\_waveform**: Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)

- **n\_number**: Number.

**Value Returned**

- **o\_waveform**: Returns a waveform object if the input argument is a waveform object or returns a family of waveforms if the input argument is a family of waveforms.

- **n\_number**: Returns a number if the input argument is a number.

- **nil**: Returns nil and an error message otherwise.

**Example**

\[ dbm( \text{ymax}( v( "\text{/net9}" ) ) ) \]

Returns a waveform representing 10 times \( \log_{10}(\text{ymax}(v("/net9"))) \) plus 30.
**delay**

```plaintext
delay( ?wf1 o_wf1 ?value1 n_value1 ?edge1 s_edge1 ?nth1 x_nth1 ?td1 n_td1 ?wf2 o_wf2 ?value2 n_value2 ?edge2 s_edge2 ?nth2 x_nth2 {
  [?td2 n_td2] | [?td2r0 n_td2r0] } ?stop n_stop ) =>
o_waveform/n_value/nil
```

**Description**
Calculates the delay between a trigger event and a target event.

The `delay` command computes the delay between two points using the `cross` command.

**Arguments**

- **o_wf1**  
  First waveform object.

- **n_value1**  
  Value at which the crossing is significant for the first waveform object.

- **s_edge1**  
  Type of the edge that must cross `n_value1`.  
  Valid values: `'rising', 'falling', either`

- **x_nth1**  
  Number that specifies which crossing is to be the trigger event.  
  For example, if `x_nth1` is 2, the trigger event is the second edge of the first waveform with the specified type that crosses `n_value1`.

- **n_td1**  
  Time at which to start the delay measurement. The simulator begins looking for the trigger event, as defined by `o_wf1`, `n_value1`, `t_edge1`, and `x_nth1`, only after the `n_td1` time is reached.

- **o_wf2**  
  Second waveform object.

- **n_value2**  
  Value at which the crossing is significant for the second waveform.

- **s_edge2**  
  Type of the edge for the second waveform.  
  Valid values: `'rising', 'falling', either`
**x_nth2**

Number that specifies which crossing is to be the target event. For example, if `x_nth2` is 2, the target event is the second edge of the second waveform with the specified type that crosses `n_value2`.

**n_td2**

Time to start observing the target event. `n_td2` is specified relative to the trigger event. This parameter cannot be specified at the same time as `n_td2r0`.

The simulator begins looking for the target event, as defined by `o_wf2`, `n_value2`, `t_edge2`, and `x_nth2`, only after the `n_td2` time is reached.

If you specify neither `n_td2` nor `n_td2r0`, the simulator begins looking for the target event at `t = 0`.

**n_td2r0**

Time to start observing the target event, relative to `t = 0`. Only applicable if both `o_wf1` and `o_wf2` are specified. This parameter cannot be specified at the same time with `n_td2`.

The simulator begins looking for the target event, as defined by `o_wf2`, `n_value2`, `t_edge2`, and `x_nth2`, only after the `n_td2r0` time is reached.

If you specify neither `n_td2` nor `n_td2r0`, the simulator begins looking for the target event at `t = 0`.

**n_stop**

Time to stop observing the target event.

**Value Returned**

**o_waveform**

Returns a waveform representing the delay if the input argument is a family of waveforms.

**n_value**

Returns the delay value if the input argument is a single waveform.

**nil**

Returns `nil` and an error message otherwise.
**Examples**

delay( ?wf1 wf1 ?value1 2.5 ?nth1 2 ?edge1 'either ?wf2 wf2 ?value2 2.5 ?nth2 1 ?edge2 'falling )

Calculates the delay starting from the time when the second edge of the first waveform reaches the value of 2.5 to the time when the first falling edge of the second waveform crosses 2.5.

delay( ?tdl 5 ?wf2 wf2 ?value2 2.5 ?nth2 1 ?edge2 'rising ?td2.5)

Calculates the delay starting when the time equals 5 seconds and stopping when the value of the second waveform reaches 2.5 on the first rising edge 5 seconds after the trigger.

delay( ?wf1 wf2 ?value1 2.5 ?nth1 1 ?edge1 'rising ?tdl 5 ?wf2 wf2 ?value2 2.5 ?nth2 2 ?edge2 'rising )

Waits until after the time equals 5 seconds, and calculates the delay between the first and the second rising edges of \textit{wf2} when the voltage values reach 2.5.
deriv

deriv( o_waveform ) => o_waveform/nil

Description

Computes the derivative of a waveform with respect to the X axis.

**Note:** After the second derivative, the results become inaccurate because the derivative is obtained numerically.

Arguments

<table>
<thead>
<tr>
<th>o_waveform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)</td>
</tr>
</tbody>
</table>

Value Returned

<table>
<thead>
<tr>
<th>o_waveform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns a waveform object representing the derivative with respect to the X axis of the input waveform. Returns a family of waveforms if the input argument is a family of waveforms.</td>
</tr>
<tr>
<td>nil</td>
</tr>
<tr>
<td>Returns nil and an error message otherwise.</td>
</tr>
</tbody>
</table>

Example

```plaintext
plot( deriv( v( "/net8" ) ) )
```

Plots the waveform representing the derivative of the voltage of "/net8".
dft

dft( o_waveform n_from n_to x_num [t_windowName [n_param1]] ) =>
o_waveform/nil

Description

Computes the discrete Fourier transform and fast Fourier transform of the input waveform.

The waveform is sampled at the following \( n \) timepoints:

\[
\text{from}, \text{from} + \delta t, \text{from} + 2 \times \delta t, \ldots, \\
\text{from} + (N - 1) \times \delta t
\]

The output of dft is a frequency waveform, \( W(f) \), which has \((N/2 + 1)\) complex values—the DC term, the fundamental, and \((N/2 - 1)\) harmonics.

**Note:** The last time point, \((\text{from} + (N - 1) \times \delta t)\), is \((\text{to} - \delta t)\) rather than \(\text{to}\). The dft command assumes that \(w(\text{from})\) equals \(w(\text{to})\).

Arguments

\(o\_waveform\)  
Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

\(n\_from\)  
Starting value for the dft computation.

\(n\_to\)  
Ending value for the dft computation.

\(x\_num\)  
Number of timepoints. If \(x\_num\) is not a power of 2, it is forced to be the next higher power of 2.

\(t\_windowName\)  
Variable representing different methods for taking a dft computation.  
**Valid values:** Rectangular, ExtCosBell, HalfCycleSine, Hanning or Cosine2, Triangle or Triangular, Half3CycleSine or HalfCycleSine3, Hamming, Cosine4, Parzen, Half6CycleSine or HalfCycleSine6, Blackman, or Kaiser.
For more information about `windowName`, see the information about Discrete Fourier Transform (dft) in the *Affirma Analog Circuit Design Environment User Guide*.

```
n_param1
```

Smoothing parameter.
Applies only if the `t_windowName` argument is set to Kaiser.

**Value Returned**

```
o_waveform
```

Returns a waveform representing the magnitude of the various harmonics for the specified range of frequencies. Returns a family of waveforms if the input argument is a family of waveforms.

```
nil
```

Returns nil and an error message otherwise.

**Example**

```
plot( dft( v( "/net8" ) 10u 20m 64 "rectangular" ) )
```

Computes the discrete Fourier transform, fast Fourier transform, of the waveform representing the voltage of "/net8". The computation is done from 10u to 20m with 64 timepoints. The resulting waveform is plotted.
flip

flip( o_w waveform ) => o_w waveform/nil

Description

Returns a waveform with the X vector values negated.

Arguments

o_w waveform

Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drw wave:XXXXX.)

Value Returned

o_w waveform

Returns a waveform object representing the input waveform mirrored about its Y axis. Returns a family of waveforms if the input argument is a family of waveforms.

nil

Returns nil and an error message otherwise.

Example

plot( flip( v("/net4") ) )

Plots the waveform for the voltage of "/net4" with the X vector values negated.
fourEval

fourEval( o_waveform n_from n_to n_by ) => o_waveform/nil

Description

Evaluates the Fourier series represented by an expression.

This function is an inverse Fourier transformation and thus the inverse of the dft command. The fourEval function transforms the expression from the frequency domain to the time domain.

Arguments

- **o_waveform**: Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

- **n_from**: Starting point on the X axis at which to start the evaluation.

- **n_to**: Increment.

- **n_by**: Ending point on the X axis.

Value Returned

- **o_waveform**: Returns a waveform object representing the inverse Fourier transformation of the input waveform. Returns a family of waveforms if the input argument is a family of waveforms.

- **nil**: Returns nil and an error message otherwise.

Example

```
plot( fourEval( v( "/net3" ) 1k 10k 10 )
```

Plots the waveform representing the inverse Fourier transformation of the voltage of "/net3" from 1k to 10k.
frequency

frequency( o_waveform ) => o_waveform/n_value/nil

Description

Computes the reciprocal of the average time between two successive midpoint crossings of
the rising waveform.

Arguments

o_waveform Waveform object representing simulation results that can be
displayed as a series of points on a grid. (A waveform object
identifier looks like this: drwave:XXXXX.)

Value Returned

o_waveform Returns a waveform representing the frequency of a family of
waveforms if the input argument is a family of waveforms.

n_value Returns a number representing the frequency of the specified
waveform.

nil Returns nil and an error message otherwise.

Example

frequency( v("/net12") )

Returns the frequency of "/net12".
ga

\[
\text{ga} (\ o_s11 \ o_s12 \ o_s21 \ o_s22 \ [\ ?gs \ n_gs]) \Rightarrow o\_\text{waveform}/\text{nil}
\]

**Description**

Returns the available gain in terms of the supplied parameters and the optional source reflection coefficient (Gs).

**Arguments**

- \(o\_s11\) Waveform object representing \(s11\).
- \(o\_s12\) Waveform object representing \(s12\).
- \(o\_s21\) Waveform object representing \(s21\).
- \(o\_s22\) Waveform object representing \(s22\).
- \(n\_gs\) Source reflection coefficient.
  Default value: 0

**Value Returned**

- \(o\_\text{waveform}\) Waveform object representing the available gain.
- \(\text{nil}\) Returns \(\text{nil}\) and an error message otherwise.

**Examples**

\[
\begin{align*}
s11 &= \text{sp}(1 \ 1) \\
s12 &= \text{sp}(1 \ 2) \\
s21 &= \text{sp}(2 \ 1) \\
s22 &= \text{sp}(2 \ 2) \\
\text{plot}(\text{ga}(s11 \ s12 \ s21 \ s22))
\end{align*}
\]
gac

\[
gac( o_{s11} \ o_{s12} \ o_{s21} \ o_{s22} \ g_{\text{level}} \ g_{\text{frequency}} ) \to o_{\text{waveform/nil}}
\]

**Description**

Computes the available gain circles.

The \( g \) data type on \( g_{\text{level}} \) and \( g_{\text{frequency}} \) allows either the level or the frequency to be swept while the other remains fixed.

**Arguments**

- **\( o_{s11} \)**: Waveform object representing \( s_{11} \).
- **\( o_{s12} \)**: Waveform object representing \( s_{12} \).
- **\( o_{s21} \)**: Waveform object representing \( s_{21} \).
- **\( o_{s22} \)**: Waveform object representing \( s_{22} \).
- **\( g_{\text{level}} \)**: Level in dB. It can be specified as a scalar or a vector. If it is specified as a vector, the level is swept. The \( \text{linRg} \) function can be called to generate a linear range. For example, \( \text{linRg}( -30 \ 30 \ 5 ) \) is the same as \( \text{list}( -30 \ -25 \ -20 \ -15 \ -10 \ -5 \ 0 \ 5 \ 10 \ 15 \ 20 \ 25 \ 30 ) \) and the \( g_{\text{level}} \) argument can be specified as either of the above. In that case, an available gain circle is calculated at each one of the 13 levels.
- **\( g_{\text{frequency}} \)**: Frequency, which can be specified as a scalar or a linear range. If it is specified as a linear range, the frequency is swept. The linear range is specified as a list with three values: the start of the range, the end of the range, and the increment. For example, \( \text{list}(100\text{M} \ 1\text{G} \ 100\text{M}) \) specifies a linear range with the following values:

\[
\{ 100\text{M}, 200\text{M}, 300\text{M}, 400\text{M}, 500\text{M}, 600\text{M}, 700\text{M}, 800\text{M}, 900\text{M}, 1\text{G} \}
\]

In that case, an available gain circle is calculated at each one of the 10 frequencies.
Value Returned

o_waveform  Waveform object representing the available gain circles.
nil  Returns nil and an error message otherwise.

Examples

s11 = sp(1 1 ?result "sp")
s12 = sp(1 2 ?result "sp")
s21 = sp(2 1 ?result "sp")
s22 = sp(2 2 ?result "sp")
plot(gac(s11 s12 s21 s22 linRg(-30 30 5) 900M))
gainBwProd

gainBwProd( o_waveform ) => o_waveform/n_value/nil

Description

Calculates the gain-bandwidth product of a waveform representing the frequency response of interest over a sufficiently large frequency range.

Returns the product of the zero-frequency-gain and 3dB-gain-frequency.

\[ \text{gainBwProd} \ (gain) = A_o \cdot f_\text{2} \]

The gain-bandwidth product is calculated as the product of the DC gain \( A_o \) and the critical frequency \( f_\text{2} \). The critical frequency \( f_\text{2} \) is the smallest frequency for which the gain equals \( 1/\sqrt{2} \) times the DC gain \( A_o \).

Arguments

- \( o\_\text{waveform} \) : Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)

Value Returned

- \( o\_\text{waveform} \) : Returns a waveform representing the gain-bandwidth product for a family of waveforms if the input argument is a family of waveforms.

- \( n\_\text{value} \) : Returns a value for the gain-bandwidth product for the specified waveform.

- \( \text{nil} \) : Returns \text{nil} and an error message otherwise.
Example

gainBwProd( v( "OUT" ) )

Returns the gain-bandwidth product for the waveform representing the voltage of the "OUT" net.
gainMargin

gainMargin( o_waveform ) => o_waveform/n_value/nil

Description

Computes the gain margin of the loop gain of an amplifier.

It requires one argument, a waveform representing the loop gain of interest over a sufficiently large frequency range. This command returns the dB value of the waveform when its phase crosses negative pi.

\[
\text{gainMargin( gain ) = } 20 \times \log_{10}( \text{value( gain } f0 ) )
\]

The gain margin is calculated as the magnitude of the gain in dB at \( f0 \). The frequency \( f0 \) is the lowest frequency in which the phase of the gain provided is -180 degrees. For stability, the gain margin must be less than 0 dB.

Arguments

- **o_waveform**: Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)

Value Returned

- **o_waveform**: Returns a waveform representing the gain margin for a family of waveforms if the input argument is a family of waveforms.
- **n_value**: Returns the value for the gain margin of the specified waveform.
- **nil**: Returns nil and an error message otherwise.

Example

gainMargin( v( "/OUT" ) )

Returns the gain margin for the waveform representing the voltage of the "/OUT" net.
**gmax**

gmax( o_s11 o_s12 o_s21 o_s22 ) => o_waveform/nil

**Description**

Returns the maximum power gain in terms of the supplied parameters.

**Arguments**

- **o_s11**: Waveform object representing s11.
- **o_s12**: Waveform object representing s12.
- **o_s21**: Waveform object representing s21.
- **o_s22**: Waveform object representing s22.

**Value Returned**

- **o_waveform**: Load reflection coefficient.
- **nil**: Returns nil and an error message otherwise.

**Examples**

```plaintext
s11 = sp(1 1)
s12 = sp(1 2)
s21 = sp(2 1)
s22 = sp(2 2)
plot(gmax(s11 s12 s21 s22))
```
gmin

gmin( o_Gopt o_Bopt f_zref ) => o_gminWave/nil

Description

Returns the optimum noise reflection coefficient in terms of o_Gopt, o_Bopt, and f_zref.

gmin is returned as follows:

yOpt = o_Gopt + (complex 0 1) * o_Bopt
return ( 1 / f_zref(1) - yOpt ) / ( 1 / f_zref(1) + yOpt )

Arguments

- o_Gopt: Waveform object representing the optimum source conductance.
- o_Bopt: Waveform object representing the optimum source susceptance.
- f_zref: Reference impedance.

Value Returned

- o_gminWave: Waveform object representing the optimum noise reflection coefficient.
- nil: Returns nil and an error message otherwise.

Examples

Gopt = getData("Gopt")
Bopt = getData("Bopt")
Zref = zref(1 ?result "sp")
plot(gmin(Gopt Bopt Zref))
gmsg

\[ gmsg( o\_s11 \ o\_s12 \ o\_s21 \ o\_s22 ) => o\_waveform/nil \]

**Description**

Returns the maximum stable power gain in terms of the supplied parameters.

**Arguments**

- \( o\_s11 \): Waveform object representing \( s11 \).
- \( o\_s12 \): Waveform object representing \( s12 \).
- \( o\_s21 \): Waveform object representing \( s21 \).
- \( o\_s22 \): Waveform object representing \( s22 \).

**Value Returned**

- \( o\_waveform \): Waveform object representing the maximum stable power gain.
- \( \text{nil} \): Returns \( \text{nil} \) and an error message otherwise.

**Example**

\[
\text{s11} = \text{sp}(1 \ 1) \\
\text{s12} = \text{sp}(1 \ 2) \\
\text{s21} = \text{sp}(2 \ 1) \\
\text{s22} = \text{sp}(2 \ 2) \\
\text{plot}(\text{gmsg(s11 s12 s21 s22)})
\]
gmux

gmux( o_s11 o_s12 o_s21 o_s22 ) => o_waveform/nil

Description
Returns the maximum unilateral power gain in terms of the supplied parameters.

Arguments
- o_s11: Waveform object representing s11.
- o_s12: Waveform object representing s12.
- o_s21: Waveform object representing s21.
- o_s22: Waveform object representing s22.

Value Returned
- o_waveform: Waveform object representing the maximum unilateral power gain.
- nil: Returns nil and an error message otherwise.

Examples
s11 = sp(1 1)
s12 = sp(1 2)
s21 = sp(2 1)
s22 = sp(2 2)
plot(gmux(s11 s12 s21 s22))
gp

gp( o_s11 o_s12 o_s21 o_s22 [?gl n_gl] ) => o_waveform/nil

Description

Computes the power gain in terms of the S-parameters.

Arguments

  o_s11                   Waveform object representing s11.
  o_s12                   Waveform object representing s12.
  o_s21                   Waveform object representing s21.
  o_s22                   Load reflection coefficient. Default value: 0

Value Returned

  o_waveform               Waveform object representing the power gain.
  nil                      Returns nil and an error message otherwise.

Example

  s11 = sp(1 1)
  s12 = sp(1 2)
  s21 = sp(2 1)
  s22 = sp(2 2)
  plot(gp(s11 s12 s21 s22))
gpc

\[ \text{gpc} \left( o_{s11} \ o_{s12} \ o_{s21} \ o_{s22} \ g_{\text{level}} \ g_{\text{frequency}} \right) \Rightarrow o_{\text{waveform/nil}} \]

**Description**

Computes the power gain circles.

The \( g \) datatype on \( g_{\text{level}} \) and \( g_{\text{frequency}} \) allows either the level or the frequency to be swept while the other remains fixed.

**Arguments**

- **o\_s11**
  - Waveform object representing \( s11 \).

- **o\_s12**
  - Waveform object representing \( s12 \).

- **o\_s21**
  - Waveform object representing \( s21 \).

- **o\_s22**
  - Waveform object representing \( s22 \).

- **g\_level**
  - Level in dB. It can be specified as a scalar or a vector. If it is specified as a vector, the level is swept. The \( \text{linRg} \) function can be called to generate a linear range. For example, \( \text{linRg} \left( -30 \ 30 \ 5 \right) \) is the same as \( \text{list}\left(-30 \ -25 \ -20 \ -15 \ -10 \ -5 \ 0 \ 5 \ 10 \ 15 \ 20 \ 25 \ 30\right) \) and the \( g_{\text{level}} \) argument can be specified as either. In that case, a power gain circle is calculated at each one of the 13 levels.

- **g\_frequency**
  - The frequency. It can be specified as a scalar or a linear range. If it is specified as a linear range, the frequency is swept. The linear range is specified as a list with three values: the start of the range, the end of the range, and the increment. For example, \( \text{list}(100\text{M} \ 1\text{G} \ 100\text{M}) \) specifies a linear range with the following values:

\[ \{100\text{M}, 200\text{M}, 300\text{M}, 400\text{M}, 500\text{M}, 600\text{M}, 700\text{M}, 800\text{M}, 900\text{M}, 1\text{G}\} \]

In that case, a power gain circle is calculated at each one of the 10 frequencies.
### Value Returned

<table>
<thead>
<tr>
<th><strong>Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>o_waveform</code></td>
<td>Waveform object representing the power gain circles.</td>
</tr>
<tr>
<td><code>nil</code></td>
<td>Returns <code>nil</code> and an error message otherwise.</td>
</tr>
</tbody>
</table>
**groupDelay**

`groupDelay( o_waveform ) => o_waveform/nil`

**Description**

Computes the group delay of a waveform.

This command returns the derivative of the phase of `o_waveform/2pi`. Group delay is defined as the derivative of the phase with respect to frequency. Group delay is expressed in seconds.

