

# **Spectre® Circuit Simulator Measurement Description Language User Guide and Reference**

**Product Version 17.1  
November 2017**

© 2003–2018 Cadence Design Systems, Inc. All rights reserved.

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

MMSIM contains technology licensed from, and copyrighted by: C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh © 1979, J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson © 1988, J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling © 1990; University of Tennessee, Knoxville, TN and Oak Ridge National Laboratory, Oak Ridge, TN © 1992-1996; Brian Paul © 1999-2003; M. G. Johnson, Brisbane, Queensland, Australia © 1994; Kenneth S. Kundert and the University of California, 1111 Franklin St., Oakland, CA 94607-5200 © 1985-1988; Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, CA 94304-1185 USA © 1994, Silicon Graphics Computer Systems, Inc., 1140 E. Arques Ave., Sunnyvale, CA 94085 © 1996-1997, Moscow Center for SPARC Technology, Moscow, Russia © 1997; Regents of the University of California, 1111 Franklin St., Oakland, CA 94607-5200 © 1990-1994, Sun Microsystems, Inc., 4150 Network Circle Santa Clara, CA 95054 USA © 1994-2000, Scriptics Corporation, and other parties © 1998-1999; Aladdin Enterprises, 35 Eyal St., Kiryat Arye, Petach Tikva, Israel 49511 © 1999 and Jean-loup Gailly and Mark Adler © 1995-2005; RSA Security, Inc., 174 Middlesex Turnpike Bedford, MA 01730 © 2005.

All rights reserved. Associated third party license terms may be found at <install\_dir>/doc/OpenSource/\*

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders.

**Restricted Permission:** This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

**Disclaimer:** Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor

---



# Contents

---

<u>Preface</u> .....	11
<u>What MDL Does</u> .....	12
<u>The MDL Flow</u> .....	13
<u>The MDL Language</u> .....	14
<u>Related Documents</u> .....	14
<u>Typographic and Syntax Conventions</u> .....	14
 <u>1</u>	
<u>Defining and Using Measurement Aliases</u> .....	17
<u>Defining a Measurement Alias</u> .....	18
<u>Using a Measurement Alias</u> .....	19
<u>Defining Measurement Aliases on the Fly</u> .....	21
<u>Propagating Variables</u> .....	21
<u>Defining a Macro</u> .....	22
<u>Accessing Netlist or Model Parameters</u> .....	22
<u>Accessing Model Names and Types</u> .....	23
<u>Accessing Noise Parameters</u> .....	23
<u>Using Named and Primitive Analyses</u> .....	24
<u>Looping Statements</u> .....	24
<u>foreach Statement</u> .....	25
<u>search Statement</u> .....	30
<u>mvarsearch Statement</u> .....	33
<u>Include Statement</u> .....	37
<u>Evaluating Expressions Selectively</u> .....	38
<u>If/Else Statement</u> .....	38
<u>Ternary Expression Statement</u> .....	40
<u>Specifying the Output File Format</u> .....	41
<u>Autostop</u> .....	44
<u>Monte Carlo</u> .....	45
<u>Supported Spectre Circuit Simulator Analyses</u> .....	46
<u>Supported Spectre Circuit Simulator Formats</u> .....	50

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<u>Optimizations and Tips and Tricks</u> .....	50
<u>Data Output Optimizations</u> .....	50
<u>Performance Optimizations</u> .....	51
<u>MDL Reuse</u> .....	51
<u>Common Pitfalls</u> .....	52
<u>Miscellaneous</u> .....	52

## 2

### Constructing MDL Expressions .....

<u>Basic Language Elements and Scope Rules</u> .....	56
<u>White Space</u> .....	56
<u>Comments</u> .....	56
<u>Identifiers</u> .....	57
<u>Scope Rules</u> .....	57
<u>Data Types</u> .....	58
<u>Numbers</u> .....	58
<u>Enumeration Names</u> .....	60
<u>Predefined Constants</u> .....	60
<u>enum</u> .....	61
<u>Net</u> .....	62
<u>Terminal</u> .....	62
<u>Analysis</u> .....	62
<u>Array</u> .....	63
<u>Declarations</u> .....	66
<u>Operators</u> .....	69
<u>Overview of Operators</u> .....	69
<u>Unary Operators</u> .....	70
<u>Binary Operators</u> .....	70
<u>Operator Precedence</u> .....	71

## 3

### Running MDL in Batch Mode .....

<u>spectremdl</u> .....	74
<u>Syntax</u> .....	74
<u>Arguments</u> .....	74

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<u>Examples</u> .....	76
-----------------------	----

## 4

<u>Running MDL in Post-processing Mode</u> .....	79
--	----

<u>mdl</u> .....	80
<u>Syntax</u> .....	80
<u>Arguments</u> .....	80
<u>Limitations</u> .....	82

## A

<u>Built-In Functions</u> .....	85
---------------------------------	----

<u>abs</u> .....	86
<u>acos</u> .....	87
<u>acosh</u> .....	88
<u>analstop</u> .....	89
<u>angle</u> .....	90
<u>argmax</u> .....	91
<u>argmin</u> .....	93
<u>asin</u> .....	94
<u>asinh</u> .....	95
<u>atan</u> .....	96
<u>atanh</u> .....	97
<u>avg</u> .....	98
<u>avgdev</u> .....	99
<u>bw (bandwidth)</u> .....	100
<u>ceil</u> .....	103
<u>cfft</u> .....	104
<u>clip</u> .....	105
<u>conj</u> .....	107
<u>convolve</u> .....	108
<u>cos</u> .....	110
<u>cosh</u> .....	111
<u>cplx</u> .....	112
<u>cross</u> .....	113
<u>crosscorr</u> .....	115

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<u>crosses</u>	116
<u>d2r (degrees-to-radians)</u>	119
<u>db</u>	120
<u>db10</u>	121
<u>dbm</u>	122
<u>deltax</u>	123
<u>deltaxes</u>	127
<u>deriv</u>	128
<u>dutycycle</u>	129
<u>dutycycles</u>	130
<u>exp</u>	132
<u>falltime</u>	133
<u>fft</u>	136
<u>flip</u>	138
<u>floor</u>	140
<u>fmt</u>	141
<u>freq</u>	142
<u>freq_jitter</u>	144
<u>gainBwProd</u>	146
<u>gainmargin</u>	147
<u>getinfo</u>	148
<u>groupdelay</u>	149
<u>histo</u>	151
<u>l</u>	153
<u>ifft</u>	154
<u>iinteg</u>	156
<u>im</u>	158
<u>int</u>	159
<u>integ</u>	160
<u>ln</u>	161
<u>log10</u>	162
<u>mag</u>	163
<u>max</u>	164
<u>min</u>	165
<u>mod</u>	166
<u>movingavg</u>	167



## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<u>overshoot</u>	168
<u>period_jitter</u>	170
<u>ph</u>	171
<u>phasemargin</u>	172
<u>pow</u>	173
<u>pp (peak-to-peak)</u>	174
<u>pzbode</u>	175
<u>pzfilter</u>	177
<u>r2d (radians-to-degrees)</u>	180
<u>re</u>	181
<u>real</u>	182
<u>risetime</u>	183
<u>rmsnoise</u>	186
<u>rms (root-mean-square)</u>	187
<u>round</u>	188
<u>S</u>	189
<u>sample</u>	191
<u>settlingtime</u>	194
<u>sign</u>	196
<u>sin</u>	197
<u>sinh</u>	198
<u>size</u>	199
<u>slewrate</u>	201
<u>slice</u>	203
<u>snr</u>	204
<u>sqrt</u>	206
<u>stathisto</u>	207
<u>stddev</u>	209
<u>sum</u>	210
<u>system</u>	211
<u>tan</u>	212
<u>tanh</u>	213
<u>trim</u>	214
<u>V</u>	216
<u>variance</u>	217
<u>window</u>	218

**Spectre Circuit Simulator Measurement Description Language User Guide and Reference**

---

<u>xval</u> .....	227
<u>Y</u> .....	229
<u>yval</u> .....	230
<u>Z</u> .....	232

**B**

<u>SPICE Compatibility for Analyses</u> .....	233
---	-----

**C**

<u>SPICE Compatibility for options supported by MDL</u> .....	237
<u>Support the SPICE option .option co=&lt;number&gt;</u> .....	237
<u>Support equal interval output for .print</u> .....	238

## Preface

---

Spectre® Circuit Simulator Measurement Description Language (MDL) is a productivity-enhancing tool for simulation and data analysis. This user guide and reference describes MDL and explains how to make the best use of it.

This preface discusses the following:

- [What MDL Does](#) on page 12
- [Related Documents](#) on page 14
- [Typographic and Syntax Conventions](#) on page 14

## What MDL Does

MDL is a scripting language that you can use to control the Spectre® circuit simulator and the Virtuoso® Visualization and Analysis tool. With MDL, you can

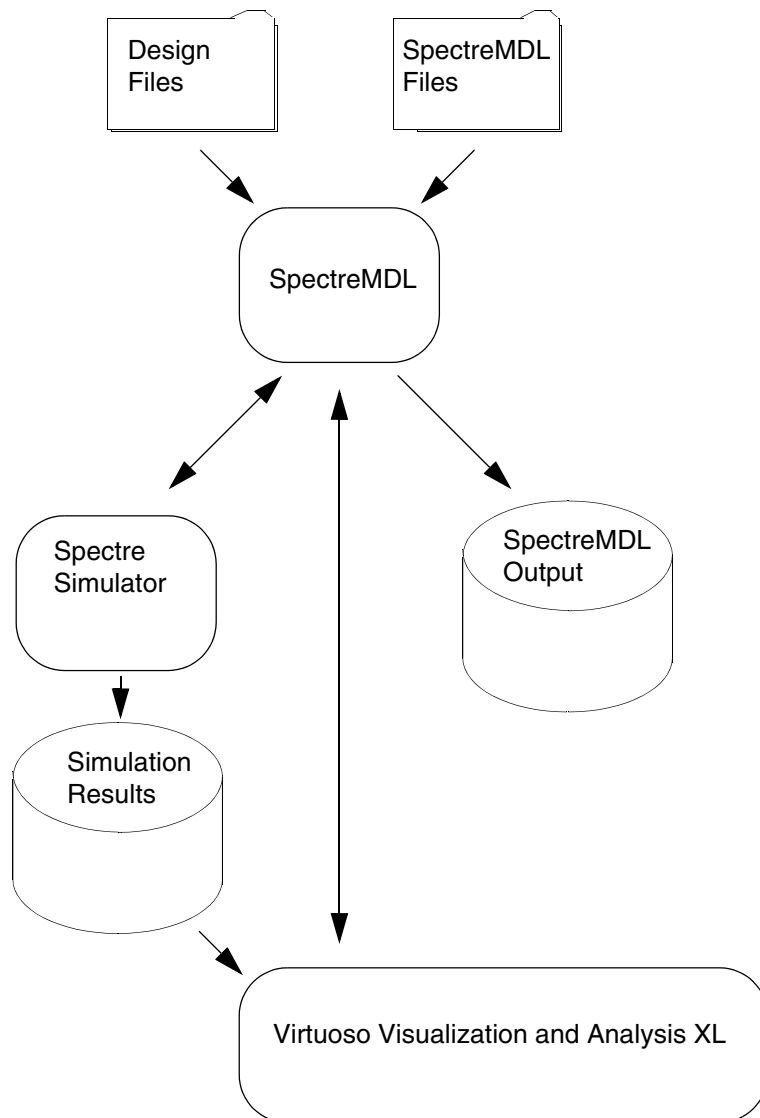
- Create measurement aliases that can be easily reused in different circuits. A measurement alias is a reusable, easily tailored procedure that includes a single analysis statement and a collection of one or more MDL expressions to be evaluated at runtime.
- Efficiently run simulations in batch mode.
- Parameterize measurement aliases, making them reusable over various applications.
- Use the wild card (\\*) in the MDL control file for all signals mapping.

**Note:** The wild card support has been added in the MMSIM 12.1 release.

With these features, MDL allows you to verify circuits easily and with confidence.

## The MDL Flow

As illustrated by the following figure, MDL interacts with a variety of inputs, outputs, and tools. Inputs consist of design files and files containing measurement blocks. Outputs include sets of simulation results, the values returned by MDL expressions, and log files. Interacting tools include Spectre, which simulates the design, and Virtuoso® Visualization and Analysis tool, which you can use to plot and post-process the results of the simulation.



## The MDL Language

Most of this document is devoted to describing the language used by the MDL tool. That language, like any language, has elements that must be used according to rules. You will find that although MDL is easily learned, the power it gives you to control simulations is great.

## Related Documents

For information about related products, consult the sources listed below.

- *Virtuoso® Analog Design Environment User Guide*
- *Cadence Analog Mixed-Signal Simulation Interface Option*
- *Spectre Circuit Simulator Reference*
- *Spectre Circuit Simulator and Accelerated Parallel Simulator User Guide*

## Typographic and Syntax Conventions

Special typographical conventions distinguish certain kinds of text in this document. The formal syntax used in this reference uses the definition operator, `:=`, to define the more complex elements of MDL in terms of less complex elements. However, for simplicity, the syntax for the user-compiled functions omits the definition operator.

- Lowercase words represent syntactic categories. For example,  
`identifier`
- Boldface words represent elements of the syntax that must be used exactly as presented. Such items include keywords, operators, and punctuation marks. For example,  
**`real`**
- Variables are set in italic font,  
*`allowed_errors`*
- Vertical bars indicate alternatives. You can choose to use any one of the items separated by the bars. For example,  

```
termID ::=
    designID
    |   unsignedInteger
```
- Square brackets enclose optional items. For example,

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
parameter_declaration_statement ::=  
    [ input ] real parameter [= expression ]
```

- Braces enclose an item that can be repeated zero or more times. For example,

```
unsigned_num ::=  
    decimal_digit { decimal_digit }
```

Code examples are set in constant-width font.

```
/* This is an example of the font used for code.*/
```

Keywords and filenames are set in constant-width font, like this: `keyword`, `file_name`.

# **Spectre Circuit Simulator Measurement Description Language User Guide and Reference**

---



---

# Defining and Using Measurement Aliases

---

A measurement alias is a Measurement Description Language (MDL) procedure that you can use to run an analysis and extract information about the performance of the circuit. For example, you might use a measurement alias to determine the bandwidth of an amplifier.

Measurement aliases provide a way for you to bind analyses to MDL expressions, creating procedures that can be called multiple times and parameterized for specific applications.

This chapter includes the following sections.

- [Defining a Measurement Alias](#) on page 18
- [Using a Measurement Alias](#) on page 19
- [Defining Measurement Aliases on the Fly](#) on page 21
- [Propagating Variables](#) on page 21
- [Accessing Netlist or Model Parameters](#) on page 22
- [Using Named and Primitive Analyses](#) on page 24
- [Looping Statements](#) on page 24
- [Include Statement](#) on page 37
- [Evaluating Expressions Selectively](#) on page 38
- [Specifying the Output File Format](#) on page 41
- [Autostop](#) on page 44
- [Monte Carlo](#) on page 45
- [Supported Spectre Circuit Simulator Analyses](#) on page 46
- [Supported Spectre Circuit Simulator Formats](#) on page 50
- [Optimizations and Tips and Tricks](#) on page 50

## Defining a Measurement Alias

Defining a measurement alias involves combining a call to an analysis with one or more MDL expressions into a reusable procedure. You can define or include an alias measurement only at the top level of an MDL control file. An alias measurement must be defined before it is used in a MDL run statement.

```
alias_measurement_statement ::=
    alias measurement measurement_name {
        {initialization_block}
        run analysis [ as othername]
        {export_block}
    }

analysis ::= =
    BuiltInAnalysis | PredefinedAnalysis | AnalysisVariable

BuiltInAnalysis ::= =
    dc | ac | tran | noise | info | sp
```

`alias measurement`

Keyword to define a measurement block. The block of MDL statements to be run when the measurement is called. The statements are executed in the defined order. A variable must be defined before it can be used.

`measurement_name`

The name of the measurement alias you are defining. **Note:** You can use special characters like hyphen (-), ampersand (&), caret (^), and so on in the measurement alias names, variable names, and analysis names. However, these special characters must be preceded by an escape symbol (\). For example,

```
alias measurement transim\ -montecarlo
```

`initialization_block`

The initialization block. Input variables can be defined only in this block, otherwise MDL ignores them and issues warning messages that they are ignored.

`run analysis`

The run statement can be used to call a built in Spectre analysis such as dc, ac or tran, to call an analysis defined in the circuit netlist , or to call an analysis variable.

`as othername`

The results dataset is named *othername* if the *as* option is used. If othername is not specified, the results dataset is given the *measurement\_name*.

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<i>export_block</i>	One or more MDL expressions that are evaluated as a result of the analysis. However, you cannot use the <code>search</code> or <code>foreach</code> commands in the <i>export_block</i> , nor can the <i>export_block</i> include a measurement alias.
<i>PredefinedAnalysis</i>	Name of the analysis defined in the netlist or MDL control file. For information on the analyses supported by MDL, see <a href="#">Supported Spectre Circuit Simulator Analyses</a> on page 46 and <a href="#">SPICE Compatibility for Analyses</a> on page 233.
<i>AnalysisVariable</i>	Name of a pre-defined analysis variable or an element of an array of analyses. For more information, see <a href="#">Analysis</a> on page 62.

For example, consider the following MDL control file.

```
alias measurement showmaxmin { // The measurement alias is defined here.
    run tran1(stop=1u)
    export real maxout=max(V(out))
    export real minout=min(V(out))
}
run showmaxmin // This statement runs the measurement.
```

This control file first defines a measurement alias called `showmaxmin`. The `run` statement then runs the measurement alias and writes the `maxout` and `minout` values to the dataset.

## Using a Measurement Alias

You must define a measurement alias before you can use it. To use a measurement alias, use the `run` command. If the measurement alias is defined appropriately, you can pass in parameters to the `run` command in the measurement alias to further specify the behavior. For example, assume you have an MDL control file with the following contents.

```
alias measurement falldelay {
    input real transtop=2u // This variable is given a default value.
    input real prop_thresh // This variable has no default value.
    run tran(stop=transtop) // A run statement is required.
    export real prop_delay_fall=deltax(sig1=V(inp), sig2=V(out),
        dir1='fall', n1=1, start1=0, thresh1=prop_thresh,
        dir2='fall', n2=2, start2=0, thresh2=prop_thresh)
}
run falldelay(transtop=1u, prop_thresh=2)
```

Notice how `transtop` and `prop_thresh` are declared as input variables in the measurement alias. The `transtop` input variable is given a default value of 2u. If you do not

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

pass a value to `transtop` on the `run` statement, MDL uses the default value. However, the `prop_thresh` input variable does not have a default value, so if you do not pass in a value for it, MDL issues an error. Notice how, in this example, the values to be used for the input variables are passed in on the `run` `runtran(transtop=1u, prop_thresh=2)` statement.

You can also use netlist variables in the MDL control file without declaring the variables as input parameters. For example, notice the use of `vdd_val` in the `risetime` expression of the following control file.