It is calculated using the `vp` function as shown below:

\[
Group\ Delay = \frac{d\phi}{d\omega} = \frac{d}{df}\left[\frac{\text{phase}(/netX)}{360}\right]
\]

**Arguments**

- `o_waveform` Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

**Value Returned**

- `o_waveform` Returns a waveform representing the group delay of the specified waveform.
- `nil` Returns nil and an error message otherwise.

**Example**

`plot( groupDelay( v( "/net3" ) ) )`

Plots the waveform representing the group delay of the voltage of "/net3".
gt

gt( o_s11 o_s12 o_s21 o_s22 [ ?gs n_gs] [ ?gl n_gl] ) =>
    o_waveform/nil

Description

Returns the transducer gain in terms of the supplied parameters and the optional source reflection coefficient (Gs) and the input reflection coefficient (Gi).

Arguments

- o_s11: Waveform object representing s11.
- o_s12: Waveform object representing s12.
- o_s22: Waveform object representing s14.
- n_gs: Source reflection coefficient. Default value: 0
- n_gl: Input reflection coefficient. Default value: 0

Value Returned

- o_waveform: Waveform object representing the transducer gain.
- nil: Returns nil and displays a message if there is an error.

Examples

s11 = sp(1 1)
s12 = sp(1 2)
s21 = sp(2 1)
s22 = sp(2 2)
plot(gt(s11 s12 s21 s22))
harmonic

harmonic( o_waveform h_index ) => o_waveform/g_value/nil

Description

Returns the waveform for a given harmonic index.

Arguments

- **o_waveform**: Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

- **h_index**: The index number that designates the harmonic information to be returned. For the `'pss', 'pac', and 'pxf` analyses, the index is an integer number. For the `'pdisto` analysis, the index is a list of integers that correspond with the frequency names listed in the `funds` analysis parameter in the netlist. If more than one `h_index` is desired at one time, a list can be specified.

Value Returned

- **o_waveform**: Returns a waveform (when a single `h_index` is specified) or family of waveforms (when more than one `h_index` is specified) if the input argument is a family of waveforms.

- **g_value**: Returns the harmonic value if the input is a single waveform with the X values being harmonics.

- **nil**: Returns nil and displays a message if there is an error.

Examples

For each of the following commands:

```python
harmonic(v("/net49" ?result "pss-fd.pss") 1)
harmonic(v("/Pif" ?result "pdisto-fi.pdisto") list(1 -1))
```

Each result is a complex number.
For each of the following commands:

\[
\text{harmonic}(v("/net54" \ ?result \ "pac-pac") \ 1) \\
\text{harmonic}(v("/net51" \ ?result \ "sweepss_pss_fd-sweep") \ \text{list}(8)) \\
\text{harmonic}(v("/Pif" \ ?result \ "sweepss_pac-sweep") \ -8) \\
\text{harmonic}(v("/net36" \ ?result \ "sweeppdisto_pdisto_completion") \ '(1 -1))
\]

Each result is a waveform.

For each of the following commands:

\[
\text{harmonic}(v("/net54" \ ?result \ "pac-pac") \ \text{list}(1 \ 5)) \\
\text{harmonic}(v("/net51" \ ?result \ "sweepss_pss_fd-sweep") \ '(1 \ 8)) \\
\text{harmonic}(v("/Pif" \ ?result \ "sweepss_pac-sweep") \ \text{list}(-8 \ 0)) \\
\text{harmonic}(v("/net36" \ ?result \ "sweeppdisto_pdisto_completion") \ '((1 \ -1) \ (2 \ -2) \ (-1 \ 2)))
\]

Each result is a family of waveforms.

Neither of the following commands should be entered:

\[
\text{harmonic}(v("/net49" \ ?result \ "pss-fd.pss") \ \text{list}(0 \ 1)) \\
\text{harmonic}(v("/Pif" \ ?result \ "pdisto-fi.pdisto") \ '((1 -1) \ (-1 \ 2)))
\]

Each resulting waveform is not in a useful format.
**harmonicFreqList**

\[
\text{harmonicFreqList( } \text{[?resultsDir t_resultsDir]} \text{[?result S_resultName]}) \Rightarrow n\text{list/nil}
\]

**Description**

Returns a list of lists, with each sublist containing a harmonic index and the minimum and maximum frequency values that the particular harmonic ranges between.

If both of these frequency values are the same, just one frequency value is returned.

**Arguments**

- **t_resultsDir**
  Directory containing the PSF files (results). If you supply this argument, you must also supply the **resultName** argument.

- **S_resultName**
  Results from an analysis.

**Value Returned**

- **n_list**
  Returns a list of lists. For the ‘pss, ‘pac, and ‘pxf analyses, the first element of each sublist is an integer number. For the ‘pdisto analysis, the first element of each sublist is a list of integers that correspond with the frequency names listed in the funds analysis parameter in the netlist. For all sublists, the remaining entries are the minimum and maximum frequency values that the particular harmonic ranges between. If both of these frequency values are the same, just one frequency value is returned.

- **nil**
  Returns nil if no harmonics are found in the data.

**Examples**

For each of the following commands:

- `harmonicFreqList( ?result "pss-fd.pss" )`
- `harmonicFreqList( ?result "pac-pac" )`
- `harmonicFreqList( ?result "sweeppss_pss_fd-sweep" )`
harmonicFreqList( ?result "sweepss_pac-sweep" )

Each result is a list of integers.

For each of the following commands:
harmonicFreqList( ?result "pdisto-fi.pdisto" )
harmonicFreqList( ?result "sweeppdisto_pdisto fi-sweep" )

Each result is a list of lists, with each sublist containing a combination of integer numbers that correspond with the frequency names listed in the funds analysis parameter in the netlist. These names can also be extracted from the PSF data by using the resultParam function to find the 'largefundname' and 'moderatefundnames' values. For example:

```
strcat(resultParam( 'largefundname ?result "pdisto-fi.pdisto" ) " "
resultParam( 'moderatefundnames ?result "pdisto-fi.pdisto" ))
```

Returns a string representing the order of the frequency names.
**harmonicList**

```lisp
harmonicList( [?resultsDir t_resultsDir] [?result S_resultName] ) => n_list
```

**Description**

Returns the list of harmonic indices available in the `resultName` or current result data.

**Arguments**

- `t_resultsDir` Directory containing the PSF files (results). If you supply this argument, you must also supply the `resultName` argument.
- `S_resultName` Results from an analysis.

**Value Returned**

- `n_list` Returns a list of harmonic indices. For the ‘pss‘, ‘pac‘, and ‘pxf‘ analyses, the index is an integer number. For the ‘pdisto‘ analysis, the index is a list of integers that correspond with the frequency names listed in the ‘funds‘ analysis parameter in the netlist.
- `nil` Returns `nil` if no harmonics are found in the data.

**Examples**

For each of the following commands:

```ruby
harmonicList( ?result "pss-fd.pss" )
harmonicList( ?result "pac-pac" )
harmonicList( ?result "sweeppss_pss_fd-sweep" )
harmonicList( ?result "sweeppss_pac-sweep" )
```

Each result is a list of integers.

For each of the following commands:

```ruby
harmonicList( ?result "pdisto-fi.pdisto" )
harmonicList( ?result "sweeppdisto_pdisto_fi-sweep" )
```
Each result is a list of lists, with each sublist containing a combination of integer numbers that correspond with the frequency names listed in the ‘funds analysis parameter in the netlist. These names can also be extracted from the PSF data by using the ‘resultParam function to find the ‘largefundname and ‘moderatefundnames values. For example:

```plaintext
strcat(resultParam( 'largefundname ?result "pdisto-fi.pdisto" ) " "
resultParam( 'moderatefundnames ?result "pdisto-fi.pdisto" ))
```

Returns a string representing the order of the frequency names.
iinteg

\text{\texttt{iinteg(o\_waveform) \Rightarrow o\_waveform/nil}}

\textbf{Description}

Computes the indefinite integral of a waveform with respect to the X-axis variable.

\textbf{Arguments}

- \textit{o\_waveform} Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \texttt{drwave:XXXXX}.)

\textbf{Value Returned}

- \textit{o\_waveform} Returns a waveform representing the indefinite integral of the input waveform.
- \texttt{nil} Returns \texttt{nil} and an error message otherwise.

\textbf{Example}

\texttt{plot(iinteg(v("/net8"))})

Computes the indefinite integral of the waveform representing the voltage of "/net8".
**imag**

\[ \text{imag( \{ o\_waveform | n\_input \} ) => o\_waveformImag/n\_numberImag/nil} \]

**Description**

Returns the imaginary part of a waveform representing a complex number or returns the imaginary part of a complex number.

**Arguments**

- **o\_waveform**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)

- **n\_input**
  Complex number.

**Value Returned**

- **o\_waveformImag**
  Returns a waveform when the input argument is a waveform.

- **n\_numberImag**
  Returns a number when the input argument is a number.

- **nil**
  Returns \text{nil} and an error message otherwise.

**Examples**

\[ \text{imag( v( "/net8" ) )} \]

Returns a waveform representing the imaginary part of the voltage of "/net8". You also can use the \text{vim alias} to perform the same command, as in \text{vim( "net8" )}.  

\[ x=\text{complex( -1 -2 ) => complex( -1, -2) } \]

\[ \text{imag( x ) => -2.0} \]

Creates a variable \( x \) representing a complex number, and returns the real portion of that complex number.
**integ**

\[
\text{integ( o\_waveform ) \Rightarrow o\_waveform/n\_value/nil}
\]

**Description**

Computes the definite integral of the waveform with respect to the X axis.

The result is the value of the area under the curve over the range of the expression.

**Arguments**

- **o\_waveform**: Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

**Value Returned**

- **o\_waveform**: Returns a waveform representing the definite integral for a family of waveforms if the input argument is a family of waveforms.
- **n\_value**: Returns a numerical value representing the integral of the input waveform if the input argument is a single waveform.
- **nil**: Returns nil and an error message otherwise.

**Example**

\[
\text{integ( v( "/net8" ) )}
\]

Returns the definite integral of the waveform representing the voltage of "/net8".
\textbf{ipn}

\texttt{ipn\(\ (\ o\_spurious\ o\_reference\ \ [\ f\_ordspur\ f\_ordref\ f\_epspur\ f\_epref\ b\_psweep\ s\_measure\ \] ) \Rightarrow o\_waveform/f\_number/nil\}

**Description**

Performs an intermodulation \textit{n-th} order intercept measurement.

The data for this measurement can be either a single input power value or a parametric input power sweep.

From each of the spurious and reference power waveforms (or points), the \texttt{ipn} function extrapolates a line of constant slope (dB/dB) according to the specified order and input power level. These lines represent constant small-signal power gain (ideal gain). The \texttt{ipn} function calculates the intersection of these two lines and returns the value of either the X coordinate (input referred) or Y coordinate.

**Arguments**

\begin{itemize}
  \item \texttt{o\_spurious} \quad Waveform or number representing the spurious output power (in dBm).
  \item \texttt{o\_reference} \quad Waveform or number representing the reference output power (in dBm).
  \item \texttt{f\_ordspur} \quad Order or slope of the spurious constant-slope power line. Default value: 3
  \item \texttt{f\_ordref} \quad Order or slope of the reference constant-slope power line. Default value: 1
  \item \texttt{f\_epspur} \quad Value (in dBm) used to indicate the point where the spurious constant-slope power line begins. If \texttt{b\_psweep} is \texttt{t}, this value is the input power value of the point on the \texttt{o\_spurious} waveform, otherwise this value is paired with the \texttt{o\_spurious} value to define the point. This point should be in the linear region of operation. (If \texttt{b\_psweep} is \texttt{t}, \texttt{f\_spspur} defaults to the X coordinate of the first point of the \texttt{o\_spurious} wave; if \texttt{s\_measure} is ’input, a number must be specified.)
  \item \texttt{f\_epref} \quad Value (in dBm) used to indicate the point where the reference constant-slope power line begins. If \texttt{b\_psweep} is \texttt{t}, this value
\end{itemize}
is the input power value of the point on the o_reference waveform, otherwise this value is paired with the o_reference value to define the point. This point should be in the linear region of operation. (If b_psweep is t, f_epref defaults to the X coordinate of the first point of the o_reference wave; if s_measure is ’input, a number must be specified.)

**b_psweep**

Boolean indicating that the input power to the circuit was a parametric sweep. The power sweep must both be in dBm and be performed at the lowest parametric level.

Default value: t

**s_measure**

Name indicating if measurement is to be input referred (’input) or output referred (’output).

Default value: ’input

**Value Returned**

**o_waveform**

Depending on setting of b_psweep and the dimension of the input waveforms, returns either a waveform or a family of waveforms.

**f_number**

If o_spurious and o_reference are numbers or they are waveforms when b_psweep is t, returns a number.

**nil**

Returns nil and an error message otherwise.

**Examples**

spurWave = db20(harmonic(wave signalHarmonic))
refWave = db20(harmonic(wave referenceHarmonic))
xloc = ipn( spurWave refWave 3.0 1.0 -25 -25 )
yloc = ipn( spurWave refWave 3.0 1.0 -25 -25 t "Output")

Computes the IP3 point for the given wave.

Each of the following examples returns an ip3 measurement.

ipn(dB20(harmonic(v("/Pif" ?result "pss_fd") 9))
   dB20(harmonic(v("/Pif" ?result "pss_fd") 8)))
ipn(dbm(harmonic(spectralPower(v("/Pif" ?result "pss_fd")/50.0
   v("/Pif" ?result "pss_fd")) 9))
OCEAN Reference
Predefined Functions and Waveform (Calculator) Functions

\[
dbm(harmonic(spectralPower(v("/Pif" ?result "pss_fd")/50.0 v("/Pif" ?result "pss_fd"))) 8))
\]

\[
\text{ipn(dbm(harmonic(spectralPower(v("/Pif" ?result "pss_fd") \rightarrow resultParam("rif:r" ?result "pss_td") v("/Pif" ?result "pss_fd")) 9)) dbm(harmonic(spectralPower(v("/Pif" ?result "pss_fd") \rightarrow resultParam("rif:r" ?result "pss_td") v("/Pif" ?result "pss_fd")) 8)))}
\]

\[
\text{ipn(dbm(harmonic(spectralPower(i("/rif/PLUS" ?result "pss_fd") v("/Pif" ?result "pss_fd")) 9)) dbm(harmonic(spectralPower(i("/rif/PLUS" ?result "pss_fd") v("/Pif" ?result "pss_fd")) 8)) t "Output")}
\]

\[
\text{ipn(dbm(harmonic(spectralPower(v("/Pif" ?result "pac") \rightarrow resultParam("rif:r" ?result "pss_td") v("/Pif" ?result "pac"))) -21))}
\]

3. 1. -25 -25 t "Output"
ipnVRI

\[ \text{ipnVRI}( \text{o\_vport}, \text{x\_harmspur}, \text{x\_harmref}, [\text{iport}, \text{o\_iport}], [\text{f\_rport}], [\text{f\_ordspur}], [\text{f\_epoint}], [\text{f\_psweep}], [\text{f\_epref}], [\text{f\_ordref}], [\text{f\_measure}], \text{s\_measure}] ) \Rightarrow \text{o\_waveform}/\text{f\_number}/\text{nil} \]

**Description**

Performs an intermodulation \( n \)th-order intercept point measurement.

Use this function to simplify the declaration of an ipn measurement. This function extracts the spurious and reference harmonics from the input waveform(s), and uses \( \text{dBm}(\text{spectralPower}((\text{i} \text{ or \ v/r}),v)) \) to calculate the respective powers. The function passes these power curves or numbers and the remaining arguments to the ipn function to complete the measurement.

From each of the spurious and reference power waveforms (or points), the ipn function extrapolates a line of constant slope (dB/dB) according to the specified order and input power level. These lines represent constant small-signal power gain (ideal gain). The ipn function calculates the intersection of these two lines and returns the value of either the X coordinate (input referred) or the Y coordinate.

**Arguments**

- **o\_vport**
  - Voltage across the output port. This argument must be a family of spectrum waveforms (1 point per harmonic), with the option of containing a parametric input power sweep (in dBm).

- **x\_harmspur**
  - Harmonic number of the spurious voltage contained in \( \text{o\_vport} \). When \( \text{o\_iport} \) is specified, also applies to a current waveform contained in \( \text{o\_iport} \).

- **x\_harmref**
  - Harmonic index of the reference voltage contained in \( \text{o\_vport} \). When \( \text{o\_iport} \) is specified, also applies to a current waveform contained in \( \text{o\_iport} \).

- **o\_iport**
  - Current into the output port. This argument must be a family of spectrum waveforms (1 point per harmonic), with the option of containing a parametric input power sweep (in dBm). When specified, power is calculated using voltage and current.
**f_rport**  
Resistance into the output port. When specified and o_iport is nil, the output power is calculated using voltage and resistance.  
Default value: 50

**f_ordspur**  
Order or slope of the spurious constant-slope power line.  
Default value: 3

**f_epoint**  
Value (in dBm) used to indicate the point where the spurious constant-slope power line begins. If b_psweep is t, this value is the input power value of the point on the o_spurious waveform, otherwise this value is paired with the o_spurious value to define the point. This point should be in the linear region of operation.  
Default value: If b_psweep is t, the lowest input power value; if s_measure is ’input, a number must be specified.

**b_psweep**  
Boolean indicating that the input power to the circuit was a parametric sweep. The power sweep must be in dBm and must be performed at the lowest parametric level.  
Default value: t

**f_epref**  
Value (in dBm) used to indicate the point where the reference constant-slope power line begins. If b_psweep is t, this value is the input power value of the point on the o_reference waveform, otherwise this value is paired with the o_reference value to define the point. This point should be in the linear region of operation.  
Default value: If f_epoint is not nil, f_epoint; else if b_psweep is t, the X coordinate of the first point of the o_reference wave; else if s_measure is ’input, a number must be specified.

**f_ordref**  
Order or slope of the reference constant-slope power line.  
Default value: 1

**s_measure**  
Symbol indicating if measurement is to be input referred (’input) or output referred (’output).  
Default value: ’input
Value Returned

- **o_waveform**: Depending on the setting of `b_psweep` and the dimension of input waveform(s), the `ipnVRI` function returns either a waveform or a family of waveforms.

- **f_number**: Depending on the setting of `b_psweep` and the dimension of input waveform(s), the `ipnVRI` function returns a number.

- **nil**: Returns nil and an error message otherwise.

Example

Each of following examples returns an ip3 measurement:

```
ipnVRI(v("/Pif" ?result "pss_fd") 9 8)
ipnVRI(v("/Pif" ?result "pss_fd") 9 8 ?rport resultParam("rif:r" ?result "pss_td"))
ipnVRI(v("/Pif" ?result "pss_fd") 9 8 ?iport i("/rif/PLUS" ?result "pss_fd") ?epoint -25 ?measure "Output")
ipnVRI(v("/Pif" ?result "pac") -21 -25 ?rport resultParam("rif:r" ?result "pss_td"))
```
**ipnVRICurves**

```lisp
ipnVRICurves( o_vport x_harmspur x_harmref [?iport o_iport]
               [?rport f_rport] [?ordspur f_ordspur] [?epoint f_epoint]
               [?psweep b_psweep] [?epref f_epref] [?ordref f_ordref] ) =>
o_waveform/nil
```

**Description**

Constructs the waveforms associated with an ipn measurement.

Use this function to simplify the creation of waves associated with an ipn measurement. This function extracts the spurious and reference harmonics from the input waveform(s), and uses dBm(spectralPower((i or v/r),v)) to calculate the respective powers.

From each of the spurious and reference power waveforms (or points), the ipnVRICurves function extrapolates a line of constant slope (dB/dB) according to the specified order and input power level. These lines represent constant small-signal power gain (ideal gain). The function returns these lines and power waveforms (when present) as a family of waveforms.

This function only creates waveforms and does not perform an ipn measurement or include labels with the waveforms. Use the ipn or ipnVRI function for making measurements.

**Arguments**

- **o_vport**: Voltage across the output port. This argument must be a family of spectrum waveforms (1 point per harmonic), with the option of containing a parametric input power sweep (in dBm).

- **x_harmspur**: Harmonic index of the spurious voltage contained in o_vport. When o_iport is specified, also applies to a current waveform contained in o_iport.

- **x_harmref**: Harmonic index of the reference voltage contained in o_vport. When o_iport is specified, also applies to a current waveform contained in o_iport.

- **o_iport**: Current into the output port. This argument must be a family of spectrum waveforms (1 point per harmonic), with the option of containing a parametric input power sweep (in dBm). When specified, power is calculated using voltage and current.

- **f_rport**: Resistance into the output port. When specified and o_iport is nil, the output power is calculated using voltage and
resistance.
Default value: 50

*f_ordspur*  
Order or slope of the spurious constant-slope power line.
Default value: 3

*f_epoint*  
Value (in dBm) used to indicate the point where the spurious constant-slope power line begins. If *b_psweep* is `true`, this value is the input power value of the point on the *o_spurious* waveform, otherwise this value is paired with the *o_spurious* value to define the point. This point should be in the linear region of operation.
Default value: If *b_psweep* is `true`, the X coordinate of the first point of the *o_spurious* wave; otherwise a number must be specified.

*b_psweep*  
Boolean indicating that the input power to the circuit was a parametric sweep. The power sweep must be in dBm and must be performed at the lowest parametric level.
Default value: `true`

*f_epref*  
Value (in dBm) used to indicate the point where the reference constant-slope power line begins. If *b_psweep* is `true`, this value is the input power value of the point on the *o_reference* waveform, otherwise this value is paired with the *o_reference* value to define the point. This point should be in the linear region of operation.
Default value: If *f_epoint* is not `nil`, *f_epoint*; else if *b_psweep* is `true`, the X coordinate of the first point of the *o_reference* wave; else a number must be specified.

*f_ordref*  
Order or slope of the reference constant-slope power line.
Default value: 1

Value Returned

*o_waveform*  
A family of waveforms that contains the spurious and reference tangent lines, and when *b_psweep* is `true`, contains the spurious and reference waveforms.

*nil*  
Returns *nil* and an error message otherwise.

Examples
Each of following examples returns curves related to an ip3 measurement:

```plaintext
ipnVRICurves(v("/Pif" ?result "pss_fd") 9 8)

ipnVRICurves(v("/Pif" ?result "pss_fd") 9 8
  ?rport resultParam("rif:r" ?result "pss_td"))

ipnVRICurves(v("/Pif" ?result "pss_fd") 9 8
  ?iport i("/rif/PLUS" ?result "pss_fd") ?epoint -25)

ipnVRICurves(v("/Pif" ?result "pac") -21 -25
  ?rport resultParam("rif:r" ?result "pss_td"))
```
kf

\[ kf(\ o_{s11} \ o_{s12} \ o_{s21} \ o_{s22} )\Rightarrow o_{\text{waveform}}/\text{nil} \]

**Description**

Returns the stability factor in terms of the supplied parameters.

**Arguments**

- \( o_{s11} \) Waveform object representing \( s_{11} \).
- \( o_{s12} \) Waveform object representing \( s_{12} \).
- \( o_{s21} \) Waveform object representing \( s_{21} \).
- \( o_{s22} \) Waveform object representing \( s_{22} \).

**Value Returned**

- \( o_{\text{waveform}} \) Waveform object representing the stability factor.
- \( \text{nil} \) Returns \( \text{nil} \) if there is an error.

**Examples**

\[
s_{11} = \text{sp}(1 \ 1) \\
s_{12} = \text{sp}(1 \ 2) \\
s_{21} = \text{sp}(2 \ 1) \\
s_{22} = \text{sp}(2 \ 2) \\
\text{plot}(kf(s_{11} \ s_{12} \ s_{21} \ s_{22}))
\]
**In**

\[ \ln(\ {o\_waveform\ |\ n\_number}\ ) \Rightarrow o\_waveform/f\_number/nil \]

**Description**

Gets the base-e (natural) logarithm of a waveform or number.

**Arguments**

- **o\_waveform**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

- **n\_number**
  Number.

**Value Returned**

- **o\_waveform**
  Returns a waveform object representing the base-e (natural) logarithm of the input waveform if the input argument is a waveform object, or returns a family of waveforms if the input argument is a family of waveforms.

- **f\_number**
  Returns a number if the input argument is a number.

- **nil**
  Returns `nil` and an error message otherwise.

**Examples**

- \[ \ln(\ v(\ "/net9"\ )) \]
  Gets a waveform that is calculated as the natural logarithm of the input waveform.

- \[ \ln(\ \text{ymax}(v("/net9"))) \]
  Gets a waveform that is calculated as the natural logarithm of the following: \( \text{ymax}(v("/net9")) \).

- \[ \ln(100) \Rightarrow 4.60517 \]
  Gets the natural logarithm of 100.
log10

log10( {o_waveform | n_number} ) => o_waveform/n_number/nil

Description

Gets the base-10 logarithm of a waveform or a number.

Arguments

o_waveform  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)
n_number    Number.

Value Returned

o_waveform  Returns a waveform object if the input argument is a waveform object or returns a family of waveforms if the input argument is a family of waveforms.
n_number    Returns a number that is calculated as the base-10 logarithm of the input number.
nil          Returns nil and an error message otherwise.

Examples

log10( v( "/net9" ) )

Gets a waveform that is calculated as the base-10 logarithm of the input waveform.

log10( ymax( v( "/net9" ) ) )

Gets a waveform representing the base-10 logarithm of ymax(v("/net9")).

log10( 100 ) => 2.0

Gets the base-10 logarithm of 100, or 2.
**lsb**

```
lsb( o_s11 o_s12 o_s21 o_s22 g_level g_frequency ) => o_waveform/nil
```

**Description**

Computes the load stability circles.

The `g` datatype on `g_level` and `g_frequency` allows either the level or the frequency to be swept while the other remains fixed.

**Arguments**

- `o_s11` Waveform object representing s11.
- `o_s12` Waveform object representing s12.
- `o_s21` Waveform object representing s21.
- `o_s22` Waveform object representing s22.
- `g_level` Level in dB. It can be specified as a scalar or a vector. The level is swept, if it is specified as a vector. The `linRg` function can be called to generate a linear range. For example, `linRg( -30 30 5 )` is the same as `list(-30 -25 -20 -15 -10 -5 0 5 10 15 20 25 30)` and the `g_level` argument can be specified as either of the above. In that case, a load stability circle is calculated at each one of the 13 levels.
- `g_frequency` Frequency. It can be specified as a scalar or a linear range. If it is specified as a linear range, the frequency is swept. The linear range is specified as a list with three values: the start of the range, the end of the range, and the increment. For example, `list(100M 1G 100M)` specifies a linear range with the following values:

\[
\{ 100M, 200M, 300M, 400M, 500M, 600M, 700M, 800M, 900M, 1G \}
\]

In that case, a load stability circle is calculated at each one of the 10 frequencies.
Value Returned

\texttt{o\_waveform} \quad \text{Waveform object representing the load stability circles.}

\texttt{nil} \quad \text{Returns \texttt{nil} and an error message otherwise.}

Examples

\texttt{plot(lsb(s11 \ s12 \ s21 \ s22 \ list(800M \ 1G \ 100M)))}
Lshift

lshift( o_waveform n_delta ) => o_waveform/nil

Description

Shifts the waveform to the left by the delta value.

This command is the inverse of the rshift command.

Arguments

o_waveform

Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

n_delta

Value by which the waveform is to be shifted.

Value Returned

o_waveform

Returns a waveform object representing the input waveform shifted to the left. Returns a family of waveforms if the input argument is a family of waveforms.

nil

Returns nil and an error message otherwise.

Example

plot( lshift( v( "/net8" ) 30u ) )

Shifts the waveform representing the voltage of "/net8" to the left by 30u and plots the resulting waveform.
mag

\[ \text{mag( \{ o\_waveform | n\_number \} ) => o\_waveform/n\_number/nil} \]

**Description**

Gets the magnitude of a waveform or number.

**Arguments**

- **o\_waveform**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

- **n\_number**
  Number.

**Value Returned**

- **o\_waveform**
  Returns a waveform object if the input argument is a waveform object or returns a family of waveforms if the input argument is a family of waveforms.

- **n\_number**
  Returns a number if the input argument is a number.

- **nil**
  Returns `nil` and an error message otherwise.

**Examples**

```
\text{mag( v( "5" ) )}
```

Gets the magnitude of the waveform representing the voltage at net 5. You can also use the `vm` alias to perform the same command, as in `vm( "5" )`.

```
\text{mag( i( "VFB" ) )}
```

Gets the magnitude of the waveform representing current through the `VFB` component. You can also use the `im` alias to perform the same command, as in `im( "VFB" )`.

```
\text{mag( \text{-10} ) => 10}
```

Returns the magnitude of \text{-10}. 
nc

\texttt{nc( o\_NFmin o\_Gmin o\_RN g\_level g\_frequency ) \Rightarrow o\_waveform/nil}

\textbf{Description}

Computes the noise circles.

\textbf{Arguments}

\begin{itemize}
  \item \texttt{o\_NFmin} Waveform object representing the minimum noise factor.
  \item \texttt{o\_Gmin} Waveform object representing the optimum noise reflection.
  \item \texttt{o\_RN} Waveform object representing the equivalent noise resistance.
  \item \texttt{g\_level} Level in dB. It can be specified as a scalar or a vector. The level is swept, if it is specified as a vector. The \texttt{linRg} function can be called to generate a linear range. For example, \texttt{linRg( -30 30 5 )} is the same as \texttt{list(-30 -25 -20 -15 -10 -5 0 5 10 15 20 25 30)} and the \texttt{g\_level} argument can be specified as either of the above. In that case, a noise circle is calculated at each one of the 13 levels.
  \item \texttt{g\_frequency} Frequency. It can be specified as a scalar or a linear range. The frequency is swept if it is specified as a linear range. The linear range is specified as a list with three values: the start of the range, the end of the range, and the increment. For example, \texttt{list(100M 1G 100M)} specifies a linear range with the following values:

  \{ 100M, 200M, 300M, 400M, 500M, 600M, 700M, 800M, 900M, 1G \}

  In that case, a noise circle is calculated at each one of the 10 frequencies.
\end{itemize}

\textbf{Value Returned}

\begin{itemize}
  \item \texttt{o\_waveform} Waveform object representing the noise circles.
nil

Returns nil and an error message otherwise.

**Examples**

Gopt = getData("Gopt")
Bopt = getData("Bopt")
Zref = zref(1 ?result "sp")
Gmin = gmin(Gopt Bopt Zref)
NFmin = getData("NFmin")
RN = getData("RN")
NC = nc(NFmin Gmin RN 10 list(100M 1G 100M))
displayMode("smith")
smithType("impedance")
plot(NC)
overshoot

overshoot( o_waveform n_initVal g_initType n_finalVal 
  g_finalType ) => o_waveform/\text{n\_value}/\text{nil}

Description

Computes the percentage by which an expression overshoots a step going from the initial value to the final value you enter.

This command returns the overshoot of $o\_\text{waveform}$ as a percentage of the difference between the initial value and the final value.

$$Overshoot = \frac{M - F}{F - I}$$

Arguments

- **o_waveform**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)

- **n_initVal**
  Initial X value at which to start the computation.

- **g_initType**
  Specifies how \text{initVal} functions.
  Valid values: a non-nil value specifies that the initial value is taken to be the value of the waveform, interpolated at \text{initVal}, and the waveform is clipped from below, as follows:

  $$o\_\text{waveform} = \text{clip}( o\_\text{waveform} \ \text{initVal} \ \text{nil} )$$
nil specifies that $initVal$ is defined by the X value entered. (The command gets the Y value for the specified X value and uses that value for $initVal$.)

$n\_finalVal$ Final value at which to end the computation.

g$\_finalType$ Specifies how $finalVal$ functions. Valid values: a non-nil value specifies that the final value is taken to be the value of the waveform, interpolated at $finalVal$, and the waveform is clipped from above, as follows:

\[
o\_waveform = \text{clip}(\ o\_waveform\ \text{nil}\ finalVal)\]

nil specifies that $finalVal$ is defined by the X value entered. (The command gets the Y value for the specified X value and uses that value for $finalVal$.)

**Value Returned**

$o\_waveform$ Returns a waveform (or family of waveforms) representing the amount of overshoot in comparison to the whole signal if the input argument is a family of waveforms.

$n\_value$ Returns a value for the amount of overshoot in comparison to the whole signal if the input is a single waveform.

nil Returns nil and an error message otherwise.

**Example**

overshoot( v( "/net8" ) 7n t 3.99u t )

Returns the value of the overshoot for the waveform representing the voltage of "/net8".
peakToPeak

peakToPeak( o_waveform ) => o_waveform/n_value/nil

Description

Returns the difference between the maximum and minimum values of a waveform.

Arguments

o_waveform  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

Value Returned

o_waveform  Returns a waveform or a family of waveforms if the input argument is a family of waveforms.

n_value  Returns the difference between the maximum and minimum values of a waveform if the input argument is a single waveform.

nil  Returns nil and an error message otherwise.

Example

peakToPeak( v( "/net2" ) )

Returns the difference between the maximum and minimum values of the waveform representing the voltage of the "/net2" net.
phase

phase( {o_waveform | n_number} ) => o_waveform/n_number/nil

Description

Gets the phase of the waveform or number.

The phase command is similar to the phaseDegUnwrapped command and returns the unwrapped phase in degrees.

Arguments

<table>
<thead>
<tr>
<th>o_waveform</th>
<th>Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_number</td>
<td>Number.</td>
</tr>
</tbody>
</table>

Value Returned

<table>
<thead>
<tr>
<th>o_waveform</th>
<th>Returns a waveform object if the input argument is a waveform object or returns a family of waveforms if the input argument is a family of waveforms.</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_number</td>
<td>Returns a number if the input argument is a number.</td>
</tr>
<tr>
<td>nil</td>
<td>Returns nil and an error message otherwise.</td>
</tr>
</tbody>
</table>

Examples

phase( v( "5" ) )

Gets the phase of the waveform representing the voltage at net 5. You can also use the vp alias to perform the same command, as in vp( "5" ).

phase( i( "VFB" ) )

Gets the phase of the waveform representing the current through the VFB component. You can also use the ip alias to perform the same command, as in ip( "VFB" ).

phase( -2.0 ) => 180.0
Gets the phase of $-2$. 
phaseDeg

phaseDeg( {o_waveform | n_number} ) => o_waveform/n_number/nil

Description
Calculates the wrapped phase in degrees of a waveform and returns a waveform.

Arguments

\[ o\_waveform \]
Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \texttt{drwave:XXXXX}.)

\[ n\_number \]
Number.

Value Returned

\[ o\_waveform \]
Returns a waveform object representing the wrapped phase in degrees of the input waveform. Returns a family of waveforms if the input argument is a family of waveforms.

\[ n\_number \]
Returns a number if the input argument is a number.

\[ \text{nil} \]
Returns \text{nil} and an error message otherwise.

Example

phaseDeg( v( "vout" ) )

Takes the input waveform, representing the voltage of the \texttt{"vout"} net, and returns the waveform object representing the wrapped phase in degrees.
phaseDegUnwrapped

phaseDegUnwrapped( {o_waveform | n_number} ) =>
  o_waveform/n_number/nil

Description
Calculates the unwrapped phase in degrees of a waveform and returns a waveform.

Arguments

  o_waveform  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

  n_number   Number.

Value Returned

  o_waveform  Returns a waveform object representing the unwrapped phase in degrees of the input waveform. Returns a family of waveforms if the input argument is a family of waveforms.

  n_number   Returns a number if the input argument is a number.

  nil        Returns nil and an error message otherwise.

Example

phaseDegUnwrapped( v( "vout" ) )

Takes the input waveform, representing the voltage of the "vout" net, and returns the waveform object representing the unwrapped phase in degrees.
phaseMargin

\[
\text{phaseMargin( o\_waveform )} \Rightarrow \text{o\_waveform/n\_value/nil}
\]

**Description**

Computes the phase margin of the loop gain of an amplifier.

You supply a waveform representing the loop gain of interest over a sufficiently large frequency range.

\[
\text{phaseMargin( gain )} = 180 + \text{phase( value( gain f0 ) )}
\]

The phase margin is calculated as the difference between the phase of the gain in degrees at f0 and at -180 degrees. The frequency f0 is the lowest frequency where the gain is 1. For stability, the phase margin must be positive.

**Arguments**

- **o\_waveform**
  
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)

![Diagram of phase margin and gain crossover frequency](image-url)
Value Returned

\textit{o\_waveform}\hspace{5mm} Returns a waveform representing the phase margin of the loop gain of an amplifier for a family of waveforms if the input argument is a family of waveforms.

\textit{n\_value}\hspace{5mm} Returns the value (in degrees) equivalent to the phase margin of the input waveform.

\textit{nil}\hspace{5mm} Returns \texttt{nil} and an error message otherwise.

Example

\texttt{phaseMargin( v( "\textquote{/OUT}\textquote" ) )}

Returns the phase margin for the waveform representing the voltage of the 
"\textquote{/OUT}\textquote" net.
phaseRad

\[ \text{phaseRad( } \{ \text{o\_waveform} \mid \text{n\_number} \} \) } \Rightarrow \text{o\_waveform/n\_number/nil} \]

Description

Calculates the wrapped (discontinuous) phase in radians of a waveform.

Arguments

- \text{o\_waveform} Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)
- \text{n\_number} Number.

Value Returned

- \text{o\_waveform} Returns a waveform representing a discontinuous value (in radians) for the phase of the input waveform. Returns a family of waveforms if the input argument is a family of waveforms.
- \text{n\_number} Returns a number when the input argument is a number.
- \text{nil} Returns \text{nil} and an error message otherwise.

Example

\[ \text{plot( phaseRad( v( "/OUT" ) ) )} \]

Returns the wrapped phase of the waveform representing the voltage of the "/OUT" net.
phaseRadUnwrapped

\[ \text{phaseRadUnwrapped}( \text{o\_waveform} ) \Rightarrow \text{o\_waveform/nil} \]

**Description**

Calculates the unwrapped (continuous) phase in radians of a waveform and returns a waveform.

**Arguments**

\( \text{o\_waveform} \)

Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)

**Value Returned**

\( \text{o\_waveform} \)

Returns a waveform representing the unwrapped (continuous) value for the phase of the input waveform in radians. Returns a family of waveforms if the input argument is a family of waveforms.

\( \text{nil} \)

Returns \text{nil} and an error message otherwise.

**Example**

\[ \text{plot}( \text{phaseRadUnwrapped}( \text{v( "/OUT" ))} ) \]

Returns the unwrapped phase of the waveform representing the voltage of the "/OUT" net.
pow

pow( {o_waveformBase | n_numberBas} {o_waveformExpn | n_numberExpn} ) => o_result/n_result/nil

Description
Takes the exponent of a given waveform or number.

Arguments

o_waveformBase
Waveform object to be used as the base for the expression.

o_waveformExpn
Waveform object to be used as the exponent for the expression.

n_numberBase
Number to be used as the base for the expression.

n_numberExpn
Number to used as the exponent for the expression.

Value Returned

o_waveform
Returns a family of waveforms if one of the input arguments is a family of waveforms or returns a waveform if one of the input arguments is a waveform (and none is a family).

n_result
Returns a number if both the input arguments are numbers.

nil
Returns nil and an error message otherwise.

Examples

pow( average( v( "/net9" ) ) 0.5 )

Gets the square root of the average value of the voltage at "/net9".

pow( 2 3 ) => 8

Gets the value of 2 to the third power, or 8.

pow( -2 2 ) => 4
Gets the value of -2 to the second power.
\[ \text{pow}( 2.5 \ -1.2 ) \Rightarrow 0.3330213 \]

Gets the value of 2.5 to the power of -1.2.
psd

psd( o_waveform f_timeStart f_timeEnd x_num ?windowName t_windowName ?smooth x_smooth ?cohGain f_cohGain ?windowsize x_windowsize ?detrending t_detrending) => o_waveformReal/nil

Description

Returns an estimate for the power spectral density of o_waveform. If x_windowsize is not a power of 2, it is forced to the next higher power of 2. If x_num is less than x_windowsize, x_num is forced to x_windowsize.

Arguments

- **o_waveform**: Time domain waveform object with units of volts or amps.
- **f_timeStart**: Starting time for the spectral analysis interval. Use this parameter and f_timeEnd to exclude part of the interval. For example, you might set these values to discard initial transient data.
- **f_timeEnd**: Ending time for the spectral analysis interval.
- **x_num**: The number of time domain points to use. The maximum frequency in the Fourier analysis is proportional to x_num and inversely proportional to the difference between f_timeStart and f_timeEnd. Default value: 512
- **x_smooth**: The Kaiser window smoothing parameter. The 0 value requests no smoothing. Valid values: 0 <= x_smooth <= 15. Default value: 1
**Predefined Functions and Waveform (Calculator) Functions**

**f_cohGain**
A scaling parameter. A non-zero value scales the power spectral density by \(1/(f_{cohGain})\).
Valid values: \(0 < f_{cohGain} < 1\) (You can use 1 if you do not want the scaling parameter to be used)
Default value: 1

**x_windowsize**
The number of frequency domain points to use in the Fourier analysis. A larger window size results in an expectation operation over fewer samples, which leads to larger variations in the power spectral density. A small window size can smear out sharp steps in the power spectral density that might really be present.
Default value: 256

**t_detrending**
The detrending mode to use.
Valid values: 'mean, 'linear, 'none
Default value: 'none

The `psd` function works by applying a moving windowed FFT to time-series data. If there is a deterministic trend to the underlying data, you might want to remove the trend before performing the spectral analysis. For example, consider analyzing phase noise in a VCO model. Without the noise, the phase increases more or less linearly with time, so it is appropriate to set the detrending mode to 'linear. To subtract an average value, set the detrending mode to 'mean. Where the spectrum of raw data is desired, set the detrending mode to 'none.

**Value Returned**

**o_waveformReal**
The power spectral density waveform returned when the command is successful.

nil
Returns nil when the command fails.

**Example**

```plaintext
psd(VT("/net32" "/hm/test_bench/spectre/schematic"), 0, 16m, 12000, 
  ?windowName 'Hanning, ?smooth 1, ?windowSize 256, 
  ?detrending 'None, ?cohGain 1)
```
Consider applying this command to one of the waveforms in the following illustration.
The result is the following spectrum, which is displayed with a logarithmic vertical scale.
psdbb

psdbb( o_waveform1 o_waveform2 f_timeStart f_timeEnd x_num
  ?windowName t_windowName ?smooth x_smooth ?cohGain f_cohGain
  ?windowsize x_windowsize ?detrending t_detrending) =>
  o_waveformReal/nil

Description

Returns an estimate for the power spectral density of \( o\_waveform1 + j \times o\_waveform2 \).
If \( x\_windowsize \) is not a power of 2, it is forced to the next higher power of 2.
If \( x\_num \) is less than \( x\_windowsize \), \( x\_num \) is forced to \( x\_windowsize \).