```
alias measurement runtran {
    input real transtop=2u
    input real prop_thresh
    run tran(stop=transtop)

    export real rise=risetime(trim(sig=V(out),
        from=10n, to=70n), initval=vdd_val/5, // vdd_val is used here...
        inittype='y, finalval=(vdd_val/5)+2, // and here.
        finaltype='y, theta1=10, theta2=90)
}
```

The `vdd_val` parameter is a top level variable in the netlist. Notice how in the following netlist fragment, `vdd_val` is defined as a parameter.

```
global 0 vdd! vss!
include "./testmodels.scs" section tt
parameters capval=2.5p vss_val=-5 vdd_val=5 // vdd_val is defined here.
//top level
v2 (vss! 0) vsourc dc=vss_val type=dc
v1 (vdd! 0) vsourc dc=vdd_val type=dc
```

Statements that require simulation results/computation should follow the analysis statement, as shown in the following example.

```
alias measurement cvmeas {
    run ac1
    export real cv = im(DUT:d) / (6.28319 * 100000)
}
```

There are two ways to define variables in a measurement alias. These are explained using the following example of a measurement alias.

```
alias measurement tranmeas {
    export real outcross, maxq
    run tran(stop=40n)
    outcross=cross(V(q), thresh=2.5)
    maxq=max(V(q))
}
```

In the following two lines in the measurement alias definition, the `maxq` variable is declared with the `export` qualifier in a statement that is separate from the statement that uses the variable.

```
export real outcross, maxq
maxq=max(q)
```

Alternatively, you can specify the `maxq` variable in the statement that uses it, as shown below, after the `run` statement.

```
export real outcross
export real maxq=max(q)
```

## Defining Measurement Aliases on the Fly

You can define measurement aliases on the fly by adding the `as` keyword to the `run` command.

```
run ac(center=1MHz, span=1kHz) as pb
run ac(start=1_Hz, stop=10MHz) as sb
```

The simulator creates the alias before running the analysis, so in these examples, the default parameter values for the base `ac` analysis are not affected.

The `as` keyword must follow a measurement and applies only to that measurement. The following example

```
temp=50
run dc as dcat50
```

reruns the `dc` analysis with `temp=50`.

You can extend the same concept to analyses. For example, in this pair of statements

```
run tran (stop=10u) as tran1
run tran1
```

the second statement runs `tran` with `stop=10u`. The first command creates an alias to `run tran(stop=10u)` as a new analysis called `tran1`.

## Propagating Variables

MDL allows you to propagate variables through your code. For example, in the following measurement alias, notice how `rise_edge` and `fall_edge` are first calculated and then used later to calculate the value `pw`.

```
alias measurement trans {
  run tran(stop=5u)
  real rise_edge=cross(sig=V(out), dir='rise, n=1, thresh=1.5, start=0)
  real fall_edge=cross(sig=V(out), dir='fall, n=1, thresh=1.5, start=0)
  export real pw=fall_edge-rise_edge
}
run trans
```

## Defining a Macro

You can define a macro in the MDL control file using the `#define` directive. However, to read and run the macro, you must specify `+=mdle` at the command line. For example, you can define a macro for the export statement in the mdl control file, as follows:

```
#define insat(device) export real insat_/**/device = Idut.device.m0:vdss
alias measurement dcmeas {
    run dc
    insat(M11)
    insat(M22)
    insat(M09)
}

run dcmeas
```

When MDL is run, the above statements expand to the following:

```
run dc
    export real insat_M11=Idut.device.m0:vdss
    export real insat_M22=Idut.device.m0:vdss
    export real insat_M09=Idut.device.m0:vdss
}
```

## Accessing Netlist or Model Parameters

MDL allows you to access the parameters in different hierarchical depths of the netlist by specifying the full path to the parameter in a measurement alias.

For example, in a MDL measurement alias,

- A global parameter `v_vdd` can be accessed by:  
`export real par_vdd = v_vdd`
- A subcircuit parameter `mm` can be accessed by:  
`export real par_mm=x1.xmm0:mm`
- An instance parameter `vth` can be accessed by:  
`export real par_vth=i1.mp0:vth`
- A model parameter `vth0` can be accessed by:  
`export real par_vth=i1.mp0.pch1:vth0`
- An `lv/lx` parameter of an element can be accessed by:

```
export real mn0_lv1=i0.mn0:lv1
export real mn0_lv1=i0.mn0:lx2
```

You can also change the value of a parameter by specifying the full path to the parameter either in a measurement alias or at the top level of a MDL control file. For example:

- The value of a global parameter `v_vdd` can be changed by:

```
v_vdd = 1.7
```

- The value of a subcircuit parameter `mm` can be changed by:

```
x1.xmm0:mm = 3
```

- The value of a model parameter `vth0` can be changed by:

```
i1.mp0.pch1:vth0 = 1.5
```

## Accessing Model Names and Types

MDL allows you to access the model names and model types defined in the netlist.

For example, in a MDL measurement alias:

- A model name, say `mod1`, can be accessed by:

```
export real mdname=mod1:_masterName_
```

If `mod1` is the name of a model, the model name is returned. If `mod1` is the name of a subcircuit, the subcircuit name is returned.

- The model type for the model `mod1` can be accessed by:

```
export real mdtype=mod1:_type_
```

If `mod1` is a model, the primitive name for the model is returned. If `mod1` is a subcircuit, "subckt" is returned.

**Note:** Primitive name refers to the built-in device names predefined in Spectre.

## Accessing Noise Parameters

MDL allows you to access the noise parameters as follows:

- The noise parameter `rd` can be accessed by:

```
export real rd_noise=mn:rd
```

**Note:** If a parameter is both a noise parameter and a device parameter, it is treated as a noise parameter.

- Spot noise at given frequency can be accessed by:

```
export real spot_noise=mn:rd @1k
```

- The total noise for a model can be accessed by:

```
export real total_noise=mn:total
```

- The total noise for a circuit can be output by:

```
export total_noise noise:out
```

## Using Named and Primitive Analyses

In MDL, you can use both primitive analyses and named analyses. The primitive analyses are those built into Spectre. Named analyses are ones that you define in the netlist. For example, the following measurement alias runs `tran1`, which must be defined in the netlist.

```
alias measurement trans {  
    run tran1 // This is an analysis specified in the netlist.  
  
    real rise_edge=cross(sig=V(out), dir='rise, n=1, thresh=1.5, start=0)  
    real fall_edge=cross(sig=V(out), dir='fall, n=1, thresh=1.5, start=0)  
  
    export real pw=fall_edge-rise_edge  
}
```

In contrast, the following measurement alias runs `tran`, one of the analyses provided by the simulator.

```
alias measurement trans {  
    run tran(stop=1u) /* This is a built-in, primitive analysis,  
                        which is not defined in the netlist. */  
  
    real rise_edge=cross(sig=V(out), dir='rise, n=1, thresh=1.5, start=0)  
    real fall_edge=cross(sig=V(out), dir='fall, n=1, thresh=1.5, start=0)  
  
    export real pw=fall_edge-rise_edge  
}
```

## Looping Statements

MDL provides the `foreach` statement to automate repetitive simulations and for sweeps, and provides the `search` statement to identify values that are associated with significant circuit events. With the `mvarsearch` statement, you can set up performance goals for a circuit along with parameters that may be varied in attempts to reach these goals. Spectre iterates to find the optimal solution.



## foreach Statement

The `foreach` statement provides a way for you to run a simulation repeatedly.

```
foreach_statement ::=
    foreach foreach_specifier [ onerror=conditions ] {
        block_of_statements
    }

foreach_specifier ::=
    param_to_vary from alternatives
    | paramset_name

alternatives ::=
    {list}
    | array
    | swp (swp_param)

swp_param ::=
    start = strt, stop = stop [, step_def ]
    | center = cntr, span = spn [, step_def ]

step_def ::=
    step = step
    | lin = lin_num_steps
    | dec = steps_per_decade
    | log = log_num_steps

conditions ::=
    'exit' | 'continue'
```

*param\_to\_vary*      The parameter that the `foreach` statement is to vary. This can be an MDL variable, a netlist parameter, or a device parameter. Valid data types are `real`, `int`, `cplx`, and `analysis`.

Each time the `block_of_statements` iterates, the *param\_to\_vary* is replaced by the next value from the *alternatives*, except when the *param\_to\_vary* is `analysis`. When the *param\_to\_vary* is `analysis`, a run statement is required to iterate the next value from the *alternatives* (see Example 3 for details).

*paramset\_name*      Name of the paramset definition from the netlist.

*block\_of\_statements*      One or more MDL statements, except variable declarations, measurement alias declarations, and include statements.

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<i>{list}</i>	<p>A list of values containing alternative values for <i>param_to_vary</i> in the form { <i>val1</i>, <i>val2</i>,...<i>valN</i> }.</p> <p>Valid data types are <i>real</i>, <i>int</i>, <i>cplx</i>, and <i>analysis</i>. If an array is present, it must be a de-referenced array element such as <i>mytran</i>[0], <i>mytran</i>[1], etc.</p>
<i>array</i>	An array of sweep values.
<i>strt</i>	The starting value for <i>param_to_vary</i> . The <i>strt</i> and <i>stop</i> parameters are used together to specify sweep limits.
<i>stop</i>	<p>The ending value for <i>param_to_vary</i>. The <i>strt</i> and <i>stop</i> parameters are used together to specify sweep limits.</p> <p>If you do not give a <i>step_def</i>, the sweep is linear when the ratio of stop to strt values is less than 10, and logarithmic when this ratio is 10 or greater.</p>
<i>cntr</i>	Center value of sweep. The <i>cntr</i> and <i>spn</i> parameters are used together to specify sweep limits.
<i>spn</i>	<p>Span of sweep. The <i>cntr</i> and <i>spn</i> parameters are used together to specify sweep limits.</p> <p>If you do not give a <i>step_def</i>, the sweep is linear when the ratio of the end point of the span to the start point of the span is less than 10, and logarithmic when this ratio is 10 or greater.</p>
<i>step</i>	Step size for linear sweeps.
<i>lin_num_steps</i>	Number of steps for linear sweeps.
<i>steps_per_decade</i>	Number of points per decade for log sweeps.
<i>log_num_steps</i>	Number of steps for logarithmic sweeps.
<b>'exit</b>	A keyword specifying that the <i>foreach</i> loop is to end when the simulation experiences an error, such as a convergence issue. This is the default behavior if the <i>onerror</i> option is not specified.
<b>'continue</b>	A keyword specifying that the <i>foreach</i> loop is to continue even when the simulation experiences an error, such as a convergence issue.

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

## Example 1

For example, you might define a measurement and `foreach` statement like the following to determine the maximum output voltage of a circuit. Notice that the `foreach` statement is *not* placed inside the `alias` measurement statement.

```
alias measurement findmax {
    run tran(stop=1u)
    export real maxout=max(V(out))
}
foreach vdd_val from swp(start=5, stop=7, step=0.5) {
    foreach temp from {25, 50, 75, 100} {
        run findmax
    }
}
```

In this example, the outer `foreach` statement varies the value of `vdd_val` and the inner `foreach` varies the value of `temp`. As a result, the `findmax` measurement alias runs with each combination of values, producing a `.measure` file like this.

```
Swept Measurements :
Measurement Name    : findmax
Analysis Type       : tran

maxout              vdd_val @ 5      temp @ 25    = 3.07027
maxout              vdd_val @ 5      temp @ 50    = 3.07326
maxout              vdd_val @ 5      temp @ 75    = 3.07565
maxout              vdd_val @ 5      temp @ 100   = 3.08015
maxout              vdd_val @ 5.5    temp @ 25    = 3.06517
maxout              vdd_val @ 5.5    temp @ 50    = 3.06917
maxout              vdd_val @ 5.5    temp @ 75    = 3.07468
maxout              vdd_val @ 5.5    temp @ 100   = 3.07433
maxout              vdd_val @ 6      temp @ 25    = 3.06553
maxout              vdd_val @ 6      temp @ 50    = 3.06703
maxout              vdd_val @ 6      temp @ 75    = 3.06865
maxout              vdd_val @ 6      temp @ 100   = 3.07364
maxout              vdd_val @ 6.5    temp @ 25    = 3.06373
maxout              vdd_val @ 6.5    temp @ 50    = 3.06524
maxout              vdd_val @ 6.5    temp @ 75    = 3.0695
maxout              vdd_val @ 6.5    temp @ 100   = 3.07092
maxout              vdd_val @ 7      temp @ 25    = 3.06274
maxout              vdd_val @ 7      temp @ 50    = 3.06595
maxout              vdd_val @ 7      temp @ 75    = 3.06798
maxout              vdd_val @ 7      temp @ 100   = 3.07035
```

## Example 2

As another example, the `paramset` statement is defined as follows in a netlist

```
data_v paramset {
    vhi vlo
    1.9 1.32
    1.8 1.2
}

data_fet paramset {
    nw nl pw pl
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
5u 3u 9u 3u
4u 3u 7u 3u
}
```

The `alias` and `foreach` statements are defined as follows in the `.mdl` file

```
alias measurement findvth {
  run tran(stop=100n)
  export real outVth = cross(sig=V(out), dir='cross', n=1, thresh=2.5, start=0)
}
foreach temp from {0, 130} {
  foreach data_v {
    foreach data_fet {
      run findvth
    }
  }
}
```

In this example, there are three-level nested sweeps: the inner `foreach` statement varies the value of `nw`, `nl`, `pw` and `pl` from the `data_fet` paramset statement defined in the netlist; the middle `foreach` statement varies the value of `vhi` and `vlo` from the `data_v` paramset statement defined in the netlist; and the outer `foreach` statement varies the value of `temp`. As a result, the `findvth` measurement alias is run eight times by MDL, producing a `.measure` file like this:

```
Swept Measurements :
  Measurement Name   :   findvth
  Analysis Type      :   tran

outVth               temp @ 0
                    vhi @ 1.9
                    vlo @ 1.32
                    nw @ 5e-06
                    nl @ 3e-06
                    pw @ 9e-06
                    pl @ 3e-06      = 9.89772e-10

outVth               temp @ 0
                    vhi @ 1.9
                    vlo @ 1.32
                    nw @ 4e-06
                    nl @ 3e-06
                    pw @ 7e-06
                    pl @ 3e-06      = 1.01348e-09

outVth               temp @ 0
                    vhi @ 1.8
                    vlo @ 1.2
                    nw @ 5e-06
                    nl @ 3e-06
                    pw @ 9e-06
                    pl @ 3e-06      = 1.37049e-09
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
outVth      temp @ 0
            vhi @ 1.8
            vlo @ 1.2
            nw @ 4e-06
            nl @ 3e-06
            pw @ 7e-06
            pl @ 3e-06      = 1.74369e-09

outVth      temp @ 130
            vhi @ 1.9
            vlo @ 1.32
            nw @ 5e-06
            nl @ 3e-06
            pw @ 9e-06
            pl @ 3e-06      = 9.56855e-10

outVth      temp @ 130
            vhi @ 1.9
            vlo @ 1.32
            nw @ 4e-06
            nl @ 3e-06
            pw @ 7e-06
            pl @ 3e-06      = 9.82614e-10

outVth      temp @ 130
            vhi @ 1.8
            vlo @ 1.2
            nw @ 5e-06
            nl @ 3e-06
            pw @ 9e-06
            pl @ 3e-06      = 1.17573e-09

outVth      temp @ 130
            vhi @ 1.8
            vlo @ 1.2
            nw @ 4e-06
            nl @ 3e-06
            pw @ 7e-06
            pl @ 3e-06      = 1.22687e-09
```

## Example 3

```
alias measurement myrun {
    input analysis tranRun = tran1
    run tranRun
}

analysis myAltergroups[] = { ag1, ag2, ag3, ag4 }
analysis ag, myAnalysis
foreach ag from myAltergroups {
    run ag      //This is required to activate the next analysis
    foreach myAnalysis from { dc1, dcswp, tran1 } {
        run myrun( tranRun = myAnalysis ) as myAnalysis
    }
}
```

```
    }  
}
```

## Specifying the foreach Statement Within the measurement Alias

MDL also supports the `foreach` statement within the measurement alias.

### Example

```
alias transient {  
    export real v1  
    export real temper  
    temper=temp  
    run tran( stop=7e-08 )  
    v1=aa*avg(V(vin))  
}  
  
alias measurement top_test {  
    export real sum =0;  
    export real each_v1[];  
    int index=0;  
    foreach aa from {10, 20 ,30} {  
        run transient as inner;  
        sum = sum + inner->v1;  
        each_v1[index] = inner->v1;  
        index = index+1;  
    }  
}
```

## search Statement

The `search` statement provides a way for you to find the value of a design parameter that corresponds to the circuit meeting or failing a specific performance criterion. The function operates by running the simulation repeatedly, varying the values of interest each time, until a specified condition is met. This capability is typically used to determine values such as setup time and maximum load.

```
search_statement ::=  
    search search_specifier [ output=conditions ] {  
        block_of_statements  
    } method ( condition_statements )  
  
search_specifier ::=  
    param_to_vary from binary ( start=strt, stop=stp, tol=tol  
    round=['no'|'yes'])  
  
conditions ::=  
    'none' | 'last' | 'all' | 'each'
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
method::=  
    until  
    | while  
    | bisection
```

*param\_to\_vary* The parameter that the search statement is to vary in a binary way.

*strt* The starting value for *param\_to\_vary*. The *strt* and *stp* values should straddle the expected value. If both *strt* and *stp* result in a *condition\_statement* that is true or if both result in a *condition\_statement* that is false, the search statement fails.

*stp* The ending value for *param\_to\_vary*. The *strt* and *stp* values should straddle the expected value. If both *strt* and *stp* result in a *condition\_statement* that is true or if both result in a *condition\_statement* that is false, the search statement fails.

*tol* The tolerance value, which specifies how precisely the final value is calculated.

*round*='no' | 'yes' If set to 'yes, each iteration of the search is an integral value that is rounded of to the nearest middle value.

*block\_of\_statements* One or more *run* statements.

*condition\_statements* A statement that determines when the search statement stops. You can use multiple boolean expressions in conditional statements.

**'none'** A keyword indicating that no analysis output is to be saved to the raw directory irrespective of whether the search succeeds or fails.

**'last'** A keyword indicating that the analysis output is to be saved only when the last iteration of the search succeeds. No output will be saved to the raw directory if the search fails.

**'all'** A keyword indicating that the analysis output is to be saved to the raw directory irrespective of whether the search succeeds or fails. This is the default keyword.

**'each'** A keyword indicating that the analysis output for each iteration of the search is saved to the raw directory irrespective of whether the search succeeds or fails.

<b>until</b>	A keyword indicating that the search statement is to continue looping until the value of <i>param_to_vary</i> causes <i>condition_statements</i> to become true and then return the first iteration value that meets the tolerance (tol) criteria when <i>condition_statements</i> is true. If you use this keyword, <i>condition_statements</i> must initially be false.
<b>while</b>	A keyword indicating that the search statement is to continue looping until the value of <i>param_to_vary</i> causes <i>condition_statements</i> to become false and then return the last iteration value that meets the tolerance (tol) criteria when <i>condition_statements</i> is true. If you use this keyword, <i>condition_statements</i> must initially be true.
<b>bisection</b>	A keyword indicating that the search statement is to continue looping till either an until or a while condition is first met.

## Example 1

For example, you might define a measurement and search statement like the following to determine the setup time of a flip-flop.