Arguments

\( o\_waveform1 \)
Time domain waveform object with units of volts or amps.

\( o\_waveform2 \)
Time domain waveform object with units of volts or amps.

\( f\_timeStart \)
Starting time for the spectral analysis interval. Use this parameter and \( f\_timeEnd \) to exclude part of the interval. For example, you might set these values to discard initial transient data.

\( f\_timeEnd \)
Ending time for the spectral analysis interval.

\( x\_num \)
The number of time domain points to use. The maximum frequency in the Fourier analysis is proportional to \( x\_num \) and inversely proportional to the difference between \( f\_timeStart \) and \( f\_timeEnd \).

\( t\_windowName \)
The window to be used for applying the moving window FFT.
Default value: ‘Hanning’

\( x\_smooth \)
The Kaiser window smoothing parameter. 0 requests no smoothing.
Valid values: 0 <= \(x_{\text{smooth}}\) <= 15.
Default value: 1

\(f_{\text{cohGain}}\)
A scaling parameter. A non-zero value scales the power spectral density by \(1/(f_{\text{cohGain}})\).
Valid values: 0 < \(f_{\text{cohGain}}\) < 1 (You can use 1 if you do not want the scaling parameter to be used)
Default value: 1

\(x_{\text{windowsize}}\)
The number of frequency domain points to use in the Fourier analysis. A larger window size results in an expectation operation over fewer samples, which leads to larger variations in the power spectral density. A small window size can smear out sharp steps in the power spectral density that might really be present.

\(t_{\text{detrending}}\)
The detrending mode to use.
Valid values: ‘mean, ‘linear, ‘none
Default value: ‘none

The \text{psd} function works by applying a moving windowed FFT to time-series data. If there is a deterministic trend to the underlying data, you might want to remove the trend before performing the spectral analysis. For example, consider analyzing phase noise in a VCO model. Without the noise, the phase increases more or less linearly with time, so it is appropriate to set the detrending mode to ‘linear. To subtract an average value, set the detrending mode to ‘mean. Where the spectrum of raw data is desired, set the detrending mode to ‘none.

Value Returned

\(o_{\text{waveformReal}}\)
The power spectral density waveform returned when the command is successful.

\text{nil}\nReturns \text{nil} when the command fails.

Example

\text{psdbb}(\text{VT}("/net32" "/hm/test_bench/spectre/schematic"),
          \text{VT}("/net11" "/hm/test_bench/spectre/schematic"), 0, 16m, 12000,
Consider applying this command to both of the waveforms in the following illustration.
The result is the following spectrum, which is displayed with a logarithmic vertical scale.

![Spectrum Graph](image)
real

real( {o_waveform | n_input} ) => o_waveformReal/n_numberReal/nil

Description

Returns the real part of a waveform representing a complex number, or returns the real part of a complex number.

Arguments

- **o_waveform**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

- **n_input**
  Complex number.

Value Returned

- **o_waveformReal**
  Returns a waveform when the input argument is a waveform.

- **n_numberReal**
  Returns a number when the input argument is a number.

- **nil**
  Returns `nil` and an error message otherwise.

Example

real( v( "/net8" ) )

Returns a waveform representing the real part of the voltage of "/net8". You also can use the `vr` alias to perform the same command, as in `vr( "net8" )`.

x=complex( -1 -2 ) => complex(-1, -2)
real( x ) => -1.0

Creates a variable `x` representing a complex number, and returns the real portion of that complex number.
riseTime

\[
\text{riseTime}( \text{o\_waveform} \; n\text{\_initVal} \; g\text{\_initType} \; n\text{\_finalVal} \; g\text{\_finalType} \\
\; n\text{\_theta1} \; n\text{\_theta2} ) \Rightarrow \text{o\_waveform}/n\text{\_value}/\text{nil}
\]

**Description**

Returns the rise time measured between \( \text{theta1} \) (percent low) to \( \text{theta2} \) (percent high) of the difference between the initial value and the final value.

The \text{riseTime} function can also be used to compute the fall time if \text{initVal} is higher than \text{finalVal}.

**Arguments**

- **\text{o\_waveform}**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)

- **\text{n\_initVal}**
  Initial value at which to start the computation.

- **\text{g\_initType}**
  Specifies how \text{n\_initVal} functions. Valid values: a non-nil value specifies that the initial value is taken to be the value of the waveform, interpolated at \text{n\_initVal}, and the waveform is clipped from below as follows:

\[
\text{o\_waveform} = \text{clip}( \text{o\_waveform} \; \text{g\_initVal} \; \text{nil} )
\]
nil specifies that \textit{n\_initVal} is defined by the X value entered. (The command gets the Y value for the specified X value and uses that value for \textit{n\_initVal}.)

\textbf{\textit{n\_finalVal}}

Final value at which to end the computation.

\textbf{\textit{g\_finalType}}

Specifies how the \textit{n\_finalVal} argument functions. Valid values: a non-nil value specifies that the final value is taken to be the value of the waveform, interpolated at \textit{n\_finalVal}, and the waveform is clipped from above, as follows:

\[
o\_\text{waveform} = \text{clip}(o\_\text{waveform} \text{ nil } n\_\text{finalVal})
\]

\text{nil} specifies that the \textit{n\_finalVal} argument is defined by the X value entered. (The command gets the Y value for the specified X value and uses that value for \textit{n\_finalVal}.)

\textbf{\textit{n\_theta1}}

Percent low.

\textbf{\textit{n\_theta2}}

Percent high.

\textbf{Value Returned}

\textbf{\textit{o\_waveform}}

Returns a waveform representing the rise time for a family of waveforms if the input argument is a family of waveforms.

\textbf{\textit{n\_value}}

Returns a value for the rise time if the input is a single waveform.

\textbf{\textit{nil}}

Returns \textit{nil} and an error message otherwise.

\textbf{Examples}

\texttt{riseTime( v( "/net8" ) 0 t 2 t 10 90 )}

Computes the rise time for the waveform representing the voltage of "/net8" from 0 to 2.

For the next example, assume that \texttt{v} is the following sinusoidal waveform:

\texttt{sin( 2 * pi * time)}

\texttt{riseTime( v 0.25 t 0.5 t 10 90 )}

Computes the fall time of the first falling edge from 1 to 0.
**rms**

```plaintext
rms( o_waveform ) => o_waveform/n_value/nil
```

**Description**

Returns the root-mean-square value of a waveform.

**Arguments**

- **o_waveform**
  
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

**Value Returned**

- **o_waveform**
  
  Returns a waveform representing the root-mean-square value for a family of waveforms if the input argument is a family of waveforms.

- **n_value**
  
  Returns a value for the root-mean-square value for the specified waveform if the input is a single waveform.

- **nil**
  
  Returns `nil` and an error message otherwise.

**Example**

```plaintext
rms( v( "/out" ) )
```

Returns the root-mean-square value of the waveform representing the voltage of the `/out` net.
rmsNoise

rmsNoise( n_from n_to ) => o_waveform/n_value/nil

Description

Computes the integrated root-mean-square noise over the specified bandwidth.

Arguments

n_from       Frequency in hertz that specifies the minimum value for the bandwidth.

n_to         Frequency in hertz that specifies the maximum value for the bandwidth.

Value Returned

o_waveform   Returns a waveform (or a family of waveforms) representing the integrated root-mean-square noise if the data being analyzed is parametric.

n_value      Returns a value for the integrated root-mean-square noise if the data being analyzed is from a single simulation run.

nil          Returns nil and an error message otherwise.

Example

rmsNoise( 100 100M ) => 250e-6

Computes the integrated root-mean-square noise from 100 to 100M.
root

root( o_waveform n_rootVal x_n ) =>
  o_waveform/n_value/l_value/nil

Description

Returns the nth X value at which the Y value equals the specified Y value (rootVal).

Arguments

o_waveform  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

n_rootVal  Y value of interest.

x_n  Number that specifies which X value to return. If n equals 1, the first X value that crosses over the Y rootVal is returned. If n equals 2, the second X value that crosses over the Y rootVal is returned, and so on. If you specify a negative integer for n, the X values that cross the rootVal are counted from right to left (from maximum to minimum). If you specify n as 0, the list of root values is returned.

Value Returned

o_waveform  Returns a waveform if the input argument is a family of waveforms.

n_value  Returns an X value when the input argument is a single waveform.

l_value  Returns a list of all the root values when n is 0.

nil  Returns nil and an error message otherwise.

Example

root( v( "vout" ), 1.0, 4 )
Returns the X value for the point at which the waveform curve crosses the 1.0 Y value for the fourth time.
rshift

\[
\text{rshift}( \text{o\_waveform n\_delta} ) \Rightarrow \text{o\_waveform/nil}
\]

Description

Shifts the waveform to the right by the \( n\_delta \) value.

This command is the inverse of the \text{lshift} command.

Arguments

- \text{o\_waveform} Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \text{drwave:XXXXX}.)
- \text{n\_delta} Value by which the waveform is to be shifted.

Value Returned

- \text{o\_waveform} Returns a waveform object. Returns a family of waveforms if the input argument is a family of waveforms.
- \text{nil} Returns \text{nil} and an error message otherwise.

Example

\[
\text{rshift}( \text{v( "vout" ) ) 10n} )
\]

Shifts the waveform representing the voltage through the "vout" net to the right by 10n.
**sample**

sample( o_waveform n_from n_to t_type n_by ) =>
   o_waveform/n_number/nil

**Description**

Samples a waveform at the specified interval.

You can use this function to reduce the time it takes to plot waveforms that have many data points. If you sample a waveform beyond its range, you get the final value of the waveform. You can use this function to demodulate a signal. Consider an AM modulated sine wave. Assume the carrier frequency is 1 GHz, and the modulation frequency is 1 MHz. If the waveform is sampled every 1 ns, the resulting signal is cleanly demodulated (the 1 GHz carrier is completely eliminated by the sampling).

**Arguments**

- **o_waveform**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: *drwave:XXXXX*.)

- **n_from**
  Starting value for the sampling.

- **n_to**
  Ending value for the sampling.

- **t_type**
  Type of the sampling.
  **Valid values:** "linear" or "log"

- **n_by**
  Interval at which to sample.

**Value Returned**

- **o_waveform**
  Returns a waveform representing the sampling you specified.

- **n_number**
  Returns a number if the output contains only one point.

- **nil**
  Returns *nil* and an error message otherwise.
Examples

sample( v( "vout" ) 0 50n "linear" 0.1n )

Takes a linear sample of the waveform representing the voltage of the "vout" net.

sample( v( "vout" ) 0 100m "log" 10 )

Takes a logarithmic sample of the waveform representing the voltage of the "vout" net.
settlingTime

settlingTime( o_waveform n_initVal g_initType n_finalVal
             g_finalType n_theta ) => o_waveform/n_value/nil

Description

Computes the time required for a waveform to settle within the specified percent of step (theta) of the difference between the initial value and final value.

Arguments

- o_waveform: Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)
- n_initVal: Initial value at which to start the computation.
- g_initType: Specifies whether the values entered are X values or Y values. Valid values: t specifies that initVal is defined by the X value...
entered; nil specifies that initVal is defined by the Y value entered

\textit{n_finalVal} Final value at which to start the computation.

\textit{g_finalType} Specifies whether the values entered are X values or Y values. Valid values: \textit{t} specifies that finalVal is defined by the X value entered; nil specifies that finalVal is defined by the Y value entered

\textit{n_theta} Percent of the total step.

\textbf{Value Returned}

\textit{o_waveform} Returns a waveform representing the settling time for a family of waveforms if the input argument is a family of waveforms.

\textit{n_value} Returns a value for the settling time for the specified waveform if the input is a single waveform.

\textit{nil} Returns \textit{nil} and an error message otherwise.

\textbf{Example}

\texttt{settlingTime( v("/out" ) 0 t 2 t 90 )}

Computes the time required for the waveform representing the voltage of the "/out" net to settle within 90 percent of the step from 0 to 2.
slewRate

slewRate( o_waveform n_initVal g_initType n_finalVal g_finalType n_theta1 n_theta2 ) => o_waveform/n_value/nil

Description

Computes the average rate at which an expression changes from \( \theta_1 \) (percent low) to \( \theta_2 \) (percent high) of the difference between the initial value and final value.

\[
slewRate = \frac{\Delta Y}{\Delta X}
\]

Arguments

- **o_waveform**  
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: \texttt{drwave:XXXXX}.)

- **n_initVal**  
  Initial X-axis value at which to start the computation.

- **g_initType**  
  Specifies whether the values entered are X values or Y values. Valid values: \texttt{t} specifies that \texttt{initVal} is defined by the X value entered; \texttt{nil} specifies that \texttt{initVal} is defined by the Y value entered

- **n_finalVal**  
  Final value at which to end the computation.

- **g_finalType**  
  Specifies whether the values entered are X values or Y values. Valid values: \texttt{t} specifies that \texttt{finalVal} is defined by the X value entered; \texttt{nil} specifies that \texttt{finalVal} is defined by the Y value entered

- **n_theta1**  
  Percent low (percentage of the total step).
**n_theta2**

Percent high (percentage of the total step).

**Value Returned**

**o_waveform**

Returns a waveform representing the slew rate for a family of waveforms if the input argument is a family of waveforms.

**n_value**

Returns a value for the slew rate for the specified waveform if the input is a single waveform.

**nil**

Returns nil and an error message otherwise.

**Example**

```
slewRate( v( "vout" ) 10n t 30n t 10 90 )
```

Computes the slew rate for the waveform representing the voltage of the "vout" net from 10n to 30n.

```
slewRate( v( "vout" ) 0 nil 10 nil 5 95 )
```

Computes the slew rate for the waveform representing the voltage of the "vout" net from 0 to 10. In this example, the initial value and final value are entered as Y values.
spectralPower

spectralPower( o_current o_voltage ) => o_power/nil

Description

Returns the spectral power given the spectral current and voltage.

To obtain a list of the harmonic frequencies, use harmonicList.

Arguments

o_current  Waveform representing the current. The current can be obtained by calling the i data access function for the desired terminal.

o_voltage  Waveform representing the voltage. The voltage can be obtained by calling the v data access function for the desired net. To obtain meaningful results, the terminal used to obtain the current must be a member of the net used to obtain the voltage.

Value Returned

o_power  Waveform representing the power of the net.

nil  Returns nil if there is an error.

Example

plot(db10(spectralPower(i("/PORT0/PLUS") v("/net28")).))

Plots power of the output "/net28". "/PORT0/PLUS" is a member of "/net28".
ssb

ssb( o_s11 o_s12 o_s21 o_s22 g_level g_frequency ) => o_waveform/nil

Description

Computes the source stability circles.

The $g$ data type on $g\_level$ and $g\_frequency$ allows either the level or the frequency to be swept while the other remains fixed.

Arguments

- **o_s11**: Waveform object representing s11.
- **o_s12**: Waveform object representing s12.
- **o_s21**: Waveform object representing s21.
- **o_s22**: Waveform object representing s22.
- **g_level**: Level in dB. It can be specified as a scalar or a vector. The level is swept, if it is specified as a vector. The `linRg` function can be called to generate a linear range. For example, `linRg(-30 30 5)` is the same as `list(-30 -25 -20 -15 -10 -5 0 5 10 15 20 25 30)` and the `g_level` argument can be specified as either of the above. In that case, a source stability circle is calculated at each one of the 13 levels.
- **g_frequency**: Frequency. It can be specified as a scalar or a linear range. The frequency is swept if it is specified as a linear range. The linear range is specified as a list with three values: the start of the range, the end of the range, and the increment. For example, `list(100M 1G 100M)` specifies a linear range with the following values:

  ```
  { 100M, 200M, 300M, 400M, 500M, 600M, 700M, 800M, 900M, 1G }
  ```

  In that case, a source stability circle is calculated at each one of the 10 frequencies.
Value Returned

\texttt{o\_waveform} \quad \text{Waveform object representing the source stability circles.}

\texttt{nil} \quad \text{Returns \texttt{nil} and an error message otherwise.}

Example

\texttt{plot(ssb(s11 \ s12 \ s21 \ s22 \ list(800M \ 1G \ 100M)))}
**tangent**

\[
\text{tangent( o\_waveform [ ?x \_x ] [ ?y \_y ] [ ?slope \_slope ] )} \Rightarrow \\
\text{o\_waveform/nil}
\]

**Description**

Returns the tangent to a waveform through the point \((n\_x, n\_y)\) with the given slope.

**Arguments**

- **o\_waveform**
  Waveform object representing the wave.
- **n\_x**
  X coordinate of the point. The default value is the X coordinate of the first point on the wave.
- **n\_y**
  Y coordinate of the point. The default value is the Y coordinate at the given or default X coordinate.
- **n\_slope**
  Slope of the line.
  Default value: 1.0

**Value Returned**

- **o\_waveform**
  Wave object representing the line.
- **nil**
  Returns nil if there is an error.

**Example**

\[
\text{refLine} = \text{tangent(refWave ?x -25 ?slope 1.0)}
\]
**thd**

\[ \text{thd}( \text{o\_waveform n\_from n\_to x\_num} ) = \rightarrow \text{o\_waveform/n\_thdValue/nil} \]

**Description**

Computes the absolute value of the total harmonic distortion of the input waveform.

The accuracy of this function depends on how you set certain simulator options and analysis parameters. The computation uses the `dft` command and takes the same arguments.

Assume the `dft` function returns the complex components f0, f1… (where f0 is the DC component, f1 is the component at frequency 1/(to - from)) then the total harmonic distortion is computed as

\[
\frac{100 \sqrt{|f_2|^2 + |f_3|^2 + \cdots}}{|f_1|}
\]

**Arguments**

- **o\_waveform**
  - Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

- **n\_from**
  - Starting value for the computation.

- **n\_to**
  - Ending value for the computation.

- **x\_num**
  - Number of timepoints. If `x\_num` is not a power of 2, it is forced to be the next higher power of 2.

**Value Returned**

- **o\_waveform**
  - Returns a waveform representing the absolute value of the total harmonic distortion if the input argument is a family of waveforms.

- **n\_thdValue**
  - Returns the absolute value of the total harmonic distortion of the input waveform.
nil

Returns nil and an error message otherwise.

Example

```
plot( thd( v( "/net8" ) 10u 20m 64 "rectangular" ) )
```

Computes the absolute value of the discrete Fourier transform, fast Fourier transform, of the waveform representing the voltage of "/net8". The computation is done from 10u to 20m with 64 timepoints. The resulting waveform is plotted.
**value**

\[
\text{value( o\_waveform } [s\_name] \text{ g\_value ) } \Rightarrow \text{ o\_waveform/g\_value/nil}
\]

**Description**

Returns the Y value of a waveform for a given X value.

**Arguments**

- **o\_waveform**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

- **s\_name**
  The name of the innermost or outermost sweep variable. If the sweep variable name is not supplied, the innermost sweep variable is used.

- **g\_value**
  Value (X value) at which to provide the Y value. If a string has been defined for a value or set of values, the string may be used instead of the value.

**Value Returned**

- **o\_waveform**
  Returns a waveform or a family of waveforms if the input argument is a family of waveforms.

- **g\_value**
  Returns the Y value if the input argument is a single waveform.

  **Note:** For parametric sweeps, the value might be a waveform that can be printed with the `ocnPrint` command.

- **nil**
  Returns `nil` and an error message if the value cannot be printed.

**Examples**

\[
\text{value( v( "/net18" ) 4.428e-05 )}
\]

Prints the value of "/net18" at `time=4.428e-05`. This is a parametric sweep of temperature over time.
value( v("/OUT")'TEMPDC 20.0 )

Returns \texttt{drwave:XXXXX}, indicating that the result is a waveform.

\texttt{print( value(v("/OUT")'TEMPDC 20.0) )}

Prints the value of \texttt{v("/OUT")} at every time point for \texttt{TEMPDC=20}.
**xmax**

```
xmax( o_waveform  x_numberOfPeaks  ) =>
o_waveform/g_value/l_value/nil
```

**Description**

Computes the value of the independent variable (X) at which the Y value attains its maximum value.

**Arguments**

- `o_waveform`: Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

- `x_numberOfPeaks`: Specifies the nth X value corresponding to the maximum Y value. For example, if `x_numberOfPeaks` is 3, the X value corresponding to the third maximum Y value is returned. If you specify a negative integer for `x_numberOfPeaks`, the X values are counted from right to left (from maximum to minimum). If `x_numberOfPeaks` is 0, `xmax` returns a list of X locations.

**Value Returned**

- `o_waveform`: Returns a waveform (or a family of waveforms) if the input argument is a family of waveforms.

- `g_value`: Returns the X value corresponding to the peak specified with `x_numberOfPeaks` if the input argument is a single waveform.

- `l_value`: Returns a list of X locations when `x_numberOfPeaks` is 0 and the input argument is a single waveform.

- `nil`: Returns `nil` and an error message otherwise.