```
alias measurement setup {
    export real vdelay, outcross, Tsetup, vcdelay, setdelay, maxq
    run tran(stop=40n)
    vdelay=cross(sig=V(data), thresh=2.5, dir='rise, n=1)
    vcdelay=cross(sig=V(clock), thresh=2.5, dir='rise, n=1)
    outcross=cross(V(q), thresh=2.5)
    maxq=max(V(q))
    setdelay=vdata:delay
    Tsetup=vcdelay-vdelay
}

search vdata:delay from binary(start=2n, stop=10n, tol=1p) {
    run setup
} until ( setup->maxq < 2.5)
```

In this example, the search statement varies the parameter *vdata:delay*, which is a parameter named *delay* on a instance named *vdata*. The simulation determines when *V(q)* crosses a threshold value. When *V(q)* fails to cross the threshold, *maxq* remains less than 2.5, making the *condition\_statements* true. The *setup->maxq* syntax in the *condition\_statement*, refers to the *maxq* value in the *setup* block.

Through repeated simulations, the search statement closes in on the value of *vdata:delay* that marks the change from *condition\_statements* being false to *condition\_statements* being true.



## Example 2

You can also choose whether to save the search results to the raw directory by specifying an output parameter. This feature can be used to prevent creating large size output files that may cause memory issues.

For example, in the following `search` statement, two conditions are specified in the `while` criteria. The simulator will exit and return the last successful value whenever one of the two conditions fails.

```
search vdata:delay from binary(start=2n,stop=10n,tol=1p) output='last {  
    run setup  
} while (setup->maxq > 1.8 && setup->outcross < 10.3n)
```

As `output='last` is set, the analysis output of the setup measurement will be saved to the raw directory only when the last iteration of the search is successful, otherwise no analysis result is saved.

**Note:** If the two conditions in the `while` criteria are defined in an OR relationship (using the `||` operator), but not in an AND relationship (using the `&&` operator), the simulator will exit and return the last successful value only when both the conditions fail.

## mvarsearch Statement

The `mvarsearch` statement provides a way for you to find the values of design parameters that correspond to the optimal solution of a group of measurements. In essence, this statement provides a multi-parameter, multi-goal search functionality.

This statement works by setting the design parameters to a value, running the defined measurement, evaluating the goal functions, calling an optimizer to determine the next set of design parameter values, and repeating. The statement iterates until an optimal solution is found, or until the maximum number of optimization iterations have been performed.

When the `mvarsearch` is used inside a `foreach` loop, the `restoreParam` option must be set to 1 to reset the parameters to their initial values after the optimization is complete and avoid any errors in subsequent `foreach` loops.

```
mvarsearch_statement ::=  
    mvarsearch  
        option {  
            options_statements  
        }  
        parameter {  
            parameter_statements  
        }  
        exec {  
            exec_statement
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
    }
    zero
        zero_statements
    }
}

options_statement ::=
    [ method = method ],
    [ accuracy = conv_tol ],
    [ deltax = diff_tol ],
    [ maxiter = maxiter ],
    [ restoreParam = restoreParam ],
    ]

parameter_statement ::=
    { param_name, init_val, lower_val, upper_val }
```

*method* The method to be used in the algorithm method. Possible values are 'newton (newton solver) and 'lm (levenburg marquardt). While 'newton is faster than 'lm, 'lm has more robust convergence properties. You should use 'newton only when *init\_val* values are close to the final solution and you want a faster optimization.

'newton assumes that the number of *param\_statement*=number of *zero\_statement*. If this is not true, mvarsearch automatically changes method to 'lm.

Default value: 'lm

*conv\_tol* Convergence tolerance. The optimization ends if the relative error in the sum of squares of the calculated objectives in the *zero\_statements* is less than *conv\_tol*. For example, given objectives *tmp1* and *tmp2*, the optimization ends if:

$$(tmp1*tmp1 + tmp2*tmp2) < conv\_tol$$

The smaller the value of *conv\_tol*, the more accurate the optimization solution.

Default value: 1.0e-04

*diff\_tol* Step length for the forward-difference approximation to compute the numerical derivatives. For a design parameter of value *x*, the approximation uses *diff\_tol*\**x* as the step length. If *x* is very small (less than the machine precision), the step length is taken as *diff\_tol*.

Default value: 5.0e-03

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<i>maxiter</i>	Maximum number of optimization iterations that should be evaluated before automatically exiting the optimization loop. Default value: 300
<i>restoreParam</i>	Specifies if the parameters to be optimized should be reset to their initial values after the optimization is complete ( <i>restoreParam</i> =1). Default value: 0
<i>param_name</i>	The parameter to be optimized.
<i>init_val</i>	Initial value for the parameter to be optimized. This value provides the optimization algorithm with an initial guess for the parameter to be optimized. Default value: 1.0
<i>lower_val</i>	Lower limit for the parameter to be optimized. The optimization limits parameter to the value of <i>lower_val</i> if it becomes less than the specified value. <i>lower_val</i> can be used to force a lower limit on physical parameters (such as MOSFET channel width) to ensure that the optimized solution is physically possible.
<i>upper_val</i>	Upper limit for the parameter to be optimized. The optimization limits the parameter to the value of <i>upper_val</i> if it exceeds the specified value. <i>upper_val</i> can be used to force an upper limit on physical parameters (such as MOSFET channel width) to ensure that the optimized solution is physical possible. Setting <i>lower_val</i> and <i>upper_val</i> too close to each other can result in discontinuities, making the optimization unsuccessful. Default value: 2.0* <i>init_val</i>
<i>exec_statement</i>	run statement to compute goal values.
<i>zero_statement</i>	Goal value to be minimized.

For example, you may define a measurement and *mvarsearch* statement as follows to obtain the optimal values of p-channel width and n-channel width, and an equivalent rise and fall time of 3ns for an inverter chain.

```
alias measurement trans {
run tran( stop=1u, autostop='yes )
    export real rise=risetime(sig=V(d), initval=0, inittype='y, finalval=3.0,
        finaltype='y, theta1=10, theta2=90) // measured from 10% to 90%
    export real fall=falltime(sig=V(d), initval=3.0, inittype='y, finalval=0.0,
        finaltype='y, theta1=90, theta2=10) // measured from 10% to 90%
```

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
}
mvarsearch {
  option {
    accuracy = 1e-3      // convergence tolerance of trans->rise
    deltax = 1e-3        // numerical difference % of design variables
    maxiter = 100        // limit to 100 iterations
  }
  parameter {
    {para_pw, 1.2u, 0.1u, 10u}
    {para_nw, 1.2u, 0.1u, 10u}
  }
  exec {
    run trans
  }
  zero {
    tmp1 = trans->rise - 3ns
    tmp2 = trans->fall - 3ns
  }
}
```

In the above example, design parameters `para_pw` and `para_nw` are varied by the optimization algorithm starting at an initial value of 1.2 microns with a maximum value of 10 microns and a lower limit of 0.1 microns. At each iteration, the measurement alias `trans` is run after the design parameter value is set. The `zero` values `tmp1` and `tmp2` are then computed using the results from the measurement alias. This iteration continues until one of the following happens:

- `tmp1` and `tmp2` satisfy the `conv_tool` criteria determined by the following equation:  
$$(\text{tmp1} * \text{tmp1} + \text{tmp2} * \text{tmp2}) < 1.0\text{e-}03$$
- the `maxiter` parameter value is exceeded

During this optimization, the parameters `para_pw` and `para_nw` are clamped between the lower limit of 0.1u and the upper limit of 10u. This clamping forces the channel widths of the MOSFETs in this circuit to remain within defined limits while the optimization is performed.

The above example results in the following output:

```
Swept Measurements      :
Measurement Name        : trans-meas_optimize
Analysis Type           : tran
fall                    para_pw @ 2.3745e-06
                        para_nw @ 1.16767e-06 = 2.99999997237325e-09
rise                    para_pw @ 2.3745e-06
                        para_nw @ 1.16767e-06 = 3.00129681779706e-09
```

You can define a `mvarsearch` statement inside `foreach` statements as follows:

```
foreach v_vdd from {3, 2.5}
{
  foreach temp from {25, 30}
  {
    mvarsearch {
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
    option {
    accuracy = 1e-3      // convergence tolerance of trans
    deltax = 1e-3        // step length
    maxiter = 100        // limit to 100 iterations
    restoreParam=1      // a must
    }
    parameter {
        {pw, 2u, 0.05u, 10u}
        {nw, 2u, 0.05u, 10u}
    }
    exec {
        run trans
    }
    zero {
        tmp1 = trans->rise-25p
        tmp2 = trans->fall-25p
    }
}
}
```

The above example results in the following output (with `-tab` option in the `spectremdl` command line):

```
Swept Measurements :
Measurement Name    : trans
Analysis Type       : tran
v_vdd  temp  pw      nw      fall      rise
3_      25    5.58951e-06 4.08718e-06 2.38087e-11 2.62996e-11
3       30    3.6754e-06  3.39895e-06 2.41e-11    2.66064e-11
2.5     25    3.72744e-06 2.05837e-06 2.32133e-11 2.87913e-11
2.5     30    3.35662e-06 1.84146e-06 2.41816e-11 2.96455e-11
```

## Include Statement

The `include` statement provides a way for you to insert the contents of an MDL control file into another control file. This feature is useful for creating an MDL control file using other control files as components.

```
include_statement ::=
    include "mdlfile"
```

*mdlfile*                      Path and filename of the MDL file to be inserted.

Remember the following rules when using the `include` statement.

1. The `include` statement is allowed at the top level of the MDL file only.
2. Alias measurement blocks, `foreach`, `search`, `mvarsearch`, and `montecarlo` statements do not support the `include` statement as a subordinate command.

In the following example,

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
//File main.mdl
include "simple.mdl"
run tr
```

the `main.mdl` and `simple.mdl` files are placed in the current directory.

In the following example,

```
//File simple.mdl
include "meas/mdl2.2"
alias measurement t
run tran(stop=140n)
    export real vall=3*V(11)
}
run tr3
include "meas/mdl1.2"
run tr2
```

the `mdl1.2` and `mdl2.2` files are placed in the `meas` directory.

## Evaluating Expressions Selectively

You can specify the criteria based on which you want MDL to run other statements and evaluate expressions.

### If/Else Statement

The `if/else` statement provides a way for MDL to run other statements selectively according to criteria that you specify.

```
if ( CONDITION ) {
    TRUESTATEMENTS
}
[
(
    else if (ELSEIFCONDITION) {
        ELSEIFSTATEMENTS
    }
)+
]
[
else {
    FALSESTATEMENTS
}
]
```

[...] Optional block.

(...)+ Indicates that the block can be repeated.

`if` Indicates that this is an if block of the `if/else` statement.

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<code>else</code>	Indicates that this is an else block of the <code>if/else</code> statement.
<code>CONDITION</code>	Conditional expression which returns 1 if the condition is true and 0 if the condition is false
<code>ELSEIFCONDITION</code>	Conditional expression which returns 1 if the condition is true and 0 if the condition is false.
<code>TRUESTATEMENTS</code>	Statements that are read only if <code>CONDITION</code> is true.
<code>ELSEIFSTATEMENTS</code>	Statements that are read only if <code>ELSEIFCONDITION</code> is true and <code>CONDITION</code> and all previous <code>ELSEIFCONDITION</code> are false.
<code>FALSESTATEMENTS</code>	Statements that are read only if <code>CONDITION</code> and all previous <code>ELSEIFCONDITION</code> are only.

The `if/else` statement is allowed

- at the top level of an MDL file, and it can include `assign`, `run`, `mvarsearch`, `search`, `foreach`, `montecarlo`, `print` statements
- inside alias measurement block and executed blocks of `mvarsearch`, `search`, `montecarlo`, `foreach` statements

## Example 1

```
int count=0
foreach count from swp(start=1, stop=3, step=1) {
    if (count==1)
    {
        run tr3
    }
    else if (count == 2)
    {
        if (tr2->val2 == 5)
        {
            run tr2
        }
        run tr1
    }
    else
    {
        run tr2
    }
}
```

## Example 2

The following example shows the value of `v(11)` change from 0 to 5 during the simulation.

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
alias measurement tr1 {
    run tran(stop=160n)
    export real val2
    export real val1
    export real val3
    export real val4
    export real val5
    export real val6
    export real val7
    export real val8
if ( v(11) > 4)
{
    if ( v(11) < 4.06 )
    {
        val1=v(11)
    }
    val2 = min(v(11))
}
else if ( v(11) > 3 )
{
    if ( (v(11) > 3.5) && (v(11)<3.7) )
    {
        val3= max( v(11)
        val4= v(11)
    }
    else
    {
        val5=v(11)
        val6=max(v(11))
    }
}
else
{
    val7 = max(v(11))
    val8 = v(11)
}
}
run tr1
```

In the example above, `val7 = 2.955`, because the `max` function is a buffered function and assign statement `val7 = max(v(11))` is executed while `v(11) <= 3`.

## Ternary Expression Statement

The ternary expression statement provides a way for MDL to evaluate expressions selectively according to criteria that you specify. A ternary expression statement may be used at any location where an expression is supported. Furthermore, as this statement is an expression, its return value may be used as the argument to an assignment statement.

*(CONDITION) ? TRUEEXPRESSION : FALSEEXPRESSION*

`?` MDL keyword indicating that this is a ternary if-operator.



# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

:	MDL keyword separates TRUEEXPRESSION and FALSEEXPRESSION.
<i>CONDITION</i>	Conditional expressions which returns 1 if the condition is true and 0 if the condition is false.
<i>TRUEEXPRESSION</i>	Expression returned if <i>CONDITION</i> is true.
<i>FALSEEXPRESSION</i>	Expression returned if <i>CONDITION</i> is false.

Remember the following when using a ternary expression statement:

- Add a space before a not-equal-mark (!=). This is to differentiate the not-equal from value assignment to a global signal which is usually suffixed with "!".
- Add a parenthesis before the colon mark (:) when a letter follows it. This is to differentiate the *TRUEEXPRESSION* from the expression of *instance:terminal*

## Example

```
val2 = (v(11) == 5)&&(val1 != 'nan) ? v(11) : 2*v(11)
val3 =(val1 != 'nan)?  mag( V(11)==5?max(V(11)):10 )+3 : 2*V(11)
```

## Specifying the Output File Format

The print statement provides a way for you to write strings and variables (such as parameters and measurement results) from an MDL control file to the standard output file or an output file defined by you.

You can add a print statement at the following levels:

- at the top level of an MDL control file
- at the level of an alias measurement
- inside a foreach looping statement
- inside an optimization looping statement (such as *search* or *mvarsearch*)
- inside a monte-carlo analysis

```
print _statement::=
    print fmt ( "format", args ) [ to=file | addto=file ]

format::=
    % [ flag ][ width][ .precision ] type [ \n| \t ]
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

`flag ::=`  
`-`

`type ::=`  
`d,i | E,e | f | G,g | o | S | s | u | X,x | V`

`flag` Symbol used to align the text in the output file. `-` left-aligns the text. If `flag` is not specified, the text is right-aligned.

`width` Minimum field width.

*precision* Maximum number of the significant digits to be printed for `g`, `G`, `S`, and `s` types, or the number of decimal digits to be printed for `e`, `E` and `f` types.  
Default value: 6

`type` Defines how the text is to be printed to the output file. The following are supported in MDL:  
`d, i` – signed decimal integer  
`E, e` – floating point (in scientific notation)  
`f` – floating point (in decimal)  
`G, g` – the shorter of `%e`, `%E` and `%f` (suppresses non-significant zeros)  
`o` – unsigned octal integer  
`S` – engineering scale number (e.g. 5m, 5K)  
`s` – string  
`u` – unsigned decimal integer  
`X, x` – unsigned hexadecimal integer.  
`V` – the value of the args

`\n` Specifies that a new line is to be inserted.

`\t` Specifies that a tab space is to be inserted.

`to` Specifies that the existing results in the output file are to be overwritten by the new results. You should use the `to` option in the first print statement to clean the old results in the specified results file.

`addto` Specifies that the results are to be appended to the output file.

*file* File name.



## Example 3

The following example shows how to print the intermediate results of an `mvarsearch` statement.

```
print fmt ("\n****Print Results of Optimization Analysis****\n\n")
to="print_opt.dat"
print fmt ("%15s%-15s%-15s%-15s\n", "pw","nw","rise","fall")
addto="print_opt.dat"
mvarsearch {
    ...
    exec {
        print fmt ("%15e%-15e", pw, nw) addto="print_opt.dat"
        run trans
        print fmt ("%15e%-15e\n", trans->rise,trans->fall) \
            addto="print_opt.dat"
    }
    ...
}
```

The simulator writes the following results to the `print_opt.dat` file:

```
****Print Results of Optimization Analysis****
pw          nw          rise          fall
2.000000e-062.000000e-062.469632e-112.371606e-11
2.063246e-062.000000e-062.451040e-112.387945e-11
...
...
2.007259e-061.719886e-062.499470e-112.500292e-11
2.007259e-061.719886e-062.499470e-112.500292e-11
```

## Autostop

Autostop is a feature that halts simulation as soon as enough data has been collected to evaluate the MDL expressions associated with a transient analysis. Using the autostop feature can save you an enormous amount of simulation time when you characterize circuits. The autostop feature is supported only for transient analysis.

Only functions that determine specific events, such as delay and event measurements, can cause an automatic stop. However, if non-event functions such as `max` and `min` are included in the same measurement alias, the functions are evaluated over the simulation period defined by the event functions.

To use the autostop feature, you turn on the autostop parameter of the `tran` statement. For example, the `tran` statement defined in the design file might look like this.

```
tran1 tran stop=6u method=gear2only autostop=yes
```

Then, in the MDL control file, you specify the information you want to gather. For example, your control file might look like this.

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
alias measurement trans {
run tran1
  export real out_1u=V(out)@1u
  export real prop_delay_fall=deltax(sig1=V(inp), sig2=V(out), dir1='fall',
    n1=1, start1=0, thresh1=1.5, dir2='fall', n2=1, start2=0, thresh2=1.5)
}
run trans
```

This example control file contains two expressions. The first measures the output voltage at 1 $\mu$ s, and the second determines the delay between the input and output falling edges. Because the autostop feature is enabled, the simulation runs only as long as necessary to calculate the two specified values.

If the control file also specifies a third expression such as

```
export real outmax=max(V(out))
```

the simulator finds the maximum value only in the part of the simulation prior to the automatic halt. If there happens to be a greater maximum that occurs after the automatic halt, the simulator does not find it when autostop is enabled.

## Monte Carlo

Monte Carlo statements provide a way to run Monte Carlo in MDL for measurement and statistical figures of merit.

```
montecarlo_statement ::=
run montecarlo ( options_statements )
{
block of statements
}
options_statement ::=
  [numruns = <int>, ]
  [seed = <int>, ]
  [variations= <'process |'mismatch |'all >, ]
  [firstrun = <int>, ]
  [ donominal = < 'yes, 'no>, ]
  [ scalarfile =filename, ]
  [ appendsd = < 'yes, 'no >, ]
```

For more information on the block of statements, see [Using a Measurement Alias](#) on page 19. The options in the options\_statement are consistent with the Monte Carlo analysis in Spectre. For more information on these options, see the *Spectre Circuit Simulator and Accelerated Parallel Simulator User Guide*.

From the MMSIM6.1 release, you can have a Monte Carlo statement inside a foreach loop.

The following information is written to the output file:

- number of iterations

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

- exported measurement values for each of runs

Statistical figures of merit are also computed and output as part of the termination of the MDL Monte Carlo run:

max	The maximum value.
min	The minimum value.
mean	The mean value.
var	The variance from the mean.
stddev	The standard deviation.
avgdev	The average deviation, mean absolute deviation.
failedtime	The number of failed runs.

When the monte carlo analysis is run in MDL for measurement, max, min, mean, var, stddev, and avgdev are computed and written to the output file and NULL values are ignored.

For more information on the functions, see [Appendix A, “Built-In Functions.”](#)

For example, you can define a measurement and Monte Carlo statement like the following for a flip-flop circuit

```
alias measurement tranmeas {
export real rise_out
  run tran(stop=40n, errpreset='conservative')
  rise_out=risetime(V(q), initval=minq, finalval=maxq, inittype='y, \\
  finaltype='y, theta1=10, theta2=90)
}
run montecarlo ( scalarfile="dflip.dat", numruns=50, seed=8, donominal='no,
variations='all, firstrun=1)
{
run tranmeas
}
```

## Supported Spectre Circuit Simulator Analyses

MDL supports the following Spectre circuit simulator analyses inside an alias measurement block:

- Transient analysis, including transient noise, transient ac, and transient info (tran)

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

- AC analysis (`ac`)
- DC analysis (`dc`)
- Sweep analysis (`sweep`)
- Noise analysis (`noise`)
- Monte Carlo (`montecarlo`)
- Circuit Information (`info`)
- Alter Group (`altergroup`)
- S-parameter analysis (`sp`)
- Stability Analysis (`stb`)
- Reliability Analysis (`rel`)

The analysis can be defined in the netlist or in the MDL control file.

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

The following table displays the syntax differences between Spectre and MDL.

Analysis	Spectre Circuit Simulator Syntax	MDL Syntax
transient	tran1 tran stop=1u errpreset=conservative	run tran1 [as tran2]  or  run tran (stop=1u, errpreset='conservative) [as tran2]
transient ac	tran2 tran actimes=[0 20n 40n] acnames=[CaptabInfo ac-2] stop=40n	run tran2 or  run tran ( actimes={0, 20n, 40n}, acnames={CaptabInfo, ac-2}, stop=40n)
transient info	tran3 tran infotimes=[10n 30n] infoname=opInfo stop=40n	run tran3 or  run tran ( infotimes={10n, 30n}, infoname=opInfo, stop=40n)
transient noise	tran1 tran stop=50n noiseseed=10 noiseffmax=30G noiseffmin=1M	run tran ( stop=50n, noiseseed=10, noiseffmax=30G, noiseffmin=1M )
AC	ac1 ac start=0.1G stop=1G dec=25	run ac1 [as ac2]  or  run ac (start=0.1G, stop=1G, dec=25) [as ac2]
DC	dc1 dc oppoint=logfile	run dc1 [as dc2] or  run dc (oppoint='logfile) [as dc2]
DC sweep	dcswp1 dc param=temp start=-40 stop=40 step=10	run dcswp1 [as dcswp2]  or  run dc (param=temp, start=-40, stop=40, step=10) [as dcswp2]
sweep	swp1 sweep param=temp values=[25 50] { swp2 sweep param=vdd values=[0.8 3.3] { tran1 tran stop=10n } }	run swp1 or  foreach temp swp from{25, 50} { foreach vdd swp from {0.8, 3.3} { run tran(stop=10n) } }



## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

<b>noise</b>	<code>findNoise (out gnd) noise oprobe=out iprobe=v4 start=1 stop=1MHz dec=10</code>	<code>run findNoise [as findNoise2]</code>  <b>or</b> <code>run noise( oprobe=out, iprobe=v4, start=1, stop=1M dec=10, terminals={"out","gnd"}) [as findNoise2]</code>
<b>montecarlo</b>	<code>mc1 montecarlo scalarfile=monte5a.dat numruns=10 variations=all seed=1 { ... }</code>	<code>run montecarlo (scalarfile="monte4.dat", numruns=10, variations='all', seed=1) { ... }</code>
<b>circuit information</b>	<code>dcOpInfo info what=oppoint where=rawfile</code>	<code>run dcOpInfo [as dcOpInfo2]</code>  <b>or</b> <code>run info (what='oppoint where='rawfile) [as dcOpInfo2]</code>
<b>alter group</b>	<code>alter1 altergroup { ...}</code>	<code>run alter1 [as alter2]</code>
<b>S- parameter</b>	<code>sp1 sp ports=[PORT_1 PORT_2] dec=20 start=0.1G stop=20G</code>	<code>run sp1 [as sp2]</code>  <b>or</b> <code>run sp (ports={PORT_1, PORT_2}, dec=20, start=0.1G, stop=20G) [as sp2]</code>
<b>Stability Analysis</b>	<code>stb1 stb start=1 stop=1e10 dec=100 probe=Vprobe</code>	<code>run stb(start=1, stop=1e10, dec=100, probe=Vprobe)</code>
<b>Reliability Analysis</b>	<code>rel reliability { age time = [10y] deltad value = 0.1 ..... //run analysis ..... }</code>	<code>run reliability (time_age=[10y], deltad_value = 0.1 ) { ..... //run MDL measurement ..... }</code>

Note that `ac` or `info` analyses used as part of a `tran` statement (`opInfo`, `ac-2`, and `CaptabInfo` in the above table) must be defined in the netlist.

Almost all Spectre netlist pre-defined core analyses are supported by MDL, except for sweep and montecarlo analyses. Therefore almost any predefined analysis can be defined and run outside the alias measurement blocks in an MDL control file, such as at the top level or anywhere a `run` statement may be present which includes inside the `foreach`, `montecarlo`, `search` and `mvarsearch` statements. But the results may not be accessible, nor may measurements be performed on the results as real time data from analyses is accessible only when the analysis is supported inside an alias measurement block.

## Supported Spectre Circuit Simulator Formats

The following Spectre circuit simulator formats allow the creation of an MDL `.measure` file:

- PSFBIN
- PSFASCII
- SST2
- FSDB
- WDF
- TR0ASCII

The following Spectre circuit simulator formats are not supported and do not allow the creation of an MDL `.measure` file:

- PSFBINF
- WSFBIN
- WSFASCII
- NUTBIN
- NUTASCII

## Optimizations and Tips and Tricks

### Data Output Optimizations

1. Use simulator option `save=nooutput` in your design file to disable simulator data save and enhance performance. If you want to save simulator data, use `save=selected` in your design file and specify the signals to be saved.
2. Use the `-rmrawfiles` command line option to delete the `.raw` directory after each MDL run. The `.measure` file is preserved. This minimizes disk space usage between runs.
3. Use `rawfmt=psfbin` (default setting) for best output performance.
4. Only export the variables that you need written to the `.measure` file.

## Performance Optimizations

1. Use the `paramset` in the `foreach` loop if multiple `foreach` runs are desired.
2. Use the `autostop` parameter on the transient analysis runs with the functions `cross()`, `trim()`, `deltax()` to specify termination of the analysis run after values have been computed.
3. Use default accuracy (`moderate`) on the transient analysis, as MDL thresholds and breakpoints ensure accuracy. You do not need to specify small timesteps for equivalent accuracy with MDL.
4. Instead of recomputing multiple identical expressions, use a single expression computation and temporary variables in the measurement aliases.
5. To speed up `mvarsearch` runs, set the nominal value of each varied parameter close to the expected value, if known.
6. Use multiple measurement functions such as `crosses()` and `dutycycles()` instead of using multiple `cross()` or `dutycycle()` measurements.

## MDL Reuse

1. Use parameterized alias measurements and `include` statements for MDL alias measurement code reuse.
2. To share data between alias measurement runs, or access data from a run at the top level, use the `->` operator in the top level constructs for accessing previous alias measurement run results.
3. When using the `->` operator to access computed data, remember that re-naming the measurement alias run using the `as` command changes the measurement alias name for the “`->`” operator. For example,

```
alias measurement maxout
{
    input net mynet;
    export real out;
    run tran1;
    out = max(V(mynet));
}

run maxout(mynet=out);
// maxout->out = max(V(out))

run maxout(mynet=dout) as maxdout;
// maxout->out = max(V(out))
```

4. Avoid the use of design file global variables, if possible. Instead, use the input variable functionality of MDL to pass parameter values to the measurement alias.

5. Use `;` at the end of each statement in a measurement alias.
6. Use the `v()` and `i()` access functions to access voltages and currents instead of just inserting the signal name itself.
7. Define the analysis to be run in the MDL control file and do not rely on the design containing the analysis definition. This is particularly important if you plan to parameterize or reuse the analysis run itself.

### Common Pitfalls

1. To avoid search failures, ensure that the start and/or stop values of the search meet the search condition(s) before running the search.
2. Use the search command when the conditions are continuous and monotonically increasing or decreasing. If the conditions are discontinuous or non-monotonic, use `mvarsearch`.
3. Run newly defined alias measurements using default settings prior to inserting them into higher level constructs such as `foreach`, `montecarlo`, `search`, or `mvarsearch` to ensure that the alias measurement does not contain errors.
4. Statements in the alias measurement block before the `run` statement are executed only once before the run statement is executed. Use this functionality to compute constant values, or testbench setup. These expressions are not evaluated at each iteration of the `run` analysis itself.
5. Ensure that variables are not forward referenced by defining them prior to their usage.
6. When specifying enumerated arguments, remember to include the single quotation mark. For example, to set `errpreset` on an MDL defined transient analysis, use `errpreset='moderate'`.

### Miscellaneous

1. If an MDL run is inadvertently terminated prior to normal termination, or the output `.measure` file is inadvertently removed, use the `processmdl` script located at `install-path/tools/mdl/bin` to recreate the `.measure` file. The syntax is as follows:  

```
processmdl [options] mdlfilename
```

This is also useful for recreating the `.measure` file from the raw directory in tabular format (`-tab`) or for changing the precision of the results (`-prec`). The initial raw directory must remain intact for this to function correctly.

2. To access online help on predefined functions, type `spectremdl -h functionname` in a terminal window (for example, `spectremdl -h cross`). For a list of predefined functions, type `spectremdl -h functions`.

# **Spectre Circuit Simulator Measurement Description Language User Guide and Reference**

---

---

## Constructing MDL Expressions

---

A Measurement Description Language (MDL) expression consists of a series of language elements that conforms to the rules of the language. This chapter defines the SpectreMDL language elements and describes the rules for combining the elements into expressions. As described in the next chapter, expressions can be used, in turn, to make measurements.

The major topics in this chapter include

- [Basic Language Elements and Scope Rules](#) on page 56
- [Data Types](#) on page 58
- [Declarations](#) on page 66
- [Operators](#) on page 69

## Basic Language Elements and Scope Rules

The basic language elements include white space, comments, and identifiers.

### White Space

The MDL tool ignores blanks, tabs, and pairs consisting of a backslash immediately followed by a new-line character, except when these characters or combinations are in strings or when they separate other language elements.

For example, in MDL, this code fragment,

```
export      real      p2p_rise=pp(trim(sig=V(out),\  
    from=0, to=100n))
```

has an effect identical to that of the following fragment.

```
export real p2p_rise=pp(trim(sig=V(out), from=0, to=100n))
```

### Comments

In MDL, you can designate a comment in either of two ways.

- An in-line comment starts with the two characters `//` (provided they are not part of a string) and ends with a new-line character. Within an in-line comment, the characters `/`, `/*`, and `*/` have no special meaning. An in-line comment can begin anywhere in the line.

```
// This code fragment contains four in-line comments.  
// Three comments affect whole lines; one is at the end of a line  
run dc // Run the analyses.  
//
```

- A block comment starts with the two characters `/*` (provided they are not part of a string) and ends with the two characters `*/`. Within a block comment, the characters `*`, `/*`, and `//` have no special meaning.

```
/*  
 * This is an example of a block comment. A block  
 * comment can continue over several lines, making it  
 * easy to add extended comments to your file.  
*/
```



## Identifiers

You use an identifier to give a unique name to an object such as a variable, a measurement alias, or an analysis name in the `run` or `run as` statement. The unique name allows you to reference the object from other places. Identifiers are case sensitive.

```
identifier ::=
    letter {letter_or_digit}

letter_or_digit ::=
    letter
    | digit

letter ::=
    a-z
    | A-Z
    | _

digit ::=
    0-9
```

For example, the following statements use identifiers that comply with this syntax.

```
real An_Identifier_Name = 15.0
real a_2nd_name = 15.0
real many__underscores = 20.
alias measurement _tran2 {
alias measurement _tran3_ {
```

The following identifier does *not* comply with this syntax.

```
real 2identifier = 15.0      // ILLEGAL! Must begin with a letter.
```

The following two identifiers are different, because their capitalization is different.

```
real rise = 14.0
real RISE = 16.0
```

## Scope Rules

The scope of an MDL variable is the measurement alias in which it is defined. For example, assume you have an MDL control file that contains the following statements:

```
alias measurement mytran1 {
    export real out_160n=V(out)@160n
}
alias measurement mytran2 {
    export real out_160n=V(out)@160n
}
run mytran1
run mytran2
```

In this example, there is no conflict between the two `out_160n` values because each is visible only within the measurement alias that defines the variable.

## Data Types

Supported data types include: numbers, enumeration names, variables, predefined constants, strings, enum, nets, terminals, arrays, and analyses.

### Numbers

MDL supports two data types for arithmetic operations: *integer numbers* and *real numbers*.

#### Integer Numbers

The syntax for an integer number is

```
integer_number ::=
    [ sign ] unsigned_num

sign ::=
    + | -

unsigned_num ::=
    decimal_digit { decimal_digit }

decimal_digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Examples of integer numbers include

```
277195000
-634           // A negative number
0005
```

#### Real Numbers

The syntax for a real number is

```
real_number ::=
    [ sign ] unsigned_num .unsigned_num
    | [ sign ] unsigned_num [.unsigned_num] e [ sign ] unsigned_num
    | [ sign ] unsigned_num [.unsigned_num] E [ sign ] unsigned_num
    | [ sign ] unsigned_num [.unsigned_num ] scale_letter

sign ::=
    + | -
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
unsigned_num ::=
    decimal_digit { decimal_digit }

decimal_digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

scale_letter ::=
    T | G | M | K | k | _ | m | u | n | p | f | a
```

scale\_letter      A scale\_letter listed in the following table. If you use scale\_letter, you must not have any white space between the number and scale\_letter. Be certain that you use the correct case for scale\_letter.

scale_letter	Scale factor
T	10 <sup>12</sup>
G	10 <sup>9</sup>
M	10 <sup>6</sup>
K	10 <sup>3</sup>
k	10 <sup>3</sup>
_	1
m	10 <sup>-3</sup>
u	10 <sup>-6</sup>
n	10 <sup>-9</sup>
p	10 <sup>-12</sup>
f	10 <sup>-15</sup>
a	10 <sup>-18</sup>

## Examples of real numbers include

```
2.5K           // 2500
1e-6           // 0.000001
1.3u
5.46M
47p
100m
50
213116.223642
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

## Complex Numbers

Complex numbers are numbers that fall on the complex plane. They consist of two real numbers, the first representing the real part and the second the imaginary part. In this release, you can use complex number declaration only to export a number of that type. For example, you can use a statement like the following one.

```
export cplx out_1u=V(out)@1u
```

Assigning a real number to a complex variable sets the real part to the real number and the imaginary part to zero. As a result, the previous statement produces output like the following.

```
out_1u          = ( 2.99983, 0 )
```

## Enumeration Names

Enumeration names consist of a single quote followed by an identifier.

The syntax for a name is

```
name ::=
    'identifier
```

Names can be used to access predefined constants and to select choices in the built-in functions.

Examples of constants include:

```
'pi
'avogadro
```

The following statement illustrates using the name 'fall in the cross function.

```
export real crossOut = cross( arg=V(out), dir='fall, n=1, thresh=1 )
```

## Predefined Constants

MDL provides the following predefined constants.