**Examples**

```
xmax( v( "/net9" ) 1 )
```
Gets the time value (X-axis value) at which the voltage of "/net9" attains its first peak value.

\[ \text{xmax} \left( \text{v} \left( \text{"/net9"} \right), 0 \right) \]

Gets the list of time values (X-axis values) at which the voltage of "/net9" attains each of its peak values.
**xmin**

 xmin( o_waveform x_numberOfValleys ) =>
  o_waveform/g_value/l_value/nil

**Description**

Computes the value of the independent variable (X) at which the Y value attains its minimum value.

**Arguments**

- **o_waveform**
  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

- **x_numberOfValleys**
  Specifies the n-th X value corresponding to the minimum Y value. For example, if x_numberOfValleys is 3, the X value corresponding to the third minimum Y value is returned. If you specify a negative integer for x_numberOfValleys, the X-values are counted from right to left (from maximum to minimum). If x_numberOfValleys is 0, xmin returns a list of X locations.

**Value Returned**

- **o_waveform**
  Returns a waveform (or a family of waveforms) if the input argument is a family of waveforms.

- **g_value**
  Returns the X value corresponding to the valley specified with x_numberOfValleys if the input argument is a single waveform.

- **l_value**
  Returns a list of X locations when x_numberOfValleys is 0 and the input argument is a single waveform.

- **nil**
  Returns nil and an error message otherwise.
Examples

\texttt{xmin( v( "/net9" ) 1 )}

Gets the time value (X axis) at which the voltage of "/net9" has its first low point or valley.

\texttt{xmin( v( "/net9" ) 0 )}

Gets the list of time values (X axis) at which the voltage of "/net9" has low points or valleys.
xval

xval( o_waveform ) => o_waveform/nil

Description

Returns a waveform whose X vector and Y vector are equal to the input waveform’s X vector.

Arguments

o_waveform  Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

Value Returned

o_waveform  Returns a waveform if the input argument is a single waveform. Returns a family of waveforms if the input argument is a family of waveforms.

nil  Returns nil and an error message otherwise.

Example

xval( v( "/net8" ))

Returns a waveform in which the X vector for the voltage of "/net8" is also used for the Y vector.
ymax

ymax( o_waveform ) => n_max/o_waveformMax/nil

Description
Computes the maximum value of the waveform's Y vector.

A waveform consists of an independent-variable X vector and a corresponding Y vector.

Arguments

o_waveform Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: drwave:XXXXX.)

Value Returned

n_max Returns a number representing the maximum value of Y if the input argument is a single waveform.

o_waveformMax Returns a waveform (or family of waveforms) representing the maximum value of Y if the input argument is a family of waveforms.

nil Returns nil and an error message otherwise.

Example

ymax( v( "/net9" ) )

Gets the maximum voltage (Y value) of "/net9".
**ymin**

`ymin( o_waveform ) => n_min/o_waveformMin/nil`

**Description**
Computes the minimum value of a waveform’s Y vector.

(A waveform consists of an independent-variable X vector and a corresponding Y vector.)

**Arguments**

`o_waveform` Waveform object representing simulation results that can be displayed as a series of points on a grid. (A waveform object identifier looks like this: `drwave:XXXXX`.)

**Value Returned**

`n_min` Returns a number representing the minimum value of Y if the input argument is a single waveform.

`o_waveformMin` Returns a waveform (or family of waveforms) representing the minimum value of Y if the input argument is a family of waveforms.

`nil` Returns `nil` and an error message otherwise.

**Example**

`ymin( v( "/net9" ) )`

Gets the minimum voltage (Y value) of "/net9".
Advanced Analysis

The OCEAN commands for advanced analyses let you run parametric analysis, corners analysis, Monte Carlo analysis, and Optimization. This chapter includes setup commands for these analyses and the special data-access or plot commands that are used for these analyses.

The following sections contain the commands and other information relating to advanced analyses.

- Parametric Analysis Commands on page 335
- Corners Analysis Commands on page 341
- Monte Carlo Analysis Commands on page 348
- Optimization Commands on page 368

Parametric Analysis Commands

These commands set up a parametric analysis. When you run a parametric analysis, you can plot the resulting data as a family of curves.
paramAnalysis

\[
\text{paramAnalysis}( \ t_{desVar} \ [?\text{start \ } n_{\text{start}}] \ [?\text{stop \ } n_{\text{stop}}] \ [?\text{center \ } n_{\text{center}}] \ [?\text{span \ } n_{\text{span}}] \ [?\text{step \ } f_{\text{step}}] \ [?\text{lin \ } n_{\text{lin}}] \ [?\text{log \ } n_{\text{log}}] \ [?\text{dec \ } n_{\text{dec}}] \ [?\text{oct \ } n_{\text{oct}}] \ [?\text{times \ } n_{\text{times}}] \\
[?\text{spanPercent \ } n_{\text{spanPercent}}] \ [?\text{values \ } l_{\text{values}}] \\
[0_{\text{paramAnalysis}}]) \Rightarrow \text{undefined/nil}
\]

Description

Sets up a parametric analysis.

Groups the PSF data so that it can be plotted as a family of curves when the analysis is finished. The commands can be nested as shown in the syntax of the command.

If you specify more than one range, the OCEAN environment uses the following precedence to select a single range to use.

\[
\begin{align*}
\text{n_start, n_stop} & \quad \text{highest precedence} \\
\text{n_center, n_span} & \\
\text{n_center, n_spanPercent} & \quad \text{lowest precedence}
\end{align*}
\]

Similarly, if you specify more than one step control, the OCEAN environment uses the following precedence.

\[
\begin{align*}
\text{f_step} & \quad \text{highest precedence} \\
\text{n_lin} & \\
\text{n_dec} & \\
\text{n_log} & \\
\text{n_oct} & \\
\text{n_times} & \quad \text{lowest precedence}
\end{align*}
\]

To run the analysis, use the \text{paramRun} command described in “paramRun” on page 340.

Arguments

\[
\begin{align*}
t_{desVar} & \quad \text{Name of the design variable to be swept.} \\
n_{\text{start}} & \quad \text{Beginning value for the design variable.}
\end{align*}
\]
### n_stop
Final value for the design variable.

### n_center
Center point for a range of values that you want to sweep.

### n_span
Range of values that you want to sweep around the center point. For example, if `n_center` is 100 and `n_span` is 20 then the sweep range extends from 90 to 110.

### f_step
Increment by which the value of the design variable changes. For example, if `n_start` is 1.0, `n_stop` is 2.1, and `f_step` is 0.2, the parametric analyzer simulates at values 1.0, 1.2, 1.4, 1.6, 1.8, and 2.0.

### n_lin
The number of steps in the analysis. The parametric analyzer automatically assigns equal intervals between the steps. With this option, there is always a simulation at both `n_start` and `n_stop`. The value for the `n_lin` argument must be an integer greater than 0.

For example, if `n_start` is 0.5, `n_stop` is 2.0, and `n_lin` is 4, the parametric analyzer simulates at values 0.5, 1.0, 1.5, and 2.0.

### n_log
The number of steps between the starting and stopping points at equal-ratio intervals using the following formula:

\[ \text{log multiplier} = \left( \frac{n_{\text{stop}}}{n_{\text{start}}} \right)^{(n_{\text{log}}-1)} \]

The number of steps can be any positive number, such as 0.5, 2, or 6.25.

Default value: 5

For example, if `n_start` is 3, `n_stop` is 15, and `n_log` is 5, the parametric analyzer simulates at values 3, 4.48605, 6.7082, 10.0311, and 15.

The ratios of consecutive values are equal, as shown below.

\[ \frac{3}{4.48605} = \frac{4.48605}{6.7082} = \frac{6.7082}{10.0311} = \frac{10.0311}{15} = 0.67 \]

### n_dec
The number of steps between the starting and stopping points calculated using the following formula:
decade multiplier = \(10^{n_{\text{dec}}}\)

The number of steps can be any positive number, such as 0.5, 2, or 6.25.
Default value: 5

For example, if \(n_{\text{start}}\) is 1, \(n_{\text{stop}}\) is 10, and \(n_{\text{dec}}\) is 5, the parametric analyzer simulates at values 1, 1.58489, 2.51189, 3.98107, 6.30957, and 10.

The values are \(10^0, 10^2, 10^4, 10^6, 10^8\), and \(10^1\).

\(n_{\text{oct}}\)

The number of steps between the starting and stopping points using the following formula:
The number of steps can be any positive number, such as 0.5, 2, or 6.25.
Default value: 5

For example, if \(n_{\text{start}}\) is 2, \(n_{\text{stop}}\) is 4, and \(n_{\text{oct}}\) is 5, the parametric analyzer simulates at values 2, 2.2974, 2.63902, 3.03143, 3.4822, and 4.

These values are \(2^1, 2^{1.2}, 2^{1.4}, 2^{1.6}, 2^{1.8}\), and \(2^2\).

octave multiplier = \(2^{1/(n_{\text{oct}})}\)

\(n_{\text{times}}\)

A multiplier. The parametric analyzer simulates at the points between \(n_{\text{start}}\) and \(n_{\text{stop}}\) that are consecutive multiples of \(n_{\text{times}}\).

For example, if \(n_{\text{start}}\) is 1, \(n_{\text{stop}}\) is 1000, and \(n_{\text{times}}\) is 2, the parametric analyzer simulates at values 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512.

\(n_{\text{spanPercent}}\)

Range specified as a percentage of the center value. For example, if \(n_{\text{center}}\) is 100 and \(n_{\text{spanPercent}}\) is 40, the sweep range extends from 80 to 120.
OCEAN Reference
Advanced Analysis

$l_values$  List of values to be swept. You can use $l_values$ by itself or in conjunction with $n_start$, $n_stop$, and $f_step$ to specify the set of values to sweep.

$o_paramAnalysis$  Value returned from another $paramAnalysis$ call used to achieve multidimensional parametric analysis.

Value Returned

$undefined$  The return value for this command is undefined.

$nil$  Returns nil and prints an error message if there are problems setting the option.

Examples

$paramAnalysis( "rs" ?start 200 ?stop 1000 ?step 200 ?values (1030 1050 1090) )$

Sets up a parametric analysis for the $rs$ design variable. The swept values are 200, 400, 600, 800, 1000, 1030, 1050, and 1090.

$paramAnalysis( "rl" ?start 200 ?stop 600 ?step 200)$

$paramAnalysis( "rs" ?start 300 ?stop 700 ?step 200 )$

$paramAnalysis( "temp" ?start -50 ?stop 100 ?step 50 )$

Sets up a parametric analysis for temperature.
paramRun

paramRun( [s_paramAnalysis] ) => t/nil

Description

Runs the specified parametric analysis.

If you do not specify a parametric analysis, all specified analyses are run. Distributed processing must be enabled using the hostmode command before parametric analyses can be run in distributed mode.

When the paramRun command finishes, the PSF directory contains a file named runObjFile that points to a family of data. To plot the family, use a normal plot command. For example, you might use plot(v("/out")).

For information about specifying a parametric analysis, see the paramAnalysis command described in “paramAnalysis” on page 336.

Arguments

s_paramAnalysis

Parametric analysis.

Value Returned

t

Returned if successful.

nil

Returns nil and prints an error message if unsuccessful.

Examples

paramRun() => t

Runs all specified parametric analyses.

paramRun( 'rs ) => t

Runs the rs parametric analysis.
Corners Analysis Commands

The corners analysis commands let you set up and run analyses to measure circuit performance with respect to variations in a semiconductor manufacturing process. This section lists the commands that you can use to configure and run corners analyses in the OCEAN environment. The following manuals provide more information on corners analysis.

- Affirma Advanced Analysis Tools User Guide
- Affirma AMS Circuit Design Environment SKILL Language Reference

The corners analysis commands follow.
cornerDesVar

cornerDesVar(t_cornerName t_desVarName t_value) => t/nil

Description
Sets the design variable value for the specified corner.

Arguments

- \textit{t\_cornerName} \quad \text{Name of the corner.}
- \textit{t\_desVarName} \quad \text{Name of the design variable.}
- \textit{t\_value} \quad \text{Value of the design variable.}

Value Returned

- \textit{t} \quad \text{Returned if successful.}
- \text{nil} \quad \text{Returns nil and prints an error message.}

Example

\texttt{cornerDesVar("slow" "vcc" "5")}

Sets the value of \texttt{vcc} to 5 for corner \texttt{slow}. 
cornerMeas

cornerMeas() => t/nil

Description

Displays all the predefined enabled measurements from a Design Customization file, either graphically (plot) or textually (print), according to your choices.

Each measurement is plotted or printed in a separate subwindow.

Arguments

None.

Value Returned

<table>
<thead>
<tr>
<th>t</th>
<th>Returned if successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>Returns nil and prints an error message.</td>
</tr>
</tbody>
</table>

Example

cornerMeas()
cornerRun

cornerRun( [t_cornerName1 t_cornerName2 ...]
[?jobName t_jobName] [?host t_hostName] [?queue t_queueName]
[?startTime t_startTime] [?termTime t_termTime] [?dependentOn t_dependentOn] [?mail t_mailingList] [?block s_block]
[?notify s_notifyFlag] ) => t/s_jobName/nil

Description

Runs the corner analysis that has been predefined in the .pcf and .dcf files and selected via the selectProcess command. If specific corners are specified, only those corners run; otherwise all the corners run.

You can load your .pcf and .dcf files with the loadPcf and loadDcf commands. See the Affirma AMS Circuit Design Environment SKILL Language Reference for information on these commands.

Arguments

t_cornerName A specific corner to be run. If you do not specify one or more corners, then all the enabled corners run.

Note: The following arguments are valid only when running in distributed processing mode.

t_jobName Used as the basis of the job name. The value entered for t_jobName is used as the job name and return value if the run command is successful. If the name given is not unique, a number is appended to create a unique job name.

t_hostName Name of the host on which to run the analysis. If no host is specified, the system assigns the analysis to an available host.

t_queueName Name of the queue. If no queue is defined, the analysis is placed in the default queue (your home machine).

t_startTime Desired start time for the job. If dependencies are specified, the job does not start until all dependencies are satisfied.

t_termTime Termination time for job. If the job is not completed by t_termTime, the job is terminated.
**OCEAN Reference**

**Advanced Analysis**

### t_dependentOn
List of jobs on which the specified analysis is dependent. The analysis is not started until after dependent jobs are complete.

### t_mailingList
List of users to be notified by e-mail when the analysis is complete.

### s_block
When `s_block` is not `nil`, the OCEAN script halts until the job is complete.
Default value: `nil`

### s_notifyFlag
When `s_notifyFlag` is not `nil`, a job completion message is echoed to the OCEAN interactive window.
Default value: `t`

### Value Returned

**t**
Returned if successful.

**s_jobName**
For a distributed process, the job name specified or assigned by the system to the analysis.

**nil**
Returns `nil` and prints an error message.

### Examples

**cornerRun()**

Runs all corners analysis defined in the `.pcf` and `.dcf` files and selected by the `selectProcess` command.

```ocean
cornerRun( ?startTime 10 ?host "mach14" ?mail "preampGroup")
```

Runs all corners analysis defined in the `.pcf` and `.dcf` files and selected by the `selectProcess` command in distributed mode with a `startTime` of 10, using `mach14` as `host`, and notifying the mail group `preampGroup` when the analysis is complete.
cornerRunTemp

cornerRunTemp(t_cornerName t_value) => t/nil

Description

Sets the analysis temperature (in degrees Celsius) to be used for a corner.

Arguments

t_cornerName  Name of the corner.
t_value       Temperature value in degrees Celsius.

Value Returned

t               Returned if successful.
nil             Returns nil and prints an error message.

Example

cornerRunTemp("slow" "50")

Sets the temperature to 50 for corner slow.
residual

description

Creates a residual plot of the given scalar expression given the upper and lower performance bounds and target.

arguments

x_scalarExpression     Scalar expression from a corners analysis.
x_upperValue           Upper performance bound.
x_targetValue          Target value.
x_lowerValue           Lower performance bound.

value returned

t            Returned if successful.
nil          Returns nil and prints an error message.

example

residual( bandwidth(v("net1"), 3, "low") ?upper 5 ?target 2.5 ?lower 0)

Creates a residual plot of v("net1") with an upper boundary of 5, a target of 2.5, and a lower boundary of 0.
selectProcess

selectProcess(t_processName) => t/nil

Description

Selects one of the processes already loaded with a `loadPcf` or `loadDcf` command.

Arguments

`t_processName` Name of the process, as specified in the `.pcf` or `.dcf` file with the `corAddProcess` function.

Value Returned

`t` Returned if successful.

`nil` Returns `nil` and prints an error message.

Example

selectProcess("fab6")

Selects the process `fab6`.

Monte Carlo Analysis Commands

The commands for running Monte Carlo in the OCEAN environment are as follows.
correlationTable

correlationTable(?suppress x_suppress) => t/nil

Description

Prints the correlation between all pairs of declared monteExpr expressions.

Pairs of the same expression, which have a correlation value of 1.0, are excluded. This exclusion means that the correlationTable command prints only the off-diagonal terms in the correlation matrix.

Arguments

x_suppress

Suppresses the printing for correlations less than this value.
Default value: .5

Value Returned

t
Returned if successful.

nil
Returned otherwise.

Example

correlationTable()
dataFilter

dataFilter(t_monteExprName ( {?sigma x_sigma | ?upper x_upper
  ?lower x_lower} ?filterBy s_filterBy ) ) => t/nil

Description

Eliminates bad data points (outliers) from a Monte Carlo data set.

Arguments

t_monteExprName

The monteExpr name with the appended swept parameter.

x_sigma

Filters data lying outside an established sigma point from the mean. For instance, you might filter data lying outside 3 standard deviations (sigma) from the mean. You can specify x_sigma or you can specify x_upper and x_lower, but you cannot specify both.
Default value: 3

x_upper

Filters data that is greater than an upper numerical limit.
Default value: inf

x_lower

Filters data that is less than a lower numerical limit.
Default value: -inf

s_filterBy

Type of filter to be used. This setting affects all of your data so you only need to specify the type of filter once.
Valid values: ’dataSet,’ point
Default value: ’dataSet

’dataSet ignores all measurements for a point if the value of any of the measurements for that point is outside the filter limits.

’point filters an outlying point only from the specific measurement that recorded the outlying point.
Value Returned

- t: Returned if successful.
- nil: Returned otherwise.

Example

dataFilter('bandwidth ?upper 10Mhz
  ?lower 0.1Mhz'); For nominal 1Mhz

dataFilter('bandwidth ?sigma 3')

The second example sets the upper limit to
mean(bandwidth) + 3*sigma(bandwidth)
and sets the lower limit to
mean(bandwidth) - 3*sigma(bandwidth)
**histogram**

```
histogram( t_monteExprName ?type s_type ?numBins x_numBins
    ?density b_density ) => t/nil
```

**Description**

Plots a histogram of Monte Carlo data.

This command plots to an individual subwindow. The value of the `s_type` argument determines the style of the line. Setting `b_density` to `t` causes the `histogram` command to plot a smooth distribution curve for the data.

**Arguments**

- **`t_monteExprName`**
  - The `monteExpr` name with the appended swept parameter.

- **`s_type`**
  - Style of line to be used.
  - Valid values: `standard`, `passFail`, `cumulativeLine`, `cumulativeBox`
  - Default value: `standard` (if you do not specify `s_type`)

  - `'standard` prints a bar graph of the output versus parameter.

  - `'passFail` requires that specification limits be specified. This option plots a bar graph where the runs that pass are shown in green and the runs that fail are shown in red.

  - `'cumulativeLine` uses a *joined* line style to plot the cumulative distribution function. The cumulative distribution function is the area under the standard histogram bars.

  - `'cumulativeBox` plots the same information as the `'cumulativeLine` option but uses a *bar* plotting style.

- **`x_numBins`**
  - Number of bins to be used for the histogram.
  - Default value: 10
b_density

If set to t, plots the probability density function for the data. Valid values: t or nil.

Value Returned

t Returned if successful.

nil Returned otherwise.

Example

monteExpr( "bw" 'bandwidth( v("vout"),3,"low") )
monteExpr( "DCgain" 'ymax( vdb("vout") ) )

histogram( "bw_27" )
histogram( "bw_27" ?numBins 12 ?density t )
iterVsValue

iterVsValue( t_monteExprName ?outputFormat s_outputFormat ) => t/ nil

Description
Prints the value of every scalar measurement for each Monte Carlo iteration.

Arguments

t_monteExprName
  The monteExpr name with the appended swept parameter.

s_outputFormat
  The output format for the printout.
  Valid values: ‘sorted,’ unsorted
  Default value: ‘sorted

  ‘sorted sorts the output from highest to lowest value.

  ‘unsorted prints the values without sorting.

Value Returned

t
  Returned if successful.

nil
  Returned otherwise.

Example

iterVsValue( "bw_27" )
monteCarlo

monteCarlo(
    [?numIters x_numIters] [?startIter x_startIter]
    [?analysisVariation s_analysisVariation] [?sweptParam
t_sweptParam] [?sweptParamVals l_sweptParamVals] [?saveData
saveData] [?append b_append] ) => t/nil

Description

Sets up a Monte Carlo analysis.

To run the analysis, use the monteRun command described in “monteRun” on page 362.

Arguments

x_numIters Number of iterations (runs).
Default value: 100

x_startIter Starting iteration.
Default value: 1

Note: x_startIter must not be 1 when

■ You want to append to existing data. For example, you run 100 Monte Carlo analyses and then want to run 100 more in addition to the previous 100. In this case, x_startIter must be 101. If x_startIter is 1, the same results are recalculated as before.