---

### Integer Constants

'yes	Boolean true	1
'no	Boolean false	0

### Real Mathematical Constants

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

'pi	$\pi$	3.14159265
'e	e	2.71828183
'inf	$\infty$	infinity
'nan	Not a number (result of an invalid operation)	NaN

### Real Physical Constants

'q	Charge of an electron	$1.6021918 \cdot 10^{-19}$ C
'c	Speed of light	$2.99792458 \cdot 10^8$ m/s
'k	Boltzmann's constant	$1.3806226 \cdot 10^{-23}$ J/K
'h	Planck's constant	$6.6260755 \cdot 10^{-34}$ J-s
'eps0	Permittivity of a vacuum	$8.85418792394420013968 \cdot 10^{-12}$ F/m
'epsrsi	Relative permittivity of silicon	11.7
'u0	Permeability of a vacuum	$\pi \times 4.0 \cdot 10^{-7}$ H/m
'celsius0	0 celsius	273.15 K
'micron		$10^{-6}$ m
'angstrom		$10^{-10}$ m
'avogadro	Avogadro's number	$6.022169 \cdot 10^{23}$
'logic0	The value of logic 0	0
'logic1	The value of logic 1	5

---

In the following example, the name 'pi corresponds to the predefined constant  $\pi$  and is automatically converted to the value  $\pi$  for the calculation.

```
export real cos2pi=cos(2*'pi) // Using pi as a parameter.
```

### enum

An enum variable can be passed as an input parameter or used as temporary storage for predefined constants or enumerated variables inside an alias measurement as illustrated by the following statements:

```
input enum outdir = 'fall
enum doubleindirection = outdir
enum mypi = 'pi
```

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

The enum variable contains a reference to a particular enumeration or constant, but does not contain the value represented by that enumeration or constant. For instance:

```
enum mypi = 'pi      //here mypi stores "'pi"
real myrealpi = 'pi  //myrealpi is 3.1415...
real myvarpi = mypi  //myvarpi is 3.1415...
export real cos1 = cos (mypi)
```

You cannot use the enum variable as the argument to an output or an export statement.

### Net

A net in the netlist for which the `V()` access function can be used. Hierarchical path of `net` such as `i0.c` is supported. `net` can be used only with the `input` qualifier. For example,

```
net in=data //data is a node in netlist
input net out=I0.vout //I0.vout is a node in the netlist
input net arrnets[]={data, q, I0.vout}
```

### Terminal

An instance terminal in the netlist for which the `I()` access function can be used. Hierarchical path of `term` such as `i0.m0:d` is supported. `term` can be used only with the `input` qualifier. For example,

```
term t1=vdd:1 //vdd:1 is a terminal in the netlist
input term t2=I0.mp0:1 //I0.mp0:1 is a terminal in the netlist
input term arrterms[]={vdd:1, I0.I1.mp0:1}
```

### Analysis

The analysis declaration statement provides a method to store an analysis defined in the netlist or created by as statement in a run statement. The `analysis` statement can be used only with the `input` qualifier.

```
analysis_declaration_statement ::=
    [ qualifier ] analysis_identifier [= initvalues ]

qualifier ::=
    input

initvalues ::=
    init_val | { value1, value2, ..., valueN }
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<i>identifier</i>	Name to be used for the analysis or analysis array variable. For example, analysis1, analysis2[ ].
<i>input</i>	Keyword to declare input data.
<i>analysis</i>	Keyword to represent the analysis type data.
<i>init_val</i>	A single initial analysis name.
<i>{value1,value2, ..., valueN}</i>	List of analyses names to initialize an array.

The following MDL control file defines an array of analyses, where at1, tran1, ag1, and tran2 are pre-defined analyses in the netlist.

```
analysis ArrAnalysis[]={at1, tran1, ag1, tran2}
alias measurement myrun {
    input analysis mytran=tran_1      //tran_1 is initial value
    run mytran
    ...
}
run ArrAnalysis[0]
run myrun (mytran= ArrAnalysis[1]) as meas1
run ArrAnalysis[2]
run myrun (mytran= ArrAnalysis[3]) as meas2
```

## Array

An array declaration statement provides a method for defining, using, storing, and outputting a vector of data. You may access this data by a 0-based index, or by passing the entire data using the array name. In addition, you can also output this data to the `.measure` file.

You can use the array declaration statement to declare a data array and indicate whether the array is used for input, export, or output.

```
array_declaration ::=
    [ MDL_qualifier ] datatype MDL_id [ = initvalues ]

MDL_qualifier ::=
    input | export | output

datatype ::=
    real | int | cplx | string | net | term | analysis

initvalues ::= {
    value1, value2,...valueN | init_val_array
```

*MDL\_id*                      Name to be used for the array. For example, arr[ ].

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<b>input</b>	Keyword to declare an array of input data.
<b>export</b>	Keyword to declare an array of export data.
<b>output</b>	Keyword to declare an array of output data.
<b>real</b>	Keyword indicating that the vector consists of real numbers.
<b>int</b>	Keyword indicating that the vector consists of integer numbers.
<b>cplx</b>	Keyword indicating that the vector consists of complex numbers.
<b>string</b>	Keyword indicating that the vector consists of strings.
<b>net</b>	Keyword indicating that the vector consists of nets.
<b>term</b>	Keyword indicating that the vector consists of instance terminals.
<b>analysis</b>	Keyword indicating that the vector consists of one or more analysis names.

*value1, value2,...valueN*

List of initial values of the array.

*init\_val\_array*

Array used to set initial values.

## Example 1

For the following MDL control file,

```
//An example of the array variable syntax.
alias measurement mytran {
    input real varr[ ] = {1.0,2.0,3.0}
    run tran(stop=160n)
    export real outvarr[ ] = varr
}

run mytran as mytran1

int i=0
// Print result
print fmt(" Default values \n") to = "print.txt"
foreach i from swp(start=0, stop=2 , step=1) {
print fmt( "varr[%V]=%V\n" ,i, mytran1->outvarr[i] )  addto="print.txt"
}

//

run mytran(varr={4.0,5.0,6.0}) as mytran2
```



# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
//Print result
print fmt(" Pass list \n") addto = "print.txt"
foreach i from swp(start=0, stop=2 , step=1) {
print fmt( "varr[%V]=%V\n" ,i, mytran2->outvarr[i] )  addto="print.txt"
}

real argarr = {7.0,8.0,9.0}
run mytran(varr=argarr) as mytran3
print fmt(" Pass array variable \n") addto = "print.txt"
foreach i from swp(start=0, stop=2 , step=1) {
print fmt( "varr[%V]=%V\n" ,i, mytran3->outvarr[i] )  addto="print.txt"
}
```

The output file, `print.txt`, looks as follows:

Default values

```
varr[0]=1
varr[1]=2
varr[2]=3
```

Pass list

```
varr[0]=4
varr[1]=5
varr[2]=6
```

Pass array variable

```
varr[0]=7
varr[1]=8
varr[2]=9
```

## Example 2

In the following example, multiple cross times of a node voltage are saved to the `.measure` file.

```
alias measurement findqcross {
    run tran (stop=200n, step=40n)
    export real outcross[]= crosses (V(q), n=2, thresh=vdd/2) }
run findqcross.
```

The `.measure` file for the above MDL control file is as follows:

```
Measurment Name: findqcross
Analysis Type    : tran
outcross[0]      = 4.07e-08
outcross[1]      = 9.017e-08
outcross[2]      = 1.207e-07
outcross[3]      = 1.702e-07
```

## Example 3

The following statement:

```
export real xOut = crosses ( sig=V(out), thresh=1 )
```

has the following result:

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
xOut[0]=0.2, xOut[1]=0.3, xOut[2]=0.5
```

You can get the maximum index in the above array by the following statement:

```
real xOutSize = max ( xval (xOut) )
```

## Example 4

The following MDL control file measures the delay on bus signals OUT[0] and OUT[1].

```
alias measurement delay {
    input net inputnets[] = {a, b}
    run tran(stop=80n)
    export real d1 = cross(V(inputnets[0]), dir='fall, n=1, thresh=vdd/2)
    export real d2 = cross(V(inputnets[1]), dir='fall, n=1, thresh=vdd/2)
}
run delay(inputnets={OUT[0], OUT[1]}) as d1
//Print results
print fmt("d1=%V, d2=%V\n", d1->d1, d1->d2) to="arr.print"
```

The .measure file for the above control file is as follows:

```
Measurement Name:  d1
Analysis Type      :  tran
d1                 =  5.0075e-08
d2                 =  4.2075e-08
```

The output file, arr.print, looks as follows:

```
d1=5.0075e-08, d2=4.2075e-08
```

## Declarations

```
variable_declaration_statement ::=
    [ qualifier ] datatype variable [= expression ] {, variable [= expression ]}
```

```
qualifier ::=
    input | export | output
```

```
datatype ::=
    real | int | cplx | string | net | term | array | enum
```

```
parameter_declaration_statement ::=
    input real parameter [= expression ] {, parameter [= expression ]}
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<i>qualifier</i>	Declares the input variables	
	<i>input</i>	Declares input variables that may be included as an argument to the alias measurement. The input variables must precede the <i>run</i> statement in an alias measurement. Although input variables can be initialized with the default value or expression (if present), the value in the parameter list in the <i>run</i> statement has higher priority to the default value.
	<i>output</i>	Declares output variables which are visible outside the alias measurement. They will not be saved to the measurement dataset, nor will they be presented in the <i>.measure</i> file. The output variables are defined and evaluated with the default value or expression (if present). If no default value is present, then they have no value (that is, 'nan').
	<i>export</i>	Declares export variables which are visible outside the alias measurement and are also written to the PSF measurement dataset and the <i>.measure</i> file. The <i>.measure</i> file name is constructed by adding the <i>.measure</i> extension to the base name of the MDL control file. The <i>.measure</i> file is placed in the same directory as the results directory. Only the numbers data type is available for export. The export variables are defined and evaluated with the default value or expression (if present). If no default value is present, then they have no value (that is, 'nan').

If you do not specify the *qualifier*, the associated parameters are considered by MDL as local variables whose value is only effective inside the alias measurement. If you calculate values that are used only in later calculations, you can omit the *qualifier* to minimize the number of expression values written to the *.measure* file.

<i>datatype</i>	The types to declare the variables. Some types you can use are:	
	<i>real</i>	Indicates a real number.
	<i>int</i>	Indicates an integer.
	<i>cplx</i>	Indicates a complex number.
	<i>string</i>	Indicates a string.

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

	<code>net</code>	Indicates a net in the netlist for which the <code>V()</code> access function can be used. <code>net</code> can be used only with the input keyword. Hierarchical path of <code>net</code> such as <code>i0.c</code> is supported.
	<code>term</code>	Indicates an instance terminal in the netlist for which the <code>I()</code> access function can be used. <code>term</code> can be used only with the input keyword. Hierarchical path of <code>term</code> such as <code>i0.m0:d</code> is supported.
	<code>array</code>	Indicates a vector of data. The data type can be integer, real, complex, string, net, term, and analysis. An array of integers, real, and complex numbers can be used with the <code>input</code> , <code>output</code> , or <code>export</code> qualifier, while an array of string, net, term, and analysis can only be used with the <code>input</code> qualifier. An array can be accessed by a 0-based index, can be passed by using the array name only, and can be initialized by a list of values with a comma in between or by an existing array.
	<code>enum</code>	Indicates an enumerated variable used as a reference to a particular enumeration or constant. It can only be used with the <code>input</code> qualifier or without a qualifier.
	<code>analysis</code>	Indicates an analysis variable.
<i>variable</i>		The variables used in the measurement aliases. You must separate multiple variables by commas. You must declare variables before you use them, but you can declare them anywhere and initialize them when they are declared. The variable name must begin with a letter. For more information, see <a href="#">Identifiers</a> on page 57.

Variables with calculated values can be used in subsequent MDL expressions. For example, you might make a complicated expression easier to read by using other expressions to calculate preliminary values.

```
real iq2c=I(i1.q2:c)
real iq2b=I(i1.q2:b)
real iq3b=I(i1.q3:b)
real iq4b=I(i1.q4:b)
export real iref = iq2c + iq2b + iq3b + iq4b
```

## Operators

The following sections describe the operators that you can use in MDL and explains how to use them to form expressions. For basic definitions, see

- [“Unary Operators”](#) on page 70
- [“Binary Operators”](#) on page 70

For information about precedence, see

- [“Operator Precedence”](#) on page 71

## Overview of Operators

An *expression* is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operators. Any legal operand is also an expression. Expressions can be used only on the right-hand side of an assignment operator.

The operators associate from left to right. That means that when operators have the same precedence, the one farthest to the left is evaluated first. In this example

`A + B - C`

the simulator does the addition before it does the subtraction.

When operators have different precedence, the operator with the highest precedence is evaluated first. In this example

`A + B / C`

the division (which has a higher precedence than addition) is evaluated before the addition. For information on precedence, see [“Operator Precedence”](#) on page 71.

You can change the order of evaluation with parentheses. If you code

`(A + B) / C`

the addition is evaluated before the division.

The operators divide into groups, according to the number of operands the operator requires. The groups are the unary operators and the binary operators.

## Unary Operators

The unary operators each require a single operand.

Operator	Definition	Type of Argument	Example
+	Unary plus	integer, real, complex	<code>val = +13 // val=13</code>
-	Unary minus	integer, real, complex	<code>val = -(4-5) // val=1</code>
!	Unary not	integer	<code>Val =!(V(out)&gt;0)</code>

## Binary Operators

The binary operators each require two operands.


Operator	Definition	Type of Argument	Example
!=	<i>a</i> not equal to <i>b</i> ; evaluates to 0 or 1	real, integer	<code>I = 5.2 != 5.2 // I=0</code>
*	<i>a</i> multiplied by <i>b</i>	real, complex, integer	<code>R = 2.2 * 2 // R=4.4</code>
+	<i>a</i> plus <i>b</i>	real, complex, integer	<code>R = 10.0 + 3.1 // R=13.1</code>
-	<i>a</i> minus <i>b</i>	real, complex, integer	<code>I = 10 - 13 // I= -3</code>
/	<i>a</i> divided by <i>b</i>	real, complex, integer	<code>I = 9/4 // I=2</code>
<	<i>a</i> less than <i>b</i> ; evaluates to 0 or 1	real, integer	<code>I = 5 &lt; 7 // I=1</code>
<=	<i>a</i> less than or equal to <i>b</i> ; evaluates to 0 or 1	real, integer	<code>I = 5.0 &lt;= 5.0 // I=1</code>
==	<i>a</i> equal to <i>b</i> ; evaluates to 0 or 1	real, integer	<code>I = 5.2 == 5.2 // I=1</code>
>	<i>a</i> greater than <i>b</i> ; evaluates to 0 or 1	real, integer	<code>I = 5 &gt; 7 // I=0</code>
>=	<i>a</i> greater than or equal to <i>b</i> ; evaluates to 0 or 1	real, integer	<code>I = 5 &gt;= 7 // I=0</code>

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

Operator	Definition	Type of Argument	Example
@	Event operator. Interpolates a signal at a particular X-axis value (abscissa).	real, complex	V(out) @ 1u  I(R1) @ cross( sig=V(out), n=1, dir='rise', thresh=1.5 )
&&	Logical AND; evaluates to 0 or 1	integer	I=(1==1)&&(2==2) // I=1 I=13&&1 // I=1
	Logical OR; evaluates to 0 or 1	integer	I=(1==2)    (2==2) // I=1 I=13    0 // I=1

## Operator Precedence

The following table summarizes the precedence information for the operators.

Operator	Precedence
+ - (unary)	Highest precedence
@	
* /	
+ - (binary)	
< <= > >=	
== !=	
&&	
	Lowest precedence

# **Spectre Circuit Simulator Measurement Description Language User Guide and Reference**

---



---

## Running MDL in Batch Mode

---

This chapter describes the syntax and options for the `spectremdl` command, which runs the Measurement Description Language (MDL) tool. The `spectremdl` command can now be used for design files written in both Spectre and SPICE languages.

## spectremdl

Runs MDL on design files written in the Spectre language.

In the following syntax, the vertical bar ( | ) separates alternatives.

### Syntax

```
spectremdl
  [options] [-batch] MDL_file
  | -batch MDL_file -design design_file
  | -usage
  | -h function_name
options ::= =
  spectre_options
  | -measure output_file
  | -mt0
  | -tab
  | -prec 'format'
  | -rmrawfiles
  | -eng numdigits
```

### Arguments

<b>-batch</b>	A command line option used to specify the MDL control file. Notice that you can omit this option and enter simply the name of the MDL control file when the base name of the MDL control file is the same as that of design file. You must specify both -batch and -design arguments if the base names are different.
<i>MDL_file</i>	The path and filename of the MDL control file to be used. You must specify the control file.
<b>-design</b>	A command line option used to specify the design file.
<i>design_file</i>	The path and filename of the design file to be simulated. If the -design option is omitted, MDL looks for a design file with the same basename as the <i>MDL_file</i> name, but with an extension of .scs or .ckt.
<b>-usage</b>	A command line option used to display syntax information for the spectremdl command.
<i>-h function_name</i>	A command line option used to display online help on predefined functions. For example, spectremdl -h cross. For a list of

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

	predefined functions, type <code>spectremdl -h</code> functions in a terminal window.
<code>spectre_options</code>	Spectre options to be passed to the Spectre simulator. For a list of the options you can use, see <code>spectre -h</code> or <a href="#">Chapter 2, “Spectre Command Options,”</a> in <i>Spectre Circuit Simulator Reference</i> .
<b>-measure</b>	A command line option used to specify the output file. If you do not use this option, the output of the measurements is placed in a file with the same base name as the <code>MDL_file</code> and with the extension <code>.measure</code> . For example, if the <code>MDL_file</code> is <code>amp.mdl</code> , the output file, by default, has the name <code>amp.measure</code> . This default file is placed in the directory that holds the <code>design_file</code> . So if the design file is <code>./d2/arith.ckt</code> , the default measure file is <code>./d2/amp.measure</code> .
<code>output_file</code>	The path and file to be used for output data generated by measurements.
<b>-mt 0</b>	A command line option used to generate <code>.mt*</code> format data files as well as the default <code>.measure</code> file.
<b>-tab</b>	A command line option used to present data in a tabular format in the <code>.measure</code> file. This is useful for swept data.
<b>-nosort</b>	A command line option to specify that variables in the measure file should not be sorted. When this option is specified, the exported variables in the mdl file appear in the <code>.measure</code> file in the order in which they are specified in the netlist. Note that by default, they appear in alphabetically sorted order in <code>.measure</code> file.
<b>-prec 'format'</b>	A command line option used to specify the number of significant digits to be displayed in the signal value in the <code>.measure</code> file. For example, <code>'%.15g'</code> displays 15 significant digits for the measured value.
<b>-rmrawfiles</b>	A command line option used to specify that the raw directory, including the MDL measure results, be deleted after the <code>.measure</code> output file is created.

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

**-eng** *numdigits*      A command line option to specify the engineering format of the signal value output to the measurement file. *numdigits* is the number of significant digits of the signal value. If *numdigits* is not specified, 6 significant digits are displayed. If both **-prec** and **-eng** arguments are specified, **-prec** is ignored.

## Examples

The following command creates a `.measure` file called `amp.measure` in the directory where you run the command,

```
spectremdl -batch amp.mdl -design amp.scs
```

You can simplify the above command because the MDL control file and the design file have the same basename. The equivalent simpler command is

```
spectremdl amp.mdl
```

You need the options when the MDL control file and the design file have different base names or when the design file has a suffix other than `.scs` or `.ckt`.

For example,

```
spectremdl -batch control.mdl -design topnetlist.scs
spectremdl -batch control.mdl -design netlist.sp
```

The following command creates a measurement result file called `mdlresults` in your home directory.

```
spectremdl amp.mdl -measure $HOME/mdlresults
```

The following command creates a `.measure` file in the directory where the netlist is located but places the database in the `./test26/amp.raw` directory.

```
spectremdl netlist/amp.mdl -raw ./test26/amp.raw
spectremdl -batch amp.mdl -design ./netlist/amp.scs -raw ./test26/amp.raw
```

The following command presents data in a tabular format.

```
spectremdl -tab -batch foreach.mdl -design dflip.scs
```

results in the following `.measure` file (called `foreach.measure`)

```
Exported variables from PSF results directory:  dflip.raw
```

```
date           : 9:54:01 AM, Tue May 10, 2005
design          : * DFF
simulator      : spectre
```

```
Swept Measurements :
Measurement Name   : findqcross
Analysis Type      : tran
```

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

temp	vdd	clk_q_delay
25	1.5	1.91866e-10
25	1.8	1.61376e-10
25	2	1.5053e-10
50	1.5	1.96657e-10
50	1.8	1.66472e-10
50	2	1.55428e-10
75	1.5	2.01521e-10
75	1.8	1.71556e-10
75	2	1.60476e-10
100	1.5	2.0633e-10
100	1.8	1.76775e-10
100	2	1.65437e-10

Without the `-tab` option, the same command results in the following `.measure` file

Exported variables from PSF results directory: `dflip.raw`

```
date           : 10:20:35 AM, Tue May 10, 2005
design          : * DFF
simulator      : spectre

Swept Measurements :
Measurement Name   : findqcross
Analysis Type      : tran
clk_q_delay        temp @ 25
                   vdd @ 1.5      = 1.91866e-10
clk_q_delay        temp @ 25
                   vdd @ 1.8      = 1.61376e-10
clk_q_delay        temp @ 25
                   vdd @ 2        = 1.5053e-10
clk_q_delay        temp @ 50
                   vdd @ 1.5      = 1.96657e-10
clk_q_delay        temp @ 50
                   vdd @ 1.8      = 1.66472e-10
clk_q_delay        temp @ 50
                   vdd @ 2        = 1.55428e-10
clk_q_delay        temp @ 75
                   vdd @ 1.5      = 2.01521e-10
clk_q_delay        temp @ 75
                   vdd @ 1.8      = 1.71556e-10
clk_q_delay        temp @ 75
                   vdd @ 2        = 1.60476e-10
clk_q_delay        temp @ 100
                   vdd @ 1.5      = 2.0633e-10
clk_q_delay        temp @ 100
                   vdd @ 1.8      = 1.76775e-10
clk_q_delay        temp @ 100
                   vdd @ 2        = 1.65437e-10
```



---

## Running MDL in Post-processing Mode

---

MDL supports the post-processing mode that enables the user to evaluate measurements after the simulation has completed. This is especially helpful if you would like to use the same language (MDL or `.measure` statements) that was used for measurements evaluated during the simulation.

In case the measurements were not set up correctly, or more measurements need to be evaluated, the simulation needs to be run repeatedly to get the desired results. With post-processing capability, you can edit the measurements (in a MDL file, or `.measure` in the netlist) and then invoke the simulator in a mode, where instead of performing the actual simulation, it reads the simulation results from a specified results directory or file, depending upon the waveform format being used.

The MDL post-processing mode allows you to execute a MDL script on an existing results database. Optionally a netlist can be provided. This allows the MDL script to reference objects in the netlist. For example the parameters in the netlist could be used in the MDL measurements.

## mdl

Runs MDL in post-processing mode.

In the following syntax, the vertical bar ( | ) separates alternatives.

### Syntax

```
mdl -batch|-b <file.mdl> [-design|-d netlist] -raw|-r <rawdir> [options]
      mdl -d <netlist containing .measure statement> -raw|-r <rawdir> [options]
```

### Arguments

-batch -b <file.mdl>	The filename of the MDL file to be executed.
-raw -r <rawdir>	Location of the results directory.
-design -d <netlist>	The filename of the netlist to be loaded by MDL. This can also be a netlist containing <code>.measure</code> statements.
-measure -m <file.measure>	The default output filename is taken from the basename of the design argument, and appended with the <code>.measure</code> extension. This option creates an output file with the specified name. If an absolute path is not specified, the output file is created in the directory of the design argument.
+log +l <logfile>	Copies all messages to `logfile'.
=log =l <logfile>	Sends all messages to `logfile'.
-prec -p '<format>'	Optional argument to specify the precision of the measured value output in the measurement file.  <b>Example:</b> <code>%.15g</code> will output the measured values to 15 significant digits. The argument is ignored if <code>-engineering</code> argument is used.
-eng -e <numdigits>	Optional argument to specify the engineering format of the signal value output to the measurement file. <numdig> is a number of significant digits of the signal value. <numdig> is optional. If <numdig> is omitted then value of the <numdig> is 6. <code>-prec</code> argument is ignored.
-tab -t	Optional argument to specify that the measure file should be displayed in tabular format.



## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<code>-append -a</code>	Optional argument to specify that the measure file should be opened in append mode.
<code>-nosort -n</code>	Optional argument to specify that variables in the measure file should not be sorted.
<code>-outdir -o</code>	Optional argument to specify an alternate output directory location for all output files. This does not change the location of the raw directory if explicitly specified with the <code>-raw</code> option.
<code>-warn -w</code>	Optional argument to issue warning for ignoring unsupported constructs. Default is to issue an error.

Most of the options are consistent with previous versions of `spectremdl` executable. However, the `-warn` option allows unsupported constructs (`search`, `mvarsearch`) to be ignored and evaluation to proceed.

### Examples

Assume that an MDL script, `test.mdl` was used to generate a raw directory, `input.raw` using Spectre.

```
$ spectremdl -batch test.mdl -design input.scs -raw input.raw
```

To rerun the script on the results, the following command will be used:

```
$ mdl -b test.mdl -r input.raw
```

In this case, the measurement results are written out to the `test.measure` file. The `-m` command-line option can override the name of the output file.

```
$ mdl -b test.mdl -r input.raw -m input.measure
```

If the netlist `input.scs` was used to generate the results, it is possible that the MDL file references parameters from the netlist. Without the netlist, the MDL measurements cannot be evaluated, hence the netlist must be provided on the command-line so that those parameters and their values can be found.

```
$ mdl -b test.mdl -d input.scs -r input.raw
```

Another usage is for users who do not have MDL script, but instead use `.MEASURE` statements in the netlist (or in a file included in the netlist). Here `test.sp` may either contain the complete design or may have just `.measure` statements.

```
$ mdl -d test.sp -r test.raw
```

```
.include "param.sp" // optional parameter definition
.include "design.sp" //optional design
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
.include "test.msr" // contains measurements
```

```
$ mdl -d test.msr -r test.raw
```

Here `test.msr` may contains some parameter definition of the parameters referenced in the `.measure` statement. There is no need for complete design.

```
.param VDD=1.8
.meas tran tfr trig v(a) val='0.5*VDD' fall=1 targ v(y) val='0.5*VDD' rise=1
.measure tran tranmaxout1 max v(out)
.measure tran tranavg avg v(out)
```

Some waveform formats do not use the result directory concept. Cadence formats (such as PSF or SST), when generated from Spectre, APS, UltraSim, etc place a logFile in the results directory that is used to associate datasets with the physical files containing the simulation data. For formats that do not support this, it is necessary to explicitly specify the file contains the data rather than the results directory itself.

An example using the FSDB format is:

```
$ mdl -d tran.msr -r input.raw/tran.fsdb
```

In this example, `tran.msr` contains the measurements for the specified transient analysis results file, `input.raw/tran.fsdb`. The content of `tran.msr` could be

```
.measure tran tranmaxout1 max v(out)
.measure tran tranavg avg v(out)
```

In another example, the some measurements are evaluated using a DC analysis results file.

```
$ mdl -d dc.msr -r input.raw/dc.fsdb
```

Here, `dc.msr` contains the measurements to be evaluated, as in

```
.measure dc dcnmaxout1 max v(out)
.measure dc dcavg avg v(out)
```

## Limitations

There are limitations when using this mdl tool to execute an MDL script.

The primary limitation that exists with the post-processing flow is a result of the nature of this flow. It can only work with the data that is in the netlist, MDL file and results database. Hence, if a signal that is used in an MDL expression is not saved in the database, then the expression will fail to evaluate.

For example, when the netlist is used, device input parameters can be used in expressions; however oppoint parameters cannot be used, unless they were already saved.

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

Other restrictions are:

- Other consequences of being only able to use information that is readily available in the MDL script, netlist (if supplied) and results database are the following;
  - ❑ The `mvarsearch` and `search` statements are not currently supported. When these are seen, mdl prints an error message and ignores these statements.
  - ❑ For montecarlo, the parameter `savefamilyplots=yes` must be set during the simulation run. Else, the waveform data for each iteration of the montecarlo run will not be saved.
    - Re-elaboration is not supported. Hence, if the measurement references the process or mismatch parameters, mdl will only see the nominal parameter values.
  - ❑ Foreach and montecarlo are only supported for PSF format.
  - ❑ Result dataset generated from `alter/sweep` specified in the netlist have different naming convention. Hence, if a netlist contains `alter/sweep` then dataset will not be located. It is recommended to use MDL scripts for `alter/sweep` so that corresponding dataset can be found.
- The MDL post-processing flow works on waveforms rather than on each individual point of a signal as occurs in the SpectreMDL flow. There are following consequences to this:
  - ❑ An `if` statement in a measurement alias can behave differently if the condition depends upon a signal value. In SpectreMDL, this could result in the true or false block being executed multiple times for each datapoint on the signal. In the post-processing flow, the condition is evaluated only once for the complete waveform.
- If there is a sweep contained in the netlist, the dataset generated by Spectre is different from the dataset generated by MDL. Hence, MDL will not be able to locate the correct dataset. This is why currently, the `.measure` flow is supported for a netlist that does not contain sweep in the netlist.

# **Spectre Circuit Simulator Measurement Description Language User Guide and Reference**

---

---

## Built-In Functions

---

The built-in functions support two syntaxes:

- Positional syntax

Requires each optional parameter up to and including the last optional parameter entered, but beyond that everything can be omitted.

```
cross( sig[, dir[, n[, thresh[, start[, xtol[, ytol[, accuracy]]]]]] )
```

- Named syntax

Allows any optional parameter to be specified — the preceding optional parameters need not be specified.

```
cross( sig=sig [, dir=dir] [, n=n] [, thresh=thresh]  
      [, start=start] [, xtol=xtol] [, ytol=ytol]  
      [, accuracy=accuracy] )
```

For example, the following statements are equivalent.

```
export real crossOut = cross(V(out), 'fall, 1, 1 )  
export real crossOut = cross( sig=V(out), dir='fall, n=1, thresh=1 )
```

## **abs**

Returns the absolute value of a signal.

### **Syntax**

```
abs ( arg )  
abs ( arg=arg )
```

### **Arguments**

*arg*                                      The scalar or signal.

### **Example**

```
export real myabs = abs ( -5 )
```

returns

```
myabs = 5  
export real outabs = abs(arg=V(out))@1m
```

returns the value of the signal `V(out)` at 1ms.

## **acos**

Returns the arc cosine of a signal.

### **Syntax**

```
acos( arg )  
acos( arg=arg )
```

### **Arguments**

*arg*                                      The scalar or signal.

### **Example**

```
export real myacos = acos( 1)  
  
returns  
myacos = 0
```

### **acosh**

Returns the hyperbolic arc cosine of a signal.

#### **Syntax**

```
acosh( arg )  
acosh( arg=arg )
```

#### **Arguments**

*arg*                                      The scalar or signal.

#### **Example**

```
export real myacosh = acosh( 1)
```

returns

```
myacosh = 0
```



## analstop

Returns the simulation stop value.

### Syntax

```
analstop( )
```

### Arguments

None

### Example1

#### Used in MDL File

```
alias measurement transient {  
    run tran( step=1e-12, pstep=1e-12, stop=2e-08 )  
    export real anal_stop= analstop()  
}  
run transient
```

#### returns

```
anal_stop = 2e-08
```

### Example2

#### Used in assert Statement

```
check_full_simu_time assert message="CHECK_FULL_SIMU_TIME"  
expr="full_simu_time=analstop();  
full_simu_time>10n"
```

## angle

Returns the angle of a real or complex number, or a waveform in degrees.

### Syntax

```
angle( arg )  
angle( arg=arg )
```

### Arguments

*arg*                                      The real or complex number, or a waveform.

### Example 1

```
export real myangle = angle( cplx(1,2) )
```

returns

```
myangle = 63.43
```

### Example 2

```
export real phasemargin = angle( s(2,1) ) @ ft
```

returns

```
phasemargin= 15.0369
```

## argmax

Returns the X value corresponding to the maximum Y value of a signal. If multiple X values are returned, the first one is used.

### Syntax

```
argmax( sig )  
argmax( sig=sig )
```

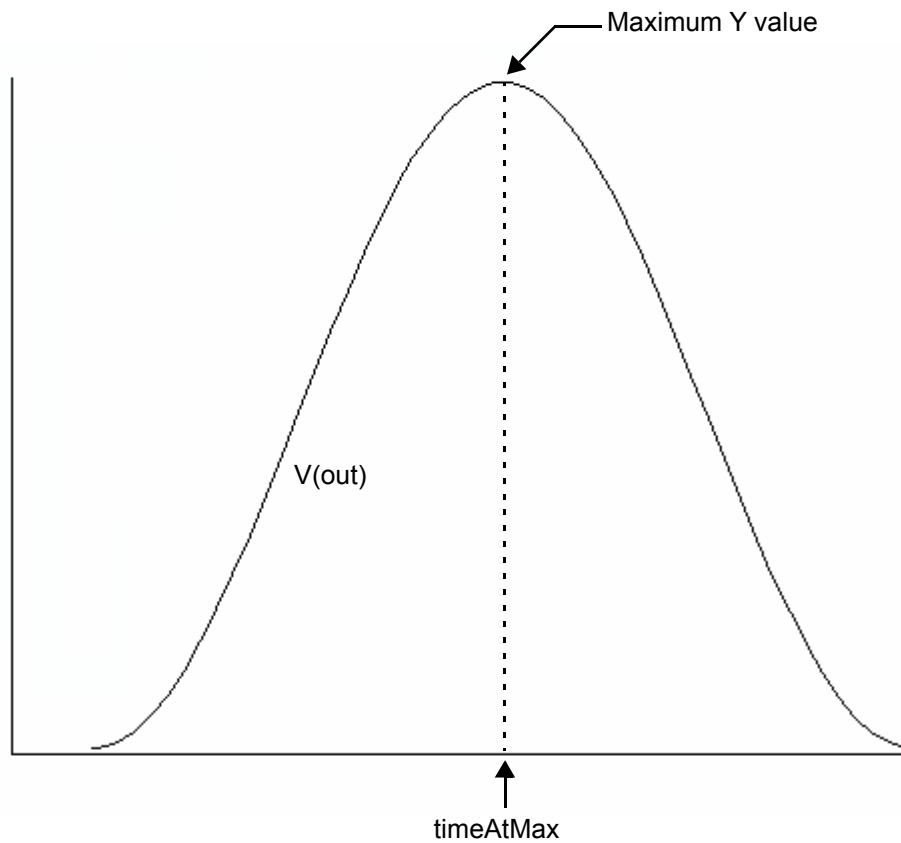
### Arguments

<i>sig</i>	The signal.
------------	-------------

### Example

```
export real timeAtMax = argmax( V(out) )
```

The following diagram illustrates how the result is determined.



## argmin

Returns the X value corresponding to the minimum Y value of a signal. If multiple X values are returned, the first one is used.

### Syntax

```
argmin( arg )  
argmin( arg=arg )
```

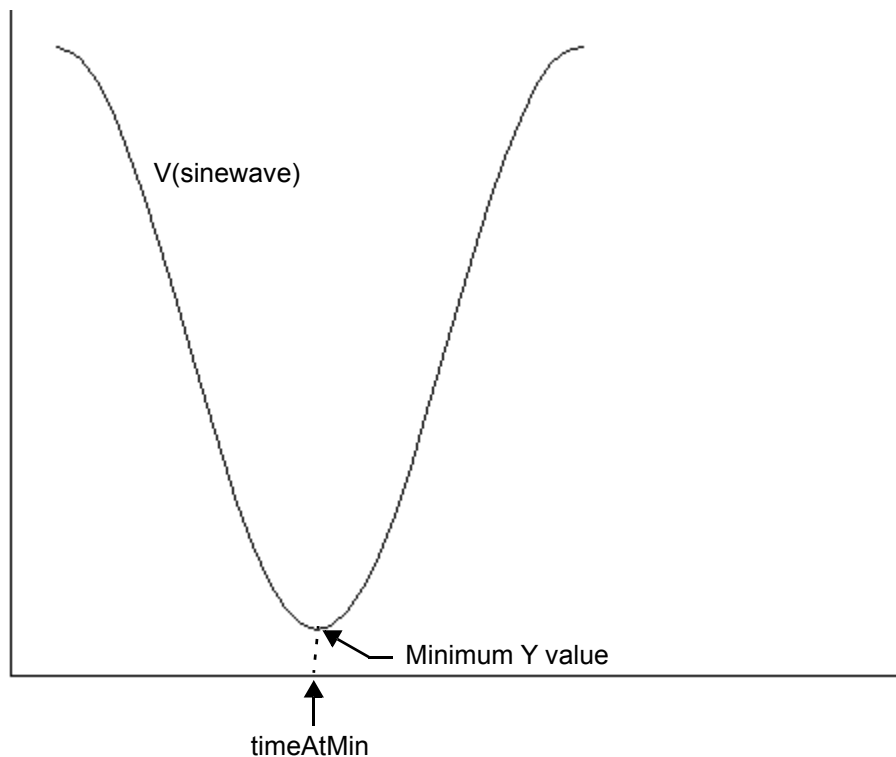
### Arguments

*arg*                                      The signal.

### Example

```
export real timeAtMin = argmin( V(sinewave) )
```

The following diagram illustrates how the result is determined.



## asin

Returns the arc sine of a signal.

### Syntax

```
asin( arg )  
asin( arg=arg )
```

### Arguments

*arg*                                      The scalar or signal.