■ You want to rerun a particular run. In this case, startIter must be the number of that particular run.

s_analysisVariation Analysis variations.
Valid values: 'process, 'mismatch,'processAndMismatch
Default value: 'process

t_sweptParam Design variable (or temperature) that can be swept with Monte Carlo.
Default value: none (this is the inner loop)
OCEAN Reference
Advanced Analysis

\textit{t\_sweptParamVals} 
List of values of \textit{sweptParam}.

\textit{saveData} 
Indicates when to save data to allow family plots. 
Default value: \texttt{nil}

\textit{b\_append} 
Appends the new results to data from a previous Monte Carlo run. 
Default value: \texttt{nil}

Value Returned
\texttt{t} 
Returned if successful.
\texttt{nil} 
Returned otherwise.

Example
\texttt{monteCarlo()}
Sets up a Monte Carlo analysis using some of the defaults.
\texttt{monteCarlo(?numRuns 300 ?analysisType \textquote{processAndMismatch} ?sweptParam temp ?sweptParamVals list(-50, 0, 50) ?nomRun n)}
monteCorrelate

monteCorrelate( f_correlationValue { t_param1 ... t_paramN | t_deviceName1 ... t_deviceNameN} ) => t/nil

Description

Specifies a correlation coefficient for a list of process parameters or a list of devices specified in individual subcircuits.

Use this command to specify matched pairs of devices or to specify mismatch of devices in excess of that specified for the process. You must not mix devices and parameters on the same command line.

Arguments

f_correlationValue

Value of the correlation coefficient that describes the correlation among the listed parameters or devices.

t_param1

Name of the first process parameter to be correlated.

t_paramN

Name of another process parameter to be correlated.

t_deviceName1

Name of the first device to be correlated.

t_deviceName2

Name of another device to be correlated.

Value Returned

t

Returned if successful.

nil

Returned otherwise.

Example

monteCorrelate( 0.7 "pbsr" "nmpbsr" )
**monteDisplay**

monteDisplay() => undefined/nil

**Description**

Displays the currently defined Monte Carlo analysis, including all expressions that are defined.

**Arguments**

None.

**Value Returned**

- **undefined**
  
  The return value for this command/function is undefined.

- **nil**
  
  Returns nil and prints an error message if the analysis is not specified.

**Example**

monteDisplay()
**monteExpr**

```
monteExpr( t_monteExprName s_expression ) => t/nil
```

**Description**

Sets up the Monte Carlo scalar expressions that are used to create the histogram file.

**Arguments**

- **t_monteExprName**: Name of the expression.
- **s_expression**: Expression.

**Value Returned**

- **t**: Returned if successful.
- **nil**: Returned otherwise.

**Example**

```
monteExpr( "bw" 'bandwidth( v("net7") 3 "low") )
```
**monteOutputs**

monteOutputs() => t/nil

**Description**

Returns the names of the `monteExpr` expressions, concatenating the `monteExprName` set in the `monteExpr` command with the value of the swept variable.

If no variable is swept, the `monteOutputs` command concatenates the default temperature to the `monteExprName`. For example, the returned name might be `bw_27`.

**Arguments**

None.

**Value Returned**

- **t**: Returned if successful.
- **nil**: Returned otherwise.

**Example**

`monteOutputs()`
monteResults

monteResults(?dataFileName  t_scalarDataFile  ?paramFileName  t_parameterFile) => t/nil

Description

Initializes the Monte Carlo data analysis tools.

The `monteResults` command reads in the specified data and parameter files, opens a new Waveform window, and adds a statistical analysis menu to the Waveform window. The menu items are equivalent to those found on the *Monte Carlo Results* menu in the Affirma™ analog circuit design environment.

Arguments

- **t_scalarDataFile**
  - Name of scalar data file to be read in.
  - Default value: *mcdata*

- **t_parameterFile**
  - Name of parameter file associated with scalar data.
  - Default value: *mcparam*

Value Returned

- **t**
  - Returned if successful.

- **nil**
  - Returned otherwise.

Examples

- `monteResults()`
- `monteResults( ?dataFileName "myData" ?paramFileName "myParams" )`
monteRun

monteRun(
    [?jobName t_jobName] [?host t_hostName] [?tasks x_tasks]
    [?queue t_queueName] [?startTime t_startTime] [?termTime
    t_termTime] [?dependentOn t_dependentOn] [?mail
    t_mailingList] [?block s_block] [?notify s_notifyFlag] ) =>
    s_jobName/nil/t

Description

Runs a Monte Carlo analysis previously set up with the monteCarlo and monteExpr commands.

The monteRun command runs all the Monte Carlo processes defined in the .pcf and .dcf files. You can load your .pcf and .dcf files with the loadPcf and loadDcf commands. See the Artist SKILL Language Reference Manual for information on these commands.

Arguments

Note: Arguments to the monteRun command are valid only when running in distributed (processing) mode.

- **t_jobName**: Used as the basis of the job name. The value entered for t_jobName is used as the job name and return value if the run command is successful. If the name given is not unique, a value is appended to create a unique job name.

- **t_hostName**: Name of the host on which to run the analysis. If no host is specified, the system assigns the analysis to an available host.

- **x_tasks**: Number of tasks in which to divide the Monte Carlo job. Default value: calculated from your setup

- **t_queueName**: Name of the queue. If no queue is defined, the analysis is placed in the default queue (your home machine).

- **t_startTime**: Desired start time for the job. If dependencies are specified, the job does not start until all dependencies are satisfied.

- **t_termTime**: Termination time for job. If the job has not completed by t_termTime, the job is terminated.
### t_dependentOn
List of jobs on which the specified analysis is dependent. The analysis is not started until after dependent jobs are complete.

### t_mailingList
List of users to be notified when the analysis is complete.

### s_block
When s_block is not nil, the OCEAN script halts until the job is complete.
Default value: nil

### s_notifyFlag
When s_notifyFlag is not nil, a job completion message is echoed to the OCEAN interactive window.
Default value: t

### Value Returned
- **t**: Returned if successful.
- **nil**: Returned otherwise.

### s_jobName
For a distributed process, the job name that the system specified or assigned to the analysis.

### Example
```
monteRun()
```
Runs all the Monte Carlo analyses defined in the .pcf and .dcf files.
monteSelectResults

monteSelectResults( ?mcdataFileName t_mcdataFileName
                   ?paramFileName  t_paramFileName ) => t/nil

Description

Selects the specified mcdata file, which is the file that contains the scalar data.

Before you use this command, you must have access to mcdata and param files, either produced by an earlier successful Monte Carlo simulation or pointed to by a previous openResults() command.

Arguments

\( t\_mcdataFileName \)

The name of the mcdata file.
Default value: mcdata

\( t\_paramFileName \)

The name of the param file.
Default value: param

Value Returned

\( t \)

Returned if successful.

\( \text{nil} \)

Returned otherwise.

Example

monteSelectResults()
monteSelectResults(?mcdataFileName mcdataRun2  
                   ?paramFileName paramRun2)
scatterplot

scatterplot( t_monteExprName_X t_monteExprName_Y ?bestFit
    b_bestFit ) => t/nil

Description

Plots different statistical measurements against each other so you can determine whether there is a relationship between two parameters.

Tightly correlated parameters show linear relationships.

Arguments

\textit{t\_monteExprName\_X}

The \texttt{monteExpr} name with the appended swept parameter for the X-axis variable.

\textit{t\_monteExprName\_Y}

The \texttt{monteExpr} name with the appended swept parameter for the Y-axis variable.

\textit{b\_bestFit}

If \texttt{t}, the \texttt{scatterplot} command computes and draws on the plot the best fitting straight line through the data. The best line is defined as the line that minimizes the sum of squares of the distances between the data points and the line.

Value Returned

\texttt{t} 

Returned if successful.

\texttt{nil} 

Returned otherwise.

Example

\texttt{monteExpr( "bw" 'bandwidth( v("vout"), 3, "low") )}
\texttt{monteExpr( "DCgain" 'ymax( vdb("vout") ) )}
\texttt{scatterplot( "bw\_27" "DCgain\_27" ?bestFit t )}
specLimits

```
specLimits(t_monteExprName ( {?sigma x_sigma | ?upper x_upper
    ?lower x_lower} ) ) => t/nil
```

**Description**

Sets specification limits for yield analysis and histograms.

You can set specification limits for each of your measured values and then analyze how many runs are outside those limits (pass/fail) or you can analyze the spec sensitivity of measured quantities to changing input parameters.

You can specify limits using `x_upper` and `x_lower` options, or you can use the `x_sigma` option to have limits calculated for you based on a specified number of standard deviations of the actual data.

**Note:** You can specify `x_sigma` or you can specify `x_upper` and `x_lower`, but you cannot specify both.

**Arguments**

- **t_monteExprName**
  - The `monteExpr` name with the appended swept parameter.

- **x_sigma**
  - Identifies data lying outside an established sigma point from the mean. For instance, you might identify data lying outside 3 standard deviations (sigma) from the mean.

- **x_upper**
  - Identifies data that is greater than this value.

- **x_lower**
  - Identifies data that is less than this value.

**Value Returned**

- **t**
  - Returned if successful.

- **nil**
  - Returned otherwise.

**Example**

```
specLimits("bw_27" ?upper 15E+06 ?lower 5+06)
```
yield

yield( s_type l_monteExprName ?given l_monteExprName ?suppress x_suppress ) => t/nil

Description

Prints simple, conditional, or multiconditional yield statistics for the Monte Carlo data set.

Arguments

s_type  The type of statistics to print.
Valid values: 'simple,' 'cond, or 'multiCond

'simple prints the yields for each measurement. Based upon the specification limits you set, the 'simple option prints the percentage of pass runs compared to the total number of Monte Carlo runs. For example, you set your specification limits for bandwidth, run 100 runs, and find that 60 of the runs pass the specification limits. For this example, the yield command calculates and displays a yield of 60% for bandwidth. The command also displays the total yield number, which is used when you have multiple measurements, each with its own limits. Total yield is the total percentage of pass runs where every parameter is within its specification limits for a Monte Carlo run.

'cond prints conditional yields. To use a conditional yield, you specify a single measurement against which all other measurements are compared. The 'cond option first sorts all of the Monte Carlo runs and picks out only the runs where the specified measurement passes. These passing runs are the starting point for the conditional yield calculation. So, in the bandwidth example above, instead of using 100 runs, the tool uses 60 runs as the base. Next, all of the other measurements are analyzed. For example, you have a second measurement called maximum_25. Out of the base 60 runs, maximum_25 passes 30 times. It has a conditional yield of 50%. In addition to the conditional yield, the tool prints the total yield (based on all Monte Carlo runs) and the difference between the conditional and total yield numbers.

'multiCond prints multiconditional yields. As in calculating the
conditional yield, multiconditional yields are calculated from a base set of passing runs. However, instead of using one parameter to build the base set, for multiconditional yields you use two. Only runs where both measurements pass become part of the base set. All other measurements are then compared against that base.

l_monteExprName

The monteExpr name with the appended swept parameter.

x_suppress

If s_type is 'simple, suppresses the printing for yields greater than this percentage of the value. Default value: 98

If s_type is 'cond or 'multiCond, suppresses the printing for delta yields less than this percentage of the value. Default value: 98

Value Returned

t Returned if successful.

nil Returned otherwise.

Example

yield('simple("bw_27" "slew_27") ?suppress 70)
yield('cond("max_27" "slew_27") ?given "bw_27")
yield('multiCond("max_27" "slew_27") ?given "bw_27" "CMPR_27")

Optimization Commands

The commands for running optimization in the OCEAN environment are as follows.
optimizeAlgoControl

optimizeAlgoControl( ?relDelta x_relDelta ?relFunTol x_relFunTol ?relVarTol x_relVarTol ) => undefined/nil

Description
Changes the internal algorithm controls.

Arguments

x_relDelta Finite difference relative perturbation.
Default value: .005

x_relFunTol Relative function convergence tolerance.
Default value: .0001

x_relVarTol Relative variable convergence tolerance.
Default value: .0001

Value Returned

undefined The return value for this function is not defined.

nil Returns nil and an error message if there was a problem.

Example
optimizeAlgoControl(?relDelta .05)
optimizeGoal

optimizeGoal( t_name t_expr s_direction x_target x_acceptable
[?percent b_percent] ) => undefined/nill

**Description**

Sets up the goals for optimization.

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_name</td>
<td>Name of the goal.</td>
</tr>
<tr>
<td>t_expr</td>
<td>Expression defining the goal.</td>
</tr>
<tr>
<td>s_direction</td>
<td>Valid values: ‘max’, ‘min’, ‘match’, ‘le’ or ‘ge’</td>
</tr>
<tr>
<td></td>
<td>Default value: ‘match’</td>
</tr>
<tr>
<td>x_target</td>
<td>The value to be matched or the lower or upper bound (depending on s_direction).</td>
</tr>
<tr>
<td>x_acceptable</td>
<td>Number or a waveform specifying the acceptable value. When a waveform is entered, each target point has its own acceptable value. Both x_target and x_acceptable must be expressions. The expression returns a number or a waveform.</td>
</tr>
<tr>
<td>b_percent</td>
<td>Specifies whether the x_acceptable field is a percentage of the target. When this is specified, x_acceptable is ignored.</td>
</tr>
</tbody>
</table>

**Value Returned**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined</td>
<td>The return value for this command is not defined.</td>
</tr>
<tr>
<td>nil</td>
<td>Returns nil and an error message if there was a problem.</td>
</tr>
</tbody>
</table>

**Example**

optimizeGoal( "bandwidth" 'bandwidth(v("\out") 3 "low") 'le 18M 15M )
optimizePlotOption

optimizePlotOption(
    ?auto b_auto ?varHist b_varHist ?scalHist b_scalHist
    ?funcObjHist b_funcObjHist ?numIter x_numIter ?fontSize
    x_fontSize ?width x_width ?height x_height ?xloc xloc ?yloc
    yloc ) => undefined/nil

Description

Sets the plot options used to view the optimization iterations.

Arguments

b_auto If set to t, auto plots after each iteration.  
Default value: t

b_varHist If set to t, displays the history of the variables.  
Default value: t

b_scalHist If set to t, displays the history of the scalars.  
Default value: t

b_funcObjHist If set to t, displays the history of the functional objectives.  
Default value: t

x_numIter Number of waveforms to display. There is one waveform stored 
available per functional iteration.  
Default value: 5

x_fontSize Font size used in the Waveform window.  
Default value: 9

x_width Width of the Waveform window.  
Default value: 630

x_height Height of the Waveform window.  
Default value: 376

xloc Specifies the top boundary of the optimize window when it is 
opened in the windowing environment.  
Default value: 511
yloc

Specifies the left boundary of the optimize window when it is opened in the windowing environment.
Default value: 3

Value Returned

undefined

The return value is for this value is not defined.

nil

Returns nil and an error message if there was a problem setting plot options.

Example

optimizePlotOption(?delta .05)
**optimizeRun**

```prolog
optimizeRun(?goals l_goalNames ?vars l_varNames ?numIter
   x_numIter  ?algo s_algoName ?continue b_continue) => t/nil
```

**Description**

Runs the optimizer using the goals specified with the `optimizeGoal` command.

**Arguments**

- **l_goalNames**
  Names of the goals to be used with this run of the optimizer. If none are specified, all declared goals are used.

- **l_varNames**
  Names of the variables to be used with this run of the optimizer. If none are specified, all declared variables are used.

- **x_numIter**
  Number of iterations that you want the optimizer to perform.

- **s_algoName**
  Algorithm that you want to use.
  Valid values: ‘lsq’, ‘cfsqp’, ‘auto’

- **b_continue**
  t indicates that this `optimizeRun` needs to continue from the previous `optimizeRun` (using the last design variables calculated from the last `optimizeRun`).

**Value Returned**

- **t**
  If the command was successful.

- **nil**
  Returns nil and an error message if there was a problem.

**Example**

```prolog
optimizeRun()
optimizeRun(?goals '("bandwidth" "slewrate")
   ?vars '("rs" vs") ?iter 5)
optimizeRun( ?numIter 5 ?continue t )
```

Continues the previous `optimizeRun` for another 5 iterations.
**optimizeVar**

optimizeVar( t_name x_initVal x_minVal x_maxVal ) => undefined/nil

**Description**

Specifies the design variables to be used with optimization.

**Arguments**

- **t_name**: Name of the design variable.
- **x_initVal**: Initial value of the variable.
- **x_minVal**: Lower bound of the variable.
- **x_maxVal**: Upper bound of the variable.

**Value Returned**

- **undefined**: The return value for this function is not defined.
- **nil**: Returns nil and an error message if there was a problem.

**Example**

optimizeVar( "res" 100 ?minVal 1 ?maxVal 1000 )
OCEAN Distributed Processing Commands

The Open Command Environment for Analysis (OCEAN) distributed processing commands let you run OCEAN jobs across a collection of computer systems. This chapter contains information on the following commands:

- **deleteJob** on page 376
- **digitalHostMode** on page 377
- **digitalHostName** on page 378
- **hostMode** on page 379
- **hostName** on page 380
- **killJob** on page 381
- **monitor** on page 382
- **remoteDir** on page 383
- **resumeJob** on page 384
- **suspendJob** on page 385
- **wait** on page 386
deleteJob

deleteJob( t_jobName [t_jobName2 t_jobName3 ... t_jobNameN] ) => t/ nil

Description

Removes a job or series of jobs from the text-based job monitor.

Deleted jobs are no longer listed in the job monitor. The deleteJob command applies only to ended jobs.

Arguments

t_jobName

Name used to identify the job.

t_jobName2...t_jobNameN

Additional jobs that you want to delete.

Value Returned

t

Returns t if successful.

nil

Returns nil and prints an error message if unsuccessful.

Example

deleteJob( ‘myckt) => t

Deletes the myckt job.
**digitalHostMode**

digitalHostMode( {'local  |  'remote} ) => t/nil

**Description**

For mixed-signal simulation, specifies whether the digital simulator will run locally or on a remote host.

**Arguments**

- **'local**
  Sets the simulation to run locally on the user’s machine.

- **'remote**
  Sets the simulation to run on a remote host. If you use this argument, you must specify the host name by using the `digitalHostName` command.

**Value Returned**

- **t**
  Returns t if successful.

- **nil**
  Returns nil and prints an error message if unsuccessful.

**Example**

digitalHostMode( 'local' )

Sets the digital simulator to run locally on the user’s host.
digitalHostName

digitalHostName( t_name ) => t/nil

Description

For mixed-signal simulation, specifies the name of the remote host for the digital simulator.

When you use the digitalHostMode('remote) command, use this command to specify the name of the remote host.

Arguments

\texttt{t\_name} \hspace{1cm} \text{Name used to identify the host for the digital simulator.}

Value Returned

\texttt{t} \hspace{1cm} \text{Returns } t \text{ if successful.}

\texttt{nil} \hspace{1cm} \text{Returns nil and prints an error message if unsuccessful.}

Example

digitalHostName( "digitalhost" )

Indicates that the digital simulator runs on the host called \texttt{digitalhost}. 
hostMode

hostMode( { 'local | 'remote | 'distributed } ) => t/nil

Description

Sets the simulation host name.

The default value for hostMode is specified in the asimenv.startup file with the hostMode environment variable.

Arguments

'local
Sets the simulation to run locally on the user's machine.

'remote
Sets the simulation to run on a remote host queue. For this release, the remote host is specified in the .cdsenv file.

'distributed
Sets the simulation to run using the distributed processing software.

Value Returned

t
Returns t if successful.

nil
Returns nil and prints an error message if unsuccessful.

Example

hostMode( 'distributed ) => t

Enables distributed processing on the current host.
**hostName**

hostName( t_name ) => t/nil

**Description**

Specifies the name of the remote host.

When you use the hostMode('remote) command, use this command to specify the name of the remote host.

**Arguments**

\[ t\_name \]

Name used to identify the remote host.

**Value Returned**

\[ t \]

Returns \[ t \] if successful.

\[ nil \]

Returns \[ nil \] and prints an error message if unsuccessful.

**Example**

hostName( "remotehost" )

Specifies that the host called remotehost is to be used for remote simulation.
killJob

killJob( t_jobName [ t_jobName2 t_jobName3 ... t_jobNameN] ) => t/nil

Description

Stops processing of a job or series of jobs.

The job might still show up in the job monitor, but it cannot be restarted. Use the deleteJob command to remove the job name from the job server and job monitor.

Arguments

\[t\_jobName\]

Name used to identify the job.

\[t\_jobName2...t\_jobNameN\]

Additional jobs that you want to stop.

Value Returned

\[t\]

Returns t if successful.

\[nil\]

Returns nil and prints an error message if unsuccessful.

Example

\[\text{killJob( 'myckt ) => t}\]

Aborts the job called myckt. If the job is in the queue and has not started running yet, it is deleted from the queue.
**monitor**

monitor( [?taskMode s_taskMode] ) => t/nil

**Description**

Monitors the jobs submitted to the distributed system.

**Arguments**

*s_taskMode*  
When not nil, multitask jobs are expanded to show individual jobs. A multitask job is one that contains several related jobs.

**Value Returned**

*t*  
Returns t if successful.

*nil*  
Returns nil and prints an error message if unsuccessful.

**Example**

monitor( ?taskMode t )

Displays the name, host, and queue for all pending tasks sorted on a queue name.
remoteDir

remoteDir( t_path ) => t/nil

Description

Specifies the project directory on the remote host to be used for remote simulation.

When you use the hostMode(‘remote) command, use this command to specify the project directory on the remote host.

Arguments

| t_path  | Specifies the path to the project directory on the remote host to be used for remote simulation. |

Value Returned

| t       | Returns t if successful. |
| nil     | Returns nil and prints an error message if unsuccessful. |

Example

remoteDir( "~/simulation" )

Specifies that the project directory is ~/simulation.
resumeJob

resumeJob( t_jobName [t_jobName2 t_jobName3 ... t_jobNameN] ) => t/nil

Description

Resumes the processing of a previously suspended job or series of jobs. The resumeJob command applies only to jobs that are suspended.

Arguments

- t_jobName
  - Name used to identify the job.
- t_jobName2...t_jobNameN
  - Additional jobs that you want to resume

Value Returned

- t
  - Returns t if successful.
- nil
  - Returns nil and prints an error message if unsuccessful.

Example

resumeJob( 'myckt' ) => t

Resumes the myckt job that was halted with the suspendJob command.
suspendJob

suspendJob( t_jobName [t_jobName2 t_jobName3 ... t_jobNameN] ) => t/nil

Description

Suspends the processing of a job or series of jobs. The suspendJob command applies only to jobs that are pending or running.

Arguments

t_jobName
Name used to identify the job.

t_jobName2...t_jobNameN
Additional jobs that you want to suspend.

Value Returned

t
Returns t if successful.

nil
Returns nil and prints an error message if unsuccessful.

Example

suspendJob( ’myckt ) => t

Suspends the job called myckt.
wait

wait( jobName [jobName2 jobName3 ... jobNameN] ) => t/nil

Description

Postpones processing of a script until the specified jobs complete. This command is ignored if distributed processing is not available.

Arguments

\textit{t_jobName} Name used to identify the job. The job name is user defined or system generated, depending on how the user submitted the job.

\textit{t_jobName2...t_jobnameN} Additional jobs that you want to postpone.

Value Returned

\textit{t} Returns \textit{t} if successful.

\textit{nil} Returns \textit{nil} and prints an error message if unsuccessful.

Example

wait( 'myckt1 ) => t

Postpones execution of all subsequent OCEAN commands until the job \textit{myckt1} completes.
Language Constructs

There are three types of SKILL language constructs:

- Conditional statements
  
  Conditional statements test for a condition and perform operations when that condition is found. These statements are `if`, `unless`, and `when`.

- Selection statements
  
  A selection statement allows a list of elements, each with a corresponding operation. A variable can then be compared to the list of elements. If the variable matches one of the elements, the corresponding operation is performed. These statements include `for`, `foreach`, and `while`.

- Iterative statements
  
  Iterative statements repeat an operation as long as a certain condition is met. These statements include `case` and `cond`.

This chapter contains information on the following statements:

- `case` on page 397
- `cond` on page 399
- `for` on page 392
- `foreach` on page 394
- `if` on page 388
- `unless` on page 390
- `when` on page 391
- `while` on page 396
if

if( g_condition g_thenExpression [g_elseExpression] ) =>
g_result

Description

Evaluates g_condition, typically a relational expression, and runs
g_thenExpression if the condition is true (that is, its value is non-nil); otherwise, runs
g_elseExpression.

The value returned by if is the value of the corresponding expression evaluated.

Arguments

g_condition Any Cadence® SKILL language expression.

g_thenExpression Any SKILL expression.

g_elseExpression Any SKILL expression.

Value Returned

Returns the value of g_thenExpression if g_condition
has a non-nil value. The value of g_elseExpression is
returned otherwise.

Examples

x = 2
if( x > 5 1 0 ) => 0

Returns 0 because x is less than 5.

a ="npn"
if(( a == "npn" ) print( a ) ) "npn" => nil

Prints the string npn and returns the result of print.

x = 5
if( x "non-nil" "nil" ) => "non-nil"
Returns "non-nil" because \( x \) was not nil. If \( x \) was nil, "nil" would be returned.

\[
x = 7 \\
\text{if}(x > 5) \Rightarrow 1
\]

Returns 1 because \( x \) is greater than 5.
**unless**

unless( g_condition g_expr1 ... ) => g_result/nil

**Description**

Evaluates a condition. If the result is true (non-nil), it returns `nil`; otherwise it evaluates the body expressions in sequence and returns the value of the last expression.

The semantics of this function can be read literally as “unless the condition is true, evaluate the body expressions in sequence.”

**Arguments**

- **g_condition**: Any SKILL expression.
- **g_expr1**: Any SKILL expression.

**Value Returned**

- **g_result**: Returns the value of the last expression of the sequence `g_expr1 ...` if `g_condition` evaluates to `nil`.
- **nil**: Returns `nil` if `g_condition` evaluates to non-nil.

**Examples**

```plaintext
x = -123
unless( x >= 0 println( "x is negative" ) -x ) => 123
```

Prints "x is negative" as a side effect.

```plaintext
unless( x < 0 println( "x is positive ") x) => nil
```

Returns `nil`. 

**when**

when( g_condition g_expr1 ... ) => g_result/nil

**Description**

Evaluates a condition.

If the result is non-nil, evaluates the sequence of expressions and returns the value of the last expression. Otherwise, returns nil.

**Arguments**

- **g_condition**: Any SKILL expression.
- **g_expr1...**: Any SKILL expression.

**Value Returned**

- **g_result**: Returns the value of the last expression of the sequence if **g_condition** evaluates to non-nil.
- **nil**: returns nil if the **g_condition** expression evaluates to nil.

**Examples**

```plaintext
x = -123
when( x < 0 println( "x is negative" ) -x ) => 123
Prints "x is negative" as a side effect.
when( x >= 0 println( "x is positive" ) x ) => nil
Returns nil.
```
**for**

```lang-ocm
for( s_loopVar x_initialValue x_finalValue g_expr1 [g_expr2 ...] ) => t
```

**Description**

Evaluates the sequence `g_expr1 g_expr2 ...` for each loop variable value, beginning with `x_initialValue` and ending with `x_finalValue`.

First evaluates the initial and final values, which set the initial value and final limit for the local loop variable named `s_loopVar`. Both `x_initialValue` and `x_finalValue` must be integer expressions. During each iteration, the sequence of expressions `g_expr1 g_expr2 ...` is evaluated and the loop variable is then incremented by one. If the loop variable is still less than or equal to the final limit, another iteration is performed. The loop ends when the loop variable reaches a value greater than the limit. The loop variable must not be changed inside the loop. It is local to the `for` loop and would not retain any meaningful value upon exit from the `for` loop.

**Note:** Everything that can be done with a `for` loop can also be done with a `while` loop.

**Arguments**

- `s_loopVar`: Name of the local loop variable that must not be changed inside the loop.
- `x_initialValue`: Integer expression setting the initial value for the local loop variable.
- `x_finalValue`: Integer expression giving final limit value for the loop.
- `g_expr1`: Expression to evaluate inside loop.
- `g_expr2 ...`: Additional expressions to evaluate inside loop.

**Value Returned**

- `t`: This construct always returns `t`. 
Examples

```plaintext
sum = 0
for( i 1 10
    sum = sum + i
    printf( "%d" sum ))
=> t

Prints 10 numbers and returns t.

sum = 0
for( i 1 5
    sum = sum + i
    println( sum )
)
=> t

Prints the value of sum with a carriage return for each pass through the loop:

1
3
6
10
15
```
foreach

foreach( s_formalVar g_exprList g_expr1 [g_expr2 ...] ) => l_valueList
  foreach( (s_formalVar1...s_formalVarN) g_exprList1...
    g_exprListN g_expr1 [g_expr2 ...] ) => l_valueList
  foreach( s_formalVar g_exprTable g_expr1 [g_expr2 ...] ) => o_valueTable

Description

Evaluates one or more expressions for each element of a list of values.

The first syntax form,

foreach( s_formalVar g_exprList g_expr1 [g_expr2 ...] ) => l_valueList

evaluates g_exprList, which returns a list l_valueList. It then assigns the first element from l_valueList to the formal variable s_formalVar and processes the expressions g_expr1 g_expr2 ... in sequence. The function then assigns the second element from l_valueList and repeats the process until l_valueList is exhausted.

The second syntax form,

foreach( (s_formalVar1...s_formalVarN) g_exprList1... g_exprListN
g_expr1 [g_expr2 ...] ) => l_valueList

can iterate over multiple lists to perform vector operations. Instead of a single formal variable, the first argument is a list of formal variables followed by a corresponding number of expressions for value lists and the expressions to be evaluated.

The third syntax form,

foreach( s_formalVar g_exprTable g_expr1 [g_expr2 ...] ) => o_valueTable

can be used to process the elements of an association table. In this case, s_formalVar is assigned each key of the association table one by one, and the body expressions are evaluated each iteration. The syntax for association table processing is provided in this syntax statement.

Arguments

s_formalVar  Name of the variable.
**OCEAN Reference**

**Language Constructs**

---

**g_exprList**
Expression whose value is a list of elements to assign to the formal variable \(s_{formalVar}\).

**g_expr1 g_expr2 ...**
Expressions to execute.

**g_exprTable**
Association table whose elements are to be processed.

---

**Value Returned**

**l_valueList**
Returns the value of the second argument, \(g\_exprList\).

**o_valueTable**
Returns the value of \(g\_exprTable\).

---

**Examples**

```plaintext
foreach( x '( 1 2 3 4 ) println( x ) )
```

1
2
3
4

=> ( 1 2 3 4 )

Prints the numbers 1 through 4 and returns the second argument to `foreach`.

```plaintext
foreach( key myTable printf( "%L : %L" key myTable[key] ) )
```

Accesses an association table and prints each key and its associated data.

```plaintext
( foreach ( x y ) '( 1 2 3 ) '( 4 5 6 ) ( println x+y ) )
```

5
7
9

=> ( 1 2 3 )

Uses `foreach` with more than one loop variable.

---

**Errors and Warnings**

The error messages from `foreach` might at times appear cryptic because some `foreach` forms get expanded to call the mapping functions `mapc`, `mapcar`, `mapcan`, and so forth.
**while**

\[
\text{while( } g\_condition \ g\_expr1 \ ... \ ) \Rightarrow t
\]

**Description**

Repeatedly evaluates \( g\_condition \) and the sequence of expressions \( g\_expr1 \ ... \) if the condition is true.

This process is repeated until \( g\_condition \) evaluates to false (\( \text{nil} \)). Note that because this form always returns \( t \), it is principally used for its side effects.

**Note:** Everything that can be done with a \textit{for} loop can also be done with a \textit{while} loop.

**Arguments**

- \( g\_condition \) Any SKILL expression.
- \( g\_expr1 \) Any SKILL expression.

**Value Returned**

\( t \) Always returns \( t \).

**Example**

\[
i = 0
while( \ (i <= 10) \ printf("%d" \ i++) ) \Rightarrow t
\]

Prints the digits 0 through 10.
case

case( g_selectionExpr l_clause1 [l_clause2 ...] ) => g_result/nil

Description

Evaluates the selection expression, matches the resulting selector values sequentially against comparators defined in clauses, and runs the expressions in the matching clause.

Each l_clause is a list of the form (g_comparator g_expr1 [g_expr2...]), where a comparator is either an atom (that is, a scalar) of any data type or a list of atoms. Comparators are always treated as constants and are never evaluated. The g_selectionExpr expression is evaluated and the resulting selector value is matched sequentially against comparators defined in l_clause1 l_clause2.... A match occurs when either the selector is equal to the comparator or the selector is equal to one of the elements in the list given as the comparator. If a match is found, the expressions in that clause and that clause only (that is, the first match) are run. The value of case is then the value of the last expression evaluated (that is, the last expression in the clause selected). If there is no match, case returns nil.

The symbol $t$ has special meaning as a comparator: it matches anything. It is typically used in the last clause to serve as a default case when no match is found with other clauses.

Arguments

g_selectionExpr

An expression whose value is evaluated and tested for equality against the comparators in each clause. When a match is found, the rest of the clause is evaluated.

l_clause1

An expression whose first element is an atom or list of atoms to be compared against the value of g_selectionExpr. The remainder of the l_clause is evaluated if a match is found.

l_clause2...

Zero or more clauses of the same form as l_clause1.

Value Returned

g_result

Returns the value of the last expression evaluated in the matched clause.
nil

Returns **nil** if there is no match.

**Example**

cornersType = "min"

type = case( cornersType
    ("min" path("./min"))
    ("typ" path("./typ"))
    ("max" path("./max"))
    (t println("you have not chosen an appropriate corner")))

=> path is set to "./min"

Sets **path** to "./min."
cond

cond( l_clause1 ... ) => g_result/nil

Description

Examines conditional clauses from left to right until either a clause is satisfied or there are no more clauses remaining.

This command is useful when there is more than one test condition, but only the statements of one test are to be carried out. Each clause is of the form ( g_condition g_expr1... ). The cond function examines a clause by evaluating the condition associated with the clause. The clause is satisfied if g_condition evaluates to non-nil, in which case expressions in the rest of the clause are evaluated from left to right, and the value returned by the last expression in the clause is returned as the value of the cond form. If g_condition evaluates to nil, however, cond skips the rest of the clause and moves on to the next clause.

Arguments

l_clause1 Each clause must be of the form ( g_condition g_expr1... ). When g_condition evaluates to non-nil, all the succeeding expressions are evaluated.

Value Returned

g_result Returns the value of the last expression of the satisfied clause.

nil Returns nil if no clause is satisfied.

Example

procedure( test(x)
    cond(((null x) (println "Arg is null"))
    ((numberp x) (println "Arg is a number"))
    ((stringp x) (println "Arg is a string"))
    (t (println "Arg is an unknown type")))
)

test( nil ) => nil; Prints "Arg is null".
test( 5 ) => nil; Prints "Arg is a number".
test( 'sym ) => nil; Prints "Arg is an unknown type".
Tests each of the arguments according to the conditions specified with $\text{cond}$. 
File Commands and Functions

This chapter contains information on the following commands:

- `close` on page 402
- `fscanf` on page 403
- `gets` on page 405
- `infile` on page 406
- `load` on page 407
- `newline` on page 409
- `outfile` on page 410
- `printf` on page 412
- `println` on page 413
close

close( p_port ) => t

Description

Drains, closes, and frees a port.

When a file is closed, it frees the FILE* associated with p_port. Do not use this function on piport, stdin, poport, stdout, or stderr.

Arguments

p_port Name of port to close.

Value Returned

t The port closed successfully.

Example

p = outfile( "~/test/myFile" ) => port:"~/test/myFile"
close( p ) => t

Drains, closes, and frees the /test/myFile port.
**fscanf**

\[ fscanf( p\_inputPort \ t\_formatString \ [s\_var1 \ldots] ) \Rightarrow x\_items/nil \]

**Description**

Reads input from a port according to format specifications and returns the number of items read in.

The results are stored into corresponding variables in the call. The `fscanf` function can be considered the inverse function of the `fprintf` output function. The `fscanf` function returns the number of input items it successfully matched with its format string. It returns `nil` if it encounters an end of file.

The maximum size of any input string being read as a string variable for `fscanf` is currently limited to 8 K. Also, the function `lineread` is a faster alternative to `fscanf` for reading Cadence® SKILL objects.

The common input formats accepted by `fscanf` are summarized below.

**Common Input Format Specifications**

<table>
<thead>
<tr>
<th>Format Specification</th>
<th>Types of Argument</th>
<th>Scans for</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>fixnum</td>
<td>An integer</td>
</tr>
<tr>
<td>%f</td>
<td>flonum</td>
<td>A floating-point number</td>
</tr>
<tr>
<td>%s</td>
<td>string</td>
<td>A string (delimited by spaces) in the input</td>
</tr>
</tbody>
</table>

**Arguments**

- `p\_inputPort` Input port to read from.
- `t\_formatString` Format string to match against in the reading.
- `s\_var1\ldots` Name of the variable in which to store results.
Value Returned

\( x_{\text{items}} \)

Returns the number of input items it successfully read in. As a side effect, the items read in are assigned to the corresponding variables specified in the call.

\( \text{nil} \)

Returns \( \text{nil} \) if an end of file is encountered.

Example

\[
\text{fscanf( p "\%d \%f" i d )}
\]

Scans for an integer and a floating-point number from the input port \( p \) and stores the values read in the variables \( i \) and \( d \), respectively.

Assume a file \texttt{testcase} with one line:

\begin{verbatim}
hello 2 3 world
\end{verbatim}

\[
\text{x = infile("testcase") => port:"testcase"
}
\text{fscanf( x "\%s \%d \%d \%s" a b c d ) => 4
}
\text{(list a b c d) => ("hello" 2 3 "world")
}
gets

`gets( s_variableName [p_inputPort] ) => t_string/nil`

**Description**

Reads a line from the input port and stores the line as a string in the variable. This is a macro.

The string is also returned as the value of `gets`. The terminating newline character of the line becomes the last character in the string.

**Arguments**

- `s_variableName` Variable in which to store the input string.
- `p_inputPort` Name of input port.
  Default value: `piport`

**Value Returned**

- `t_string` Returns the input string when successful.
- `nil` Returns `nil` when the end of file is reached.
  `(s_variableName maintains its last value.)`

**Example**

Assume the `test1.data` file has the following first two lines:

```
#This is the data for test1
0001 1100 1011 0111
```

```
p = infile("test1.data") => port:"test1.data"
gets(s p) => "#This is the data for test1"
gets(s p) => "0001 1100 1011 0111"
gets(s p) => "0001 1100 1011 0111"
s => "0001 1100 1011 0111"
```

Gets a line from the `test1.data` file and stores it in the variable `s`. The `s` variable contains the last string stored in it by the `gets` function.
infile

infile( S_fileName ) => p_inport/nil

Description

Opens an input port ready to read a file.

Always remember to close the port when you are done. The file name can be specified with either an absolute path or a relative path. In the latter case, the current SKILL path is used if it is not nil.

Arguments

S_fileName Name of the file to be read; it can be either a string or a symbol.

Value Returned

p_inport Returns the port opened for reading the named file.

nil Returns nil if the file does not exist or cannot be opened for reading.

Examples

in = infile( "~/test/input.il" ) => port:"~/test/input.il"
close( in ) => t

Closes the /test/input.il port.

Opens the input port /test/input.il.

infile("myFile") => nil

Returns nil if myFile does not exist according to the current setting of the SKILL path or exists but is not readable.
load

load( t_fileName [t_password]) => t

Description

Opens a file and repeatedly calls lineread to read in the file, immediately evaluating each form after it is read in.

This function uses the file extension to determine the language mode (.il for SKILL, .ils for SKILL++, and .ocn for a file containing OCEAN commands) for processing the language expressions contained in the file. For a SKILL++ file, the loaded code is always evaluated in the top-level environment.

load closes the file when the end of file is reached. Unless errors are discovered, the file is read in quietly. If load is interrupted by pressing Control-c, the function skips the rest of the file being loaded.

SKILL has an autoload feature that allows applications to load functions into SKILL on demand. If a function being run is undefined, SKILL checks to see if the name of the function (a symbol) has a property called autoload attached to it. If the property exists, its value, which must be either a string or an expression that evaluates to a string, is used as the name of a file to be loaded. The file should contain a definition for the function that triggered the autoload. Processing proceeds normally after the function is defined.

Arguments

\textit{t\_fileName} \quad \text{File to be loaded. Uses the file name extension to determine the language mode to use.} \quad \text{Valid values:}

- .ils \quad \text{Means the file contains SKILL++ code.}
- .il \quad \text{Means the file contains SKILL code.}
- .ocn \quad \text{Means the file contains OCEAN commands (with SKILL or SKILL++ commands)}

\textit{t\_password} \quad \text{Password, if t\_fileName is an encrypted file.}

Value Returned

\textit{t} \quad \text{Returns } t \text{ if the file is successfully loaded.
Example

load( "test.ocn" )

Loads the test.ocn file.

procedure( trLoadSystem()
    load( "test.il" );;; SKILL code
    load( "test.ils" );;; SKILL++ code
) ; procedure

You might have an application partitioned into two files. Assume that test.il contains SKILL code and test.ils contains SKILL/SKILL++ code. This example loads both files.
newline

newline( [p_outputPort] ) => nil

Description
Prints a newline (backslash n) character and then flushes the output port.

Arguments

p_outputPort  Output port.
          Defaults value: poport

Value Returned

nil         Prints a newline and then returns nil.

Example

print( "Hello" ) newline() print( "World!" )
"Hello"
"World!"
=> nil

Prints a newline character after Hello.
outfile

outfile( S_fileName [t_mode] ) => p_outport/nil

Description

Opens an output port ready to write to a file.

Various print commands can write to this file. Commands write first to a character buffer, which writes to the file when the character buffer is full. If the character buffer is not full, the contents are not written to the file until the output port is closed or the drain command is entered. Use the close or drain command to write the contents of the character buffer to the file. The file can be specified with either an absolute path or a relative path. If a relative path is given and the current SKILL path setting is not nil, all directory paths from SKILL path are checked in order, for that file. If found, the system overwrites the first updatable file in the list. If no updatable file is found, it places a new file of that name in the first writable directory.