### Example

```
export real myasin = asin( 1 )
```

returns

```
myasin = 1.57
```

## asinh

Returns the hyperbolic arc sine of a signal.

### Syntax

```
asinh( arg )  
asinh( arg=arg )
```

### Arguments

<i>arg</i>	The scalar or signal.
------------	-----------------------

### Example

```
export real myasinh = asinh( 1 )
```

returns

```
myasinh = 0.88
```

## atan

Returns the arc tangent of a signal.

### Syntax

```
atan( arg )  
atan( arg=arg )
```

### Arguments

<i>arg</i>	The scalar or signal.
------------	-----------------------

### Example

```
export real myatan = atan( 1 )
```

returns

```
myatan = 1.56
```



## **atanh**

Returns the hyperbolic arc tangent of a signal.

### **Syntax**

```
atanh( arg )
```

```
atanh( arg=arg )
```

### **Arguments**

<i>arg</i>	The scalar or signal.
------------	-----------------------

### avg

Returns the average value of a signal.

### Syntax

```
avg( arg )  
avg( arg=arg )
```

### Arguments

<i>arg</i>	The signal.
------------	-------------

### Example

```
export real myavg = avg( V(out) )
```

## avgdev

Returns the mean absolute deviation of a scalar argument or waveform. The mean absolute deviation is defined as follows:

$$1/N * ( |X1-mean| + |X2-mean| + \dots + |XN-mean| )$$

where  $|$  is the absolute value of the difference and  $N$  is the total number of Samples.

## Syntax

**avgdev** ( *arg* )

**avgdev** ( **arg**=*arg* )

## Arguments

*arg*                                      The scalar argument or waveform.

## bw (bandwidth)

Calculates the bandwidth of a waveform.

### Syntax

```
bw( sig, response, db, max )
```

```
bw( sig=sig, response=response, db=db, max=max )
```

### Arguments

*sig* The signal. In the SKILL mode, Virtuoso Visualization and Analysis XL wraps the signal with the `mag` function. In the MDL mode, you need to wrap the signal name with the `mag` function, otherwise Virtuoso Visualization and Analysis XL returns an error.

*response* The response type:

When `'low`, computes the low-pass bandwidth by determining the smallest frequency at which the magnitude of the input waveform drops *db* decibels below the DC gain.

When `'high`, computes the high-pass bandwidth by determining the largest frequency at which the magnitude of the input waveform drops *db* decibels below the gain at the highest frequency in the response waveform.

When `'band`, computes the band-pass bandwidth by:

1. Determining the lowest frequency ( $f_{max}$ ) at which the magnitude of the input waveform is maximized;
2. Determining the highest frequency less than  $f_{max}$  at which the input waveform magnitude drops *db* decibels below the maximum;
3. Determining the lowest frequency greater than  $f_{max}$  at which the input waveform magnitude drops *db* decibels below the maximum;
4. Subtracting the value returned by step 2 from the value returned by step 3. The value returned by step 2 or step 3 must exist.

Valid values: `'low`, `'high`, `'band`

Default: `'low`

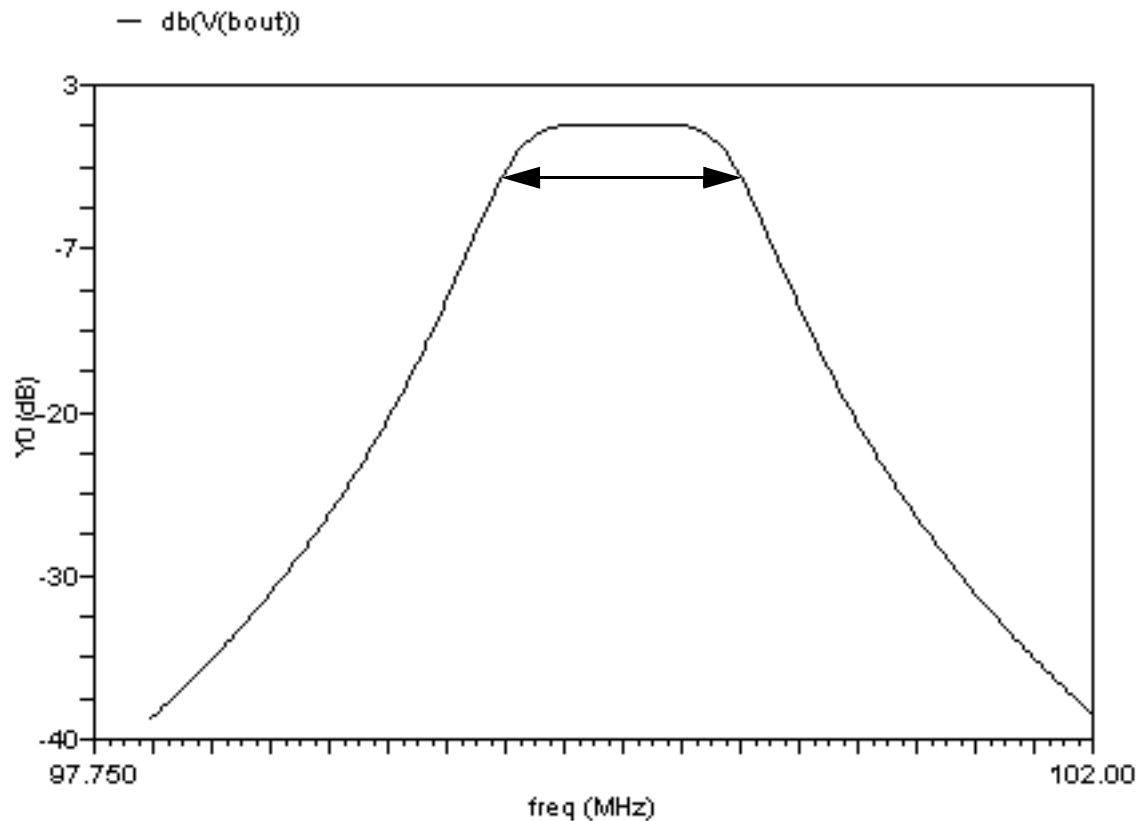
## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<i>db</i>	The decibels down from the peak. In the SKILL mode, <i>db</i> is a number equal to or greater than zero. In the MDL mode, <i>db</i> is a number less than zero. Default: -3.01029995664
<i>max</i>	The maximum amplitude of the waveform. You only need to specify this if you want to set the maximum amplitude lower than the waveform's maximum value. If the <i>max</i> is specified, Measurement Description Language (MDL) uses this value instead of computing a value.

### Example

Assume you have the following signal.



Then the following statement

```
export real bwOut = bw(mag(V(bout)), response='band)
```

generates, at the default *db* value of -3, the bw value

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

1000004.1627941281Hz

Note that the output in MDL includes the unit (Hz in the above example), whereas in SKILL it does not.

This value (approximately 1MHz) is illustrated on the graph by the double-ended arrow.

## ceil

Rounds a real number up to the closest integer value.

### Syntax

```
ceil( arg )  
ceil( arg=arg )
```

### Arguments

<i>arg</i>	The real number.
------------	------------------

### Example

```
export real myceil = ceil( 1.6 )
```

returns

```
myceil = 2
```

## cfft

Performs a Fast Fourier Transform on a complex time domain waveform and returns its frequency spectrum. The `cfft` function takes two time signals that in combination form a complex input signal.

### Syntax

```
cfft( sig_re, sig_im, from, to, numPoints[, window ])
```

```
cfft( sig_re=sig_re, sig_im=sig_im, from=from, to=to, numPoints=numPoints  
[, window=window ])
```

### Arguments

<i>sig_re</i>	The real part of the signal.
<i>sig_im</i>	The imaginary part of the signal.
<i>from</i>	The starting X value.
<i>to</i>	The ending Y value.
<i>numPoints</i>	The number of data points to be used for calculating the cfft. If this number is not a power of 2, it is automatically raised to the next higher power of 2.
<i>window</i>	The algorithm used for calculating the cfft. In this release only one algorithm is supported. Valid value: 'rectangular' Default: 'rectangular'



## clip

Returns the portion of a signal between two points along the Y-axis.

### Syntax

```
clip( sig, from, to )  
clip( sig=sig, from=from, to=to )
```

### Arguments

<i>sig</i>	The signal.
<i>from</i>	The starting point on the Y-axis.
<i>to</i>	The ending point on the Y-axis.

### Example 1

The following example works in an MDL control file.

```
export real clipOut = avg ( clip (sig=V(sinewave), from=0, to=2.5) )
```

### Example 2

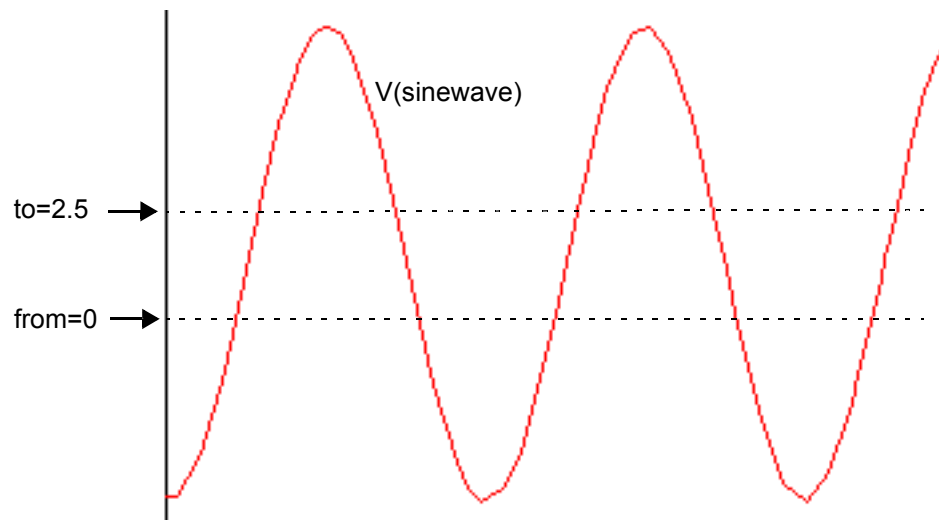
In Virtuoso Visualization and Analysis XL,

```
clip (sig=V(sinewave), from=0, to=2.5)
```

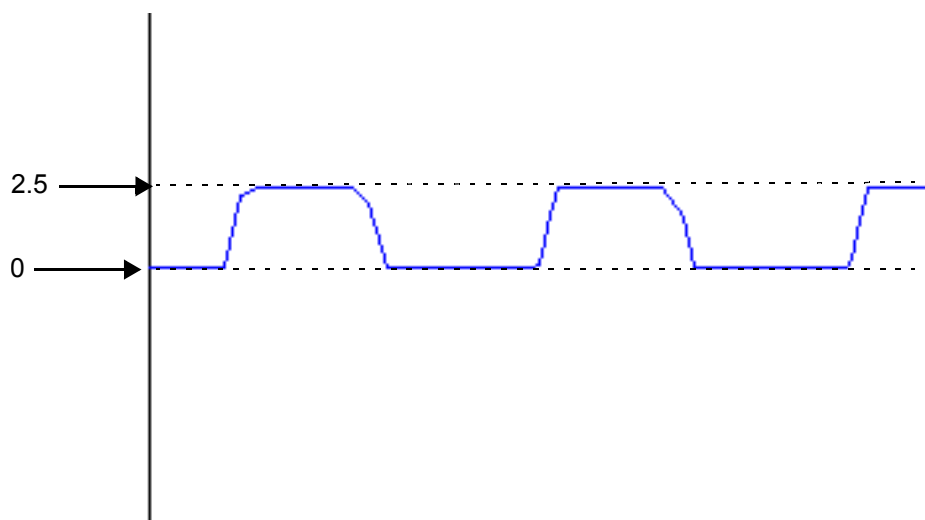
## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

transforms the following input signal



into the following output signal.



## conj

Returns the conjugate of a complex number.

### Syntax

```
conj( arg )  
conj( arg=arg )
```

### Arguments

<i>arg</i>	The complex number.
------------	---------------------

### Example

```
export cplx mycplx = cplx ( 1,2 )  
export cplx conj_mycplx = conj ( mycplx )
```

### returns

```
mycplx = (1,2)  
conj_mycplx = (1,-2)
```

## convolve

Returns a waveform consisting of the time domain convolution of two signals. This function is available in Virtuoso Visualization and Analysis XL only.

### Syntax

```
convolve( sig1, sig2[, n_interp_steps ] )  
clip( sig1=sig1, sig2=sig2[, n_interp_steps=n_interp_steps ] )
```

### Arguments

<i>sig1</i>	The first signal.
<i>sig2</i>	The second signal.
<i>n_interp_steps</i>	Number of steps for interpolating waveforms.

### Equation

Convolution is defined by the following equation:

$$\int\limits_{from}^{to} f1(s)f2(t-s)ds$$

### Example

```
real vcdelay[]=crosses(sig=V(clock), thresh=0.9, dir='rise, n=1)  
real outcross[]=crosses(V(q), n=6, thresh=vdd/2)  
export real myconv[] = convolve(vcdelay,outcross,5)
```

### returns

```
myconv[00]      = 2.57108e-13  
myconv[01]      = 2.02971e-13  
myconv[02]      = 1.74678e-13  
myconv[03]      = 1.81539e-13  
myconv[04]      = 2.03094e-13  
myconv[05]      = 2.3083e-13  
myconv[06]      = 2.65196e-13  
myconv[07]      = 3.06643e-13  
myconv[08]      = 3.55544e-13
```

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
myconv[09]      = 3.84546e-13
myconv[10]      = 3.94928e-13
myconv[11]      = 3.96003e-13
myconv[12]      = 3.88035e-13
myconv[13]      = 3.70677e-13
myconv[14]      = 3.43269e-13
myconv[15]      = 3.05072e-13
```

### **COS**

Returns the cosine of a signal.

### **Syntax**

```
cos( arg )  
cos( arg=arg )
```

### **Arguments**

*arg*                                      The scalar or signal.

### **Example**

```
export real mycos = cos( 1 )  
  
returns  
mycos = 0.54
```

## cosh

Returns the hyperbolic cosine of a signal.

### Syntax

```
cosh( arg )  
cosh( arg=arg )
```

### Arguments

*arg*                                      The scalar or signal.

### Example

```
export real mycosh = cosh( 1 )
```

returns

```
mycosh = 1.54
```

## **cplx**

Returns a complex number created from two real arguments.

### **Syntax**

```
cplx( R [, I] )  
cplx( R=R [, I=I] )
```

### **Arguments**

<i>R</i>	The value representing the real part.
<i>I</i>	The value representing the imaginary part.

### **Example**

```
export cplx mycplx = cplx( 1,2 )
```

#### **returns**

```
mycplx = (1,2)
```



## cross

Returns the X value where a signal crosses the threshold Y value.

### Syntax

```
cross( sig[, dir[, n[, thresh[, start[, xtol[, ytol[, accuracy]]]]]] )
```

```
cross( sig=sig[, dir=dir] [, n=n] [, thresh=thresh] [, start=start] [,  
      xtol=xtol] [, ytol=ytol] [, accuracy=accuracy] )
```

### Arguments

<i>sig</i>	The signal.
<i>dir</i>	The direction of the crossing event. 'rise directs the function to look for crossings where the Y value is increasing, 'fall for crossings where the Y value is decreasing, and 'cross for crossings in either direction. Valid values: 'cross, 'rise, 'fall Default: 'cross
<i>n</i>	The occurrence of the crossing. The first crossing is n=1, the second crossing is n=2, and so on. The value of n can be negative numbers: n=-1 for the last occurrence before the end of the waveform, n=-2 for the second-last occurrence before the end of the waveform, and so on. Default: 1
<i>thresh</i>	The threshold to be crossed. Default: 0
<i>start</i>	The time at which the function is enabled. Default: 0
<i>xtol</i>	The relative tolerance in percentage value in the X direction. Default: 1
<i>ytol</i>	The relative tolerance in percentage value in the Y direction. Default: 1
<i>accuracy</i>	Specifies whether the function should use interpolation, or use iteration controlled by the absolute tolerances to calculate the

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

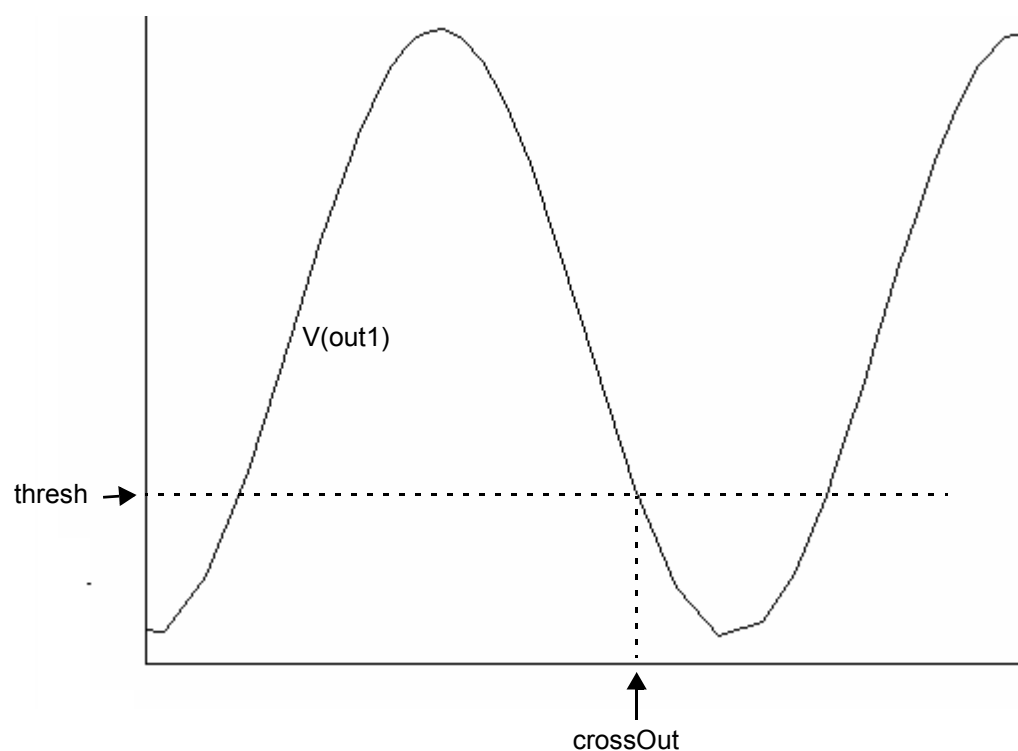
---

value. 'interp directs the function to use interpolation, and 'exact directs the function to consider the xtol and yval values.  
Data types: name for scalar  
Valid values: 'interp, 'exact  
Default: 'exact

### Example