Arguments

S_fileName Name of the file to open or create.

t_mode Mode in which to open the file. If a, the file is opened in append mode; If w, a new file is created for writing (any existing file is overwritten).

Default value: w

Value Returned

p_outport An output port ready to write to the specified file.

nil returns nil if the named file cannot be opened for writing. An error is signaled if an illegal mode string is supplied.

Examples

p = outfile( "/tmp/out.il" "w" ) => port:"/tmp/out.il"

Opens the /tmp/out.il port.

outfile( "/bin/ls" ) => nil
Returns \texttt{nil}, indicating that the specified port could not be opened.
printf

printf( t_formatString [g_arg1 ...] ) => t

Description

Writes formatted output to poport, which is the standard output port.

The optional arguments following the format string are printed according to their corresponding format specifications. Refer to the "Common Output Format Specifications" table for fprintf in the SKILL Language User Guide.

printf is identical to fprintf except that it does not take the p_port argument and the output is written to poport.

Arguments

\[ t\text{\_formatString} \]

Characters to be printed verbatim, intermixed with format specifications prefixed by the "\%" sign.

\[ g\text{\_arg1} \ldots \]

Arguments following the format string are printed according to their corresponding format specifications.

Value Returned

\[ t \]

Prints the formatted output and returns \( t \).

Example

\[ x = 197.9687 \Rightarrow 197.9687 \]

printf( "The test measures %10.2f." x )

Prints the following line to poport and returns \( t \).

The test measures 197.97. => t
println
println( g_value [p_outputPort] ) => nil

Description
Prints a SKILL object using the default format for the data type of the value, and then prints a newline character.

A newline character is automatically printed after printing g_value. The println function flushes the output port after printing each newline character.

Arguments

g_value Any SKILL value.
p_outputPort Port to be used for output.

Default value: poport

Value Returned

nil Prints the given object and returns nil.

Example

for( i 1 3 println( "hello" ))
"hello"
"hello"
"hello"
=> t

Prints hello three times. for always returns t.
OCEAN 4.4.3 Issues

For the 4.4.3 release of OCEAN, there are some restrictions and requirements.

The netlist file that you specify for the Affirma™ Spectre® analog circuit simulator interface with the design command must be netlist. The full path can be specified. For example, `/usr/netlist` is acceptable. The netlistHeader and netlistFooter files are searched in the same directory where the netlist is located. Cadence recommends that you use the netlist generated from the Affirma analog circuit design environment. Netlists from other sources can also be used, as long as they contain only connectivity. You might be required to make slight modifications.

- Cadence recommends full paths for the Spectre simulator model files, definition files, and stimulus files.
- The Affirma Cadence SPICE circuit simulator is still used to parse netlists for socket interfaces (spectreS and cdsSpice, for example). Therefore, the netlist that you specify with the design command must be in Cadence SPICE syntax. Cadence recommends that you use the raw netlist generated from the Affirma analog circuit design environment. Netlists from other sources can also be used, as long as they can pass through Cadence SPICE. You might be required to make slight modifications.
- Any presimulation commands that you specify are appended to the final netlist (as is currently the case in the design environment). Therefore, if you have control cards already in your netlist, and specify simulation setup commands, you might duplicate control cards, which causes a warning or an error from the simulator. You might want to remove control cards from your netlist file to avoid the warnings.
- Models, include files, stimulus files, and PWLF files must be found according to the path specified with the path command.

Mixed-Signal in OCEAN 4.4.3

All of the analog OCEAN features are available in mixed-signal. This means you can set up analyses, change options, change the path, and so forth.
For the 4.4.2 release, there are limits in the area of digital simulation. You must create the
digital final netlist in the Affirma analog circuit design environment before you can work in
OCEAN.

In the 4.4.2 release, there are no commands that operate on Verilog-XL final netlists. If you
need to change anything in the final netlist, you must make those changes by hand.

However, you can change any of the command line arguments that are sent to the Verilog-XL
simulator. This means you can change any of the digital options or any of the mixed-signal
options. To see what these options are, choose Simulation – Options – Digital in the
Affirma Analog Circuit Design Environment window.

For example, you can change acceleration, keep nodes, and library files.

For detailed information, please refer to the Affirma Mixed-Signal Circuit Design
Environment User Guide.
Index

Symbols

&& (and) operator  44
,... in syntax  16
... in syntax  16
/ in syntax  16
[] in syntax  16
{} in syntax  16
| in syntax  16
|| (or) operator  45

Numerics

1  17
2  18

A

abs  187
abs function  187
acos  188
add1  189
addSubwindow  151
addSubWindowTitle  152
addSubwindowTitle  152
addTitle  153
addWaveLabel  154
addWindowLabel  156
aliases  179
Allocating an Array of a Given Size  56
alphalessp function  57
alphaNumCmp function  58
analysis  70
Appending a maximum number of characters from two input strings
(buildString)  56
appendPath  72
arithmetic operators  42
predefined functions  185
Arithmetic and Logical Expressions  49
Arithmetic Operators  42
Arrays  56
arrays
declaring  56
definition  56
asin  190
atan  191
Atoms  55
average  207

B

b1f  209
bandwidth  210
binary minus operator  47
Blocking and Nonblocking Modes  29
Blocking Mode  29
braces in syntax  16
brackets in syntax  16
buildString function  56

C

C language comparison
escape characters  56
parentheses  48
strings  55
case  397
case statement  397
clearAll  157
clearSubwindow  158
clip  211
clip function  211
close  402
close function  402
command types  20
commands
corner
cornerDesVar  342
cornerMeas  343
cornerRun  344
cornerRunTemp  346
residual  347
selectProcess  348
data access
dataTypes  113
**getData** 114
**i** 116
**noiseSummary** 118
**ocnHelp** 121
**ocnPrint** 123
**openResults** 28, 125
**outputParams** 127
**outputs** 128
**pv** 130
**report** 132
**results** 137
**selectResult** 138
**sweepNames** 141
**sweepValues** 142
**v** 143
**plotting**
**addSubwindow** 151
**addSubwindowTitle** 152
**addTitle** 153
**addWaveLabel** 154
**addWindowLabel** 156
**clearAll** 157
**clearSubwindow** 158
**currentSubwindow** 159
**currentWindow** 160
**dbCompressionPlot** 161
**deleteSubwindow** 162
**deleteWaveform** 163
**displayMode** 164
**graphicsOff** 165
**graphicsOn** 166
**hardCopy** 167
**hardCopyOptions** 168
**ip3Plot** 170
**newWindow** 171
**plot** 172
**plotStyle** 174
**removeLabel** 175
**xLimit** 176
**yLimit** 177
**return values** 25
**simulation**
**ac** 69
**analysis** 70
**appendPath** 72
**createFinalNetlist** 73
**dc** 75
**delete** 78
**design** 80
**desVar** 82
**envOption** 83
**forcenode** 84
**ic** 85
**includeFile** 86
**monteCarlo** 355
**monteExpr** 359
**monteRun** 362
**nodeset** 88
**noise** 89
**ocnDisplay** 90
**off** 92
**optimizeAlgoControl** 369
**optimizeGoal** 370
**optimizePlotOption** 371
**optimizeRun** 373
**optimizeVar** 374
**option** 93
**paramAnalysis** 27, 336
**paramRun** 340
**path** 94
**prependPath** 95
**restore** 96
**resultsDir** 97
**run** 98
**save** 101
**simulator** 105
**store** 107
**temp** 108
**tran** 109
**commenting code** 47
**Comments** 47
**Common SKILL Syntax Characters Used In**
**OCEAN** 21
**Comparing Strings** 57
**Comparing Two String or Symbol Names**
  **Alphanumerically or Numerically**
  (**alphaNumCmp**) 58
**Comparing Two Strings Alphabetically**
  (**strcmp**) 58
**Comparing Two Strings or Symbol Names**
  **Alphabetically (alphalessp)** 57
**complex** 220
**complexp** 221
**concatenating a list of strings with**
  **separation characters**
  (**buildString**) 56
**Concatenating Strings (Lists)** 56
**Concatenating two or more input strings**
  (**strcat**) 57
### SKILL

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>187</td>
</tr>
<tr>
<td>acos</td>
<td>188</td>
</tr>
<tr>
<td>add</td>
<td>189</td>
</tr>
<tr>
<td>asin</td>
<td>190</td>
</tr>
<tr>
<td>atan</td>
<td>191</td>
</tr>
<tr>
<td>cos</td>
<td>192</td>
</tr>
<tr>
<td>exp</td>
<td>193</td>
</tr>
<tr>
<td>max</td>
<td>197</td>
</tr>
<tr>
<td>min</td>
<td>198</td>
</tr>
<tr>
<td>mod</td>
<td>199</td>
</tr>
<tr>
<td>phaseNoise</td>
<td>129</td>
</tr>
<tr>
<td>random</td>
<td>200</td>
</tr>
<tr>
<td>resultParam</td>
<td>135</td>
</tr>
<tr>
<td>round</td>
<td>201</td>
</tr>
<tr>
<td>sin</td>
<td>202</td>
</tr>
<tr>
<td>sp</td>
<td>140</td>
</tr>
<tr>
<td>sqrt</td>
<td>203</td>
</tr>
<tr>
<td>srandom</td>
<td>204</td>
</tr>
<tr>
<td>sub1</td>
<td>205</td>
</tr>
<tr>
<td>tan</td>
<td>206</td>
</tr>
<tr>
<td>vswr</td>
<td>145</td>
</tr>
<tr>
<td>zm</td>
<td>146</td>
</tr>
<tr>
<td>zref</td>
<td>147</td>
</tr>
</tbody>
</table>

### waveform

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>207</td>
</tr>
<tr>
<td>b1f</td>
<td>209</td>
</tr>
<tr>
<td>bandwidth</td>
<td>210</td>
</tr>
<tr>
<td>clip</td>
<td>211</td>
</tr>
<tr>
<td>compression</td>
<td>213</td>
</tr>
<tr>
<td>conjugate</td>
<td>219</td>
</tr>
<tr>
<td>convolve</td>
<td>222</td>
</tr>
<tr>
<td>cross</td>
<td>228</td>
</tr>
<tr>
<td>db10</td>
<td>230</td>
</tr>
<tr>
<td>db20</td>
<td>231</td>
</tr>
<tr>
<td>dbm</td>
<td>232</td>
</tr>
<tr>
<td>delay</td>
<td>233</td>
</tr>
<tr>
<td>deriv</td>
<td>236</td>
</tr>
<tr>
<td>dft</td>
<td>237</td>
</tr>
<tr>
<td>flip</td>
<td>239</td>
</tr>
<tr>
<td>fourEval</td>
<td>240</td>
</tr>
<tr>
<td>frequency</td>
<td>241</td>
</tr>
</tbody>
</table>

### Miscellaneous

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>fscanf</td>
<td>402</td>
</tr>
<tr>
<td>gets</td>
<td>405</td>
</tr>
<tr>
<td>inline</td>
<td>406</td>
</tr>
<tr>
<td>load</td>
<td>407</td>
</tr>
<tr>
<td>newline</td>
<td>409</td>
</tr>
<tr>
<td>outfile</td>
<td>410</td>
</tr>
<tr>
<td>printf</td>
<td>412</td>
</tr>
<tr>
<td>printn</td>
<td>413</td>
</tr>
<tr>
<td>ga</td>
<td>242</td>
</tr>
<tr>
<td>gac</td>
<td>243</td>
</tr>
<tr>
<td>gainBwProd</td>
<td>245</td>
</tr>
<tr>
<td>gainMargin</td>
<td>247</td>
</tr>
<tr>
<td>gmax</td>
<td>248</td>
</tr>
<tr>
<td>gmin</td>
<td>249</td>
</tr>
<tr>
<td>gmux</td>
<td>251</td>
</tr>
<tr>
<td>gpc</td>
<td>253</td>
</tr>
<tr>
<td>groupDelay</td>
<td>255</td>
</tr>
<tr>
<td>gsmg</td>
<td>250</td>
</tr>
<tr>
<td>Harmonic</td>
<td>257</td>
</tr>
<tr>
<td>harmonicList</td>
<td>261</td>
</tr>
<tr>
<td>int</td>
<td>263</td>
</tr>
<tr>
<td>imag</td>
<td>264</td>
</tr>
<tr>
<td>integ</td>
<td>265</td>
</tr>
<tr>
<td>ipn</td>
<td>266</td>
</tr>
<tr>
<td>kf</td>
<td>275</td>
</tr>
<tr>
<td>ln</td>
<td>276</td>
</tr>
<tr>
<td>log10</td>
<td>277</td>
</tr>
<tr>
<td>lsb</td>
<td>278</td>
</tr>
<tr>
<td>lshift</td>
<td>280</td>
</tr>
<tr>
<td>mag</td>
<td>281</td>
</tr>
<tr>
<td>nc</td>
<td>282</td>
</tr>
<tr>
<td>overshoot</td>
<td>284</td>
</tr>
<tr>
<td>peakToPeak</td>
<td>286</td>
</tr>
<tr>
<td>phase</td>
<td>287</td>
</tr>
<tr>
<td>phaseDeg</td>
<td>289</td>
</tr>
<tr>
<td>phaseDegUnwrapped</td>
<td>290</td>
</tr>
<tr>
<td>phaseMargin</td>
<td>291</td>
</tr>
<tr>
<td>phaseRad</td>
<td>293</td>
</tr>
<tr>
<td>phaseRadUnwrapped</td>
<td>294</td>
</tr>
<tr>
<td>pow</td>
<td>295</td>
</tr>
<tr>
<td>psd</td>
<td>297</td>
</tr>
<tr>
<td>psdbb</td>
<td>301</td>
</tr>
<tr>
<td>real</td>
<td>305</td>
</tr>
<tr>
<td>riseTime</td>
<td>306</td>
</tr>
<tr>
<td>rms</td>
<td>308</td>
</tr>
<tr>
<td>rmsNoise</td>
<td>309</td>
</tr>
<tr>
<td>root</td>
<td>310</td>
</tr>
<tr>
<td>rshift</td>
<td>312</td>
</tr>
<tr>
<td>sample</td>
<td>313</td>
</tr>
<tr>
<td>settingTime</td>
<td>315</td>
</tr>
<tr>
<td>slewRate</td>
<td>317</td>
</tr>
<tr>
<td>spectralPower</td>
<td>319</td>
</tr>
<tr>
<td>ssb</td>
<td>320</td>
</tr>
<tr>
<td>tangent</td>
<td>322</td>
</tr>
<tr>
<td>thd</td>
<td>323</td>
</tr>
<tr>
<td>value</td>
<td>325</td>
</tr>
<tr>
<td>xmax</td>
<td>327</td>
</tr>
<tr>
<td>xmin</td>
<td>329</td>
</tr>
</tbody>
</table>
Interactive Session Demonstrating the OCEAN Use Model 33

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xval</td>
<td>331</td>
</tr>
<tr>
<td>ymax</td>
<td>332</td>
</tr>
<tr>
<td>ymin</td>
<td>333</td>
</tr>
<tr>
<td>ga</td>
<td>242</td>
</tr>
<tr>
<td>gac</td>
<td>243</td>
</tr>
<tr>
<td>gainBwProd</td>
<td>245</td>
</tr>
<tr>
<td>gainMargin</td>
<td>247</td>
</tr>
<tr>
<td>getData</td>
<td>114</td>
</tr>
<tr>
<td>gets</td>
<td>405</td>
</tr>
<tr>
<td>gmax</td>
<td>248</td>
</tr>
<tr>
<td>gmin</td>
<td>249</td>
</tr>
<tr>
<td>gmsg</td>
<td>250</td>
</tr>
<tr>
<td>gmux</td>
<td>251</td>
</tr>
<tr>
<td>gp</td>
<td>252</td>
</tr>
<tr>
<td>gpc</td>
<td>253</td>
</tr>
<tr>
<td>graphicsOff</td>
<td>165</td>
</tr>
<tr>
<td>graphicsOn</td>
<td>166</td>
</tr>
<tr>
<td>groupDelay</td>
<td>255</td>
</tr>
<tr>
<td>gt</td>
<td>256</td>
</tr>
<tr>
<td>hardCopy</td>
<td>167</td>
</tr>
<tr>
<td>hardCopyOptions</td>
<td>168</td>
</tr>
<tr>
<td>harmonic</td>
<td>257</td>
</tr>
<tr>
<td>harmonicFreqList</td>
<td>259</td>
</tr>
<tr>
<td>harmonicList</td>
<td>261</td>
</tr>
<tr>
<td>help</td>
<td></td>
</tr>
<tr>
<td>command examples</td>
<td>20</td>
</tr>
<tr>
<td>online</td>
<td>20</td>
</tr>
<tr>
<td>histogram</td>
<td>352</td>
</tr>
<tr>
<td>hostMode</td>
<td>379</td>
</tr>
<tr>
<td>hostmode</td>
<td>379</td>
</tr>
<tr>
<td>i</td>
<td>116</td>
</tr>
<tr>
<td>i</td>
<td>85</td>
</tr>
<tr>
<td>ic</td>
<td>85</td>
</tr>
<tr>
<td>if</td>
<td>388</td>
</tr>
<tr>
<td>if statement</td>
<td>388</td>
</tr>
<tr>
<td>iim alias</td>
<td>180</td>
</tr>
<tr>
<td>iinteg</td>
<td>263</td>
</tr>
<tr>
<td>im alias</td>
<td>180</td>
</tr>
<tr>
<td>imag</td>
<td>264</td>
</tr>
<tr>
<td>includeFile</td>
<td>86</td>
</tr>
<tr>
<td>infile</td>
<td>406</td>
</tr>
<tr>
<td>infix arithmetic operators</td>
<td>46</td>
</tr>
<tr>
<td>infix operators</td>
<td>48, 51</td>
</tr>
<tr>
<td>input lines</td>
<td>49</td>
</tr>
<tr>
<td>integ</td>
<td>265</td>
</tr>
<tr>
<td>integer</td>
<td>53</td>
</tr>
<tr>
<td>italics in syntax</td>
<td>16</td>
</tr>
<tr>
<td>iterVsValue</td>
<td>354</td>
</tr>
<tr>
<td>keywords</td>
<td>16</td>
</tr>
<tr>
<td>kf</td>
<td>275</td>
</tr>
<tr>
<td>killJob</td>
<td>381</td>
</tr>
<tr>
<td>let</td>
<td>59</td>
</tr>
<tr>
<td>Line Continuation</td>
<td>49</td>
</tr>
<tr>
<td>linRg</td>
<td>194</td>
</tr>
<tr>
<td>literal characters</td>
<td>15</td>
</tr>
<tr>
<td>ln</td>
<td>276</td>
</tr>
<tr>
<td>load</td>
<td>407</td>
</tr>
<tr>
<td>Loading OCEAN Scripts</td>
<td>37</td>
</tr>
<tr>
<td>local variables</td>
<td>59</td>
</tr>
<tr>
<td>log</td>
<td>195</td>
</tr>
<tr>
<td>log10</td>
<td>277</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>44</td>
</tr>
<tr>
<td>logical operators</td>
<td>44</td>
</tr>
<tr>
<td>logRg</td>
<td>196</td>
</tr>
<tr>
<td>lsb</td>
<td>278</td>
</tr>
<tr>
<td>lshift</td>
<td>280</td>
</tr>
<tr>
<td>mag</td>
<td>281</td>
</tr>
<tr>
<td>max</td>
<td>197</td>
</tr>
<tr>
<td>min</td>
<td>198</td>
</tr>
<tr>
<td>Mixed-Signal in OCEAN 4.4.3</td>
<td>415</td>
</tr>
<tr>
<td>mod</td>
<td>199</td>
</tr>
</tbody>
</table>
OCEAN Reference

printf 412
println 413
procedure 60
procedures, definition 60
See also SKILL functions
psd 297
psdb 301
pv 130

Q
Question Mark 24
question mark 24

R
random 200
real 305
Recovering from an Omitted Double Quote 22
Related Documents 15
Relational and Logical Operators 43
Relational Operators 44
relational operators 44
removeLabel 175
report 132
residual 347
restore 96
resultParam 135
results 137
resultsDir 97
resumeJob 384
return value (=>) 60
return values 25
right arrow in syntax 16
riseTime 306
rms 308
rmsNoise 309
Role of Parentheses 48
root 310
round 201
rshift 312
run 98
Running Multiple Simulators 38

S
sample 313
save 101
saveOption 103
Scaling Factors 42
scaling factors 42
scatterplot 365
Selectively Creating Scripts 34
selectProcess 348
selectResult 138
settingTime 315
settlingTime 315
setup 64
simulation commands
See commands, simulation
simulator 105
sin 202
Single Quotes 23
single quotes 23
SKILL
commenting code 47
primitives 49
white space in code 47
SKILL data types 24
Skill Function Return Values 60
SKILL functions
arguments 60
declaring 58
defining parameters 59
definition 60
parameters 60
terminology 60
Skill Functions 53
SKILL functions, syntax conventions 17
SKILL Syntax 46
SKILL syntax 21
SKILL Syntax Examples 17
slash in syntax 16
slewRate 317
sp 140
Special Characters 46
specLimits 366
spectralPower 319
sqrt 203
srandom 204
ssb 320
stimulusFile 106
store 107
strcat function 57
strcmp function 58
Strings 55
strings
comparing 57
Z

zm  146
zref  147