```
export real crossOut = cross( sig=V(out), dir='fall', n=1, thresh=1 )
```

The following diagram illustrates how the result is determined.



### crosscorr

Returns the cross correlation of the specified signals. This function is available only in Virtuoso Visualization and Analysis XL.

When the input signals are double waveforms,

```
crosscorr(sig1, sig2) = convolve (sig1, flip(sig2))
```

When one of the input signals is a complex waveform (*sig2* in the following case),

```
crosscorr(sig1, sig2) = convolve (sig1, flip(conj(sig2)))
```

### Syntax

```
crosscorr( sig1, sig2[, n_interp_steps ])
```

```
crosscorr( sig1=sig1, sig2=sig2[, n_interp_steps=n_interp_steps ])
```

### Arguments

<i>sig1</i>	The first signal.
<i>sig2</i>	The second signal.
<i>n_interp_steps</i>	Number of steps for interpolating waveforms.

## crosses

Returns the X values where a signal crosses the threshold Y value.

### Syntax

```
crosses( sig[, dir[, n[, thresh[, start[, xtol[, ytol[, accuracy]]]]]] )
```

```
crosses( sig=sig [, dir=dir] [, n=n] [, thresh=thresh] [, start=start] [,  
        xtol=xtol] [, ytol=ytol] [, accuracy=accuracy] )
```

### Arguments

<i>sig</i>	The signal.
<i>dir</i>	The direction of the crossing event. 'rise directs the function to look for crossings where the Y value is increasing, 'fall for crossings where the Y value is decreasing, and 'cross for crossings in either direction. Valid values: 'cross, 'rise, 'fall Default: 'cross
<i>n</i>	The occurrence of the crossing. If n=1, the function returns the first crossing and all subsequent crossings. If n=3, the function returns the third crossing and all subsequent crossings. The value of n can be negative numbers: if n=-2, only the last two crossings are returned. Default: 1
<i>thresh</i>	The threshold to be crossed. Default: 0
<i>start</i>	The time at which the function is enabled. Default: 0
<i>xtol</i>	The relative tolerance in percentage value in the X direction. Default: 1
<i>ytol</i>	The relative tolerance in percentage value in the Y direction. Default: 1
<i>accuracy</i>	Specifies whether the function should use interpolation, or use iteration controlled by the absolute tolerances to calculate the

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

value. 'interp directs the function to use interpolation, and 'exact directs the function to consider the xtol and yval values.

Data types: name for scalar

Valid values: 'interp, 'exact

Default: 'exact

### Example

```
export real crossesOut[] = crosses( sig=V(out), dir='rise', thresh=0.0 )
```

returns

```
crossesOut[0] = 1e-05
```

```
crossesOut[1] = 2e-05
```

```
crossesOut[2] = 3e-05
```

```
crossesOut[3] = 4e-05
```

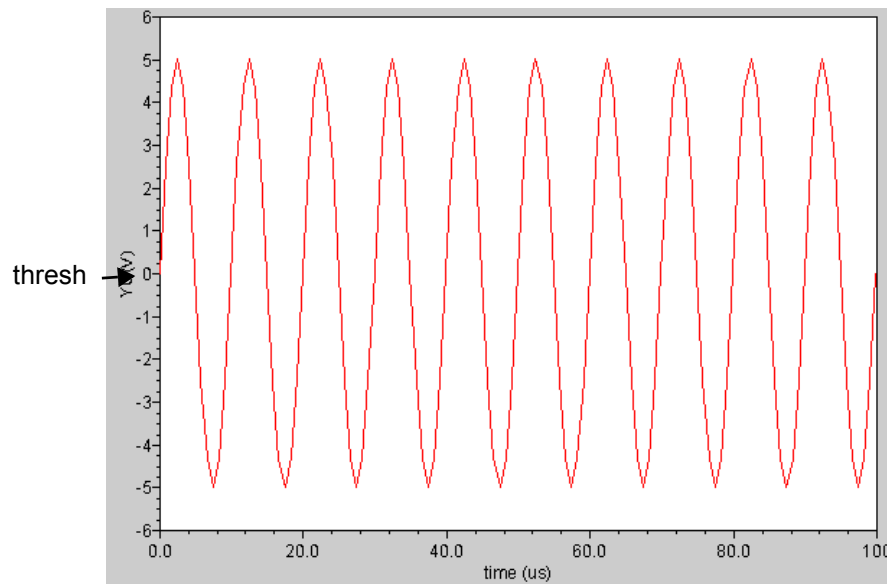
```
crossesOut[4] = 5e-05
```

```
crossesOut[5] = 6e-05
```

```
crossesOut[6] = 7e-05
```

```
crossesOut[7] = 8e-05
```

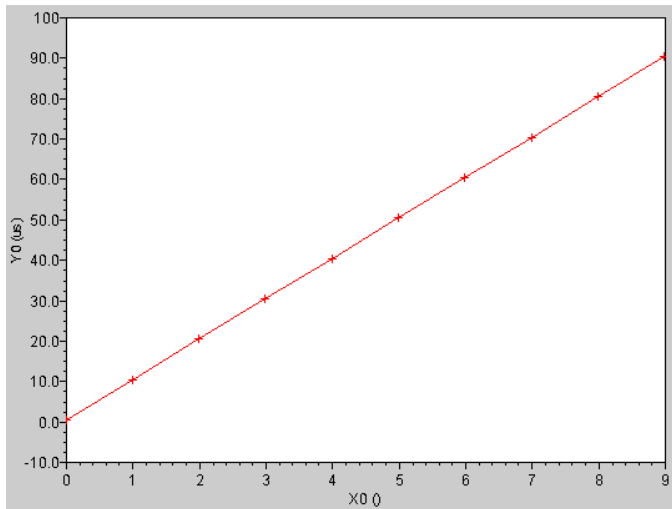
```
crossesOut[8] = 9e-05
```



## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

The output waveform looks as shown below:



### d2r (degrees-to-radians)

Converts a waveform from degrees to radians.

#### Syntax

```
d2r( arg )  
d2r( arg=arg )
```

#### Arguments

<i>arg</i>	The scalar or signal.
------------	-----------------------

#### Example

```
export real myd2r = d2r( 180 )
```

### db

Converts a signal to db where  $db=20*\log(x)$ . This function usually applies to voltage or current signals in volts or amperes.

### Syntax

```
db( arg )  
db( arg=arg )
```

### Arguments

*arg*                                      The scalar or signal.

### Example

```
export real dcgain = db( V(out) / V(in)) @1MHz
```

The above example assumes that *out* and *in* are signals from an ac dataset.



## db10

Converts a signal to db where  $db=10*\log(x)$ . This function usually applies to power signals in watts.

### Syntax

```
db10( arg )  
db10( arg=arg )
```

### Arguments

<i>arg</i>	The scalar or signal.
------------	-----------------------

### Example

```
export real mydb10= db10( v0:pwr )
```

## dbm

Converts a signal to dbm where  $\text{dbm} = 10 \cdot \log(x) + 30$ . This function usually applies to power signals in milliwatts (mW).

### Syntax

```
dbm ( arg )  
dbm ( arg=arg )
```

### Arguments

<i>arg</i>	The scalar or signal.
------------	-----------------------

### Example

```
export real mydbm= dbm ( v0:pwr )
```

## deltax

Returns the difference in the abscissas of two cross events.

### Syntax

```
deltax( sig1[, sig2 [, dir1[, n1[, thresh1[, start1[, dir2[, n2[, thresh2[,  
    start2], xtol[, ytol[, accuracy]]]]]]]]]] )  
  
deltax( sig1=sig1, sig2=sig2 [, dir1=dir1] [, n1=n1] [, thresh1=thresh1] [,  
    start1=start1] [, dir2=dir2] [, n2=n2] [, thresh2=thresh2] [,  
    start2=start2] [, xtol=xtol][, ytol=ytol][, accuracy=accuracy])
```

### Arguments

<i>sig1</i>	The signal whose cross event begins the measurement interval.
<i>sig2</i>	The signal whose cross event ends the measurement interval.
<i>dir1</i>	The direction of the cross at the beginning of the measurement interval. 'rise' directs the function to look for crossings where the Y value is increasing, 'fall' for crossings where the Y value is decreasing, and 'cross' for crossings in either direction. Valid values : 'cross' 'rise', 'fall' Default: 'cross'
<i>n1</i>	The occurrence of the crossing for the beginning of the measurement interval. The first crossing is n=1, the second crossing is n=2, and so on. Default: 1
<i>thresh1</i>	The Y value whose crossing begins the measurement interval. Default: 0
<i>start1</i>	The time at which the function is enabled. Default: 0
<i>dir2</i>	The direction of the cross at the end of the measurement interval. 'rise' directs the function to look for crossings where the Y value is increasing, 'fall' for crossings where the Y value is decreasing, and 'cross' for crossings in either direction. Valid values: 'cross', 'rise', 'fall' Default: 'cross'

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<i>n2</i>	The occurrence of the crossing for the end of the measurement interval. The first crossing is n=1, the second crossing is n=2, and so on. Default: 1
<i>thresh2</i>	The Y value whose crossing ends the measurement interval. Default: 0
<i>start2</i>	The offset from <i>start1</i> where the function begins looking for the cross that ends the delay measurement. Default: 0
<i>absstart2</i>	The absolute offset from the beginning of the signal where the function begins looking for the cross that ends the delay measurement. Default: 0
<i>xtol</i>	The relative tolerance in percentage value in the X direction. Default: 1
<i>ytol</i>	The relative tolerance in percentage value in the Y direction. Default: 1
<i>accuracy</i>	Specifies whether the function should use interpolation, or use iteration controlled by the absolute tolerances to calculate the value. 'interp directs the function to use interpolation, and 'exact directs the function to consider the xtol and yval values. Data types: name for scalar Valid values: 'interp, 'exact Default: 'exact

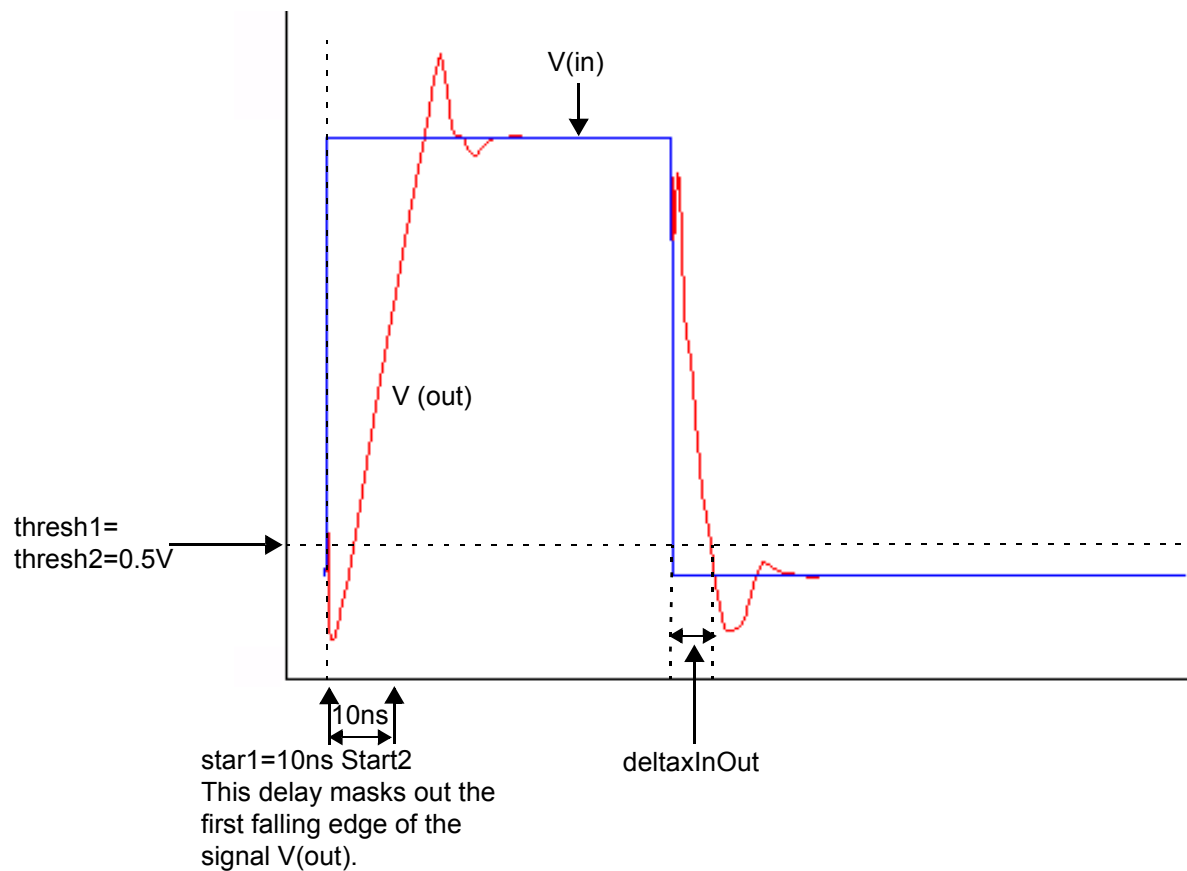
# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

## Example 1

```
export real deltaxInOut = deltax( sig1=V(in), sig2=V(out), dir1='fall, \
thresh1 = 0.5, dir2='fall, thresh2=0.5, start1=10n, start2=10n )
```

The following diagram illustrates how the result from the above example is determined.



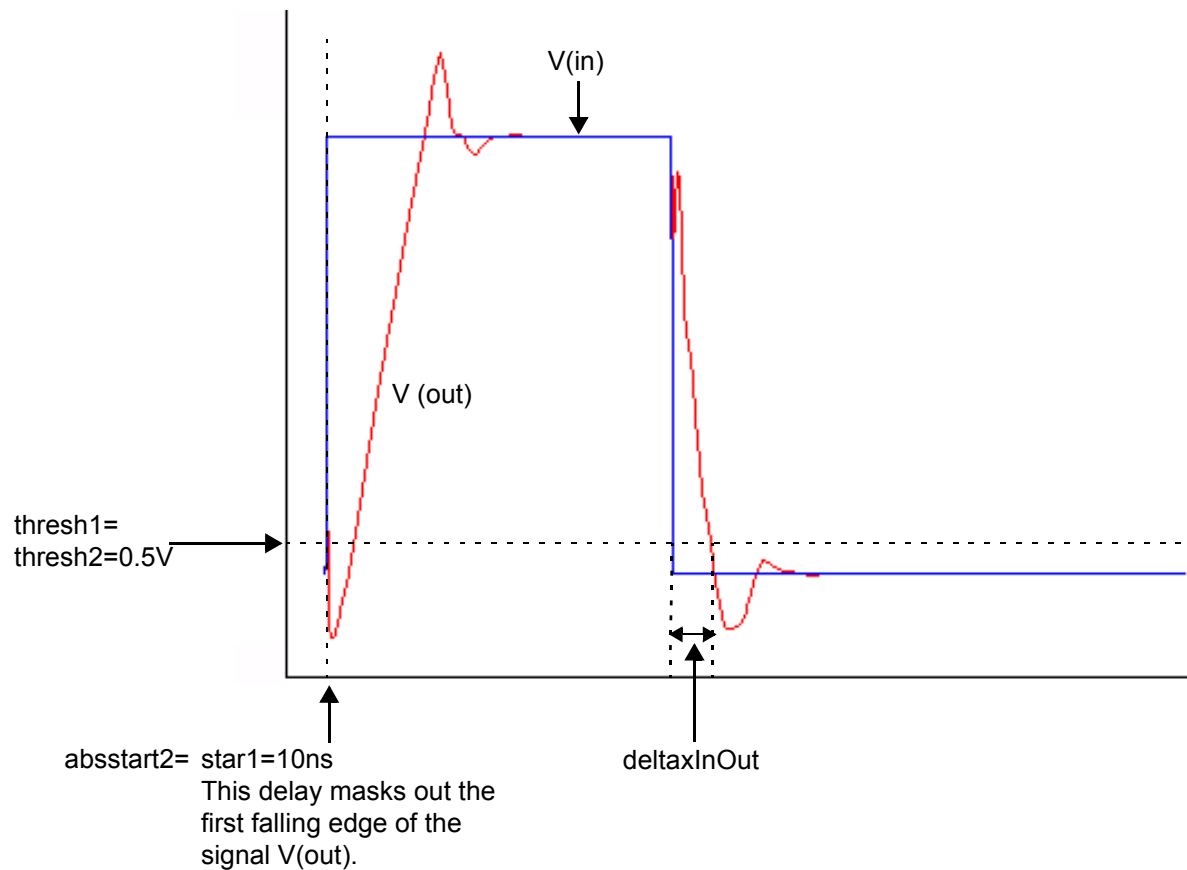
## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

### Example 2

```
export real delay2 = deltax( sig1=V(in), sig2=V(out), dir1='fall', \  
thresh1 = 0.5, dir2='fall', thresh2=0.5, start1=10n, absstart2=10n )
```

The following diagram illustrates how the result from the above example is determined.



### **deltaxes**

The `deltaxes` function is similar to the `deltax` function. However, it returns the differences in the abscissas of two cross events in the form of an array.

**Note:** For syntax and arguments, please see [deltax](#).

## deriv

Returns the derivative of a signal.

### Syntax

```
deriv( sig )  
deriv( sig=sig )
```

### Arguments

*sig*                                      The signal.

### Example1

```
export real out_4n= deriv( V(out) )@4n
```

The derivative is calculated for signal `V(out)` at `t=4ns`.

### Example 2

```
export real out_dvdt_fall=deriv(out)@cross(out, dir='fall, n=1, thresh=1.5)
```

The derivative is calculated for signal `V(out)` at its first crossing point at 1.5V in the `fall` direction.



## dutycycle

Calculates the ratio of the time for which the signal remains high to the period of the signal. You should use this function on periodic signals only.

### Syntax

**dutycycle**( *sig*, *theta*, *mode*)

**dutycycle**( **sig**=*sig*, **theta**=*theta*, [ **mode**=*'integrate | 'percentage | 'threshold 1* ] )

### Arguments

<i>sig</i>	The signal.
<i>theta</i>	Percentage that defines the logic high of the signal. A threshold value is calculated as follows: $yThresh = ((Ymax - Ymin) * theta * 0.01) + Ymin$ The portion of the signal above yThresh is taken as high. Default value: 50.0
<i>mode</i> = <i>'integrate</i>	If <i>mode</i> is set to <i>integrate</i> , <i>theta</i> is ignored and the threshold value is calculated as in the SKILL mode.
<i>mode</i> = <i>'percentage</i>	If <i>mode</i> is set to <i>percentage</i> , <i>theta</i> will be a percentage value. This is the default value.
<i>mode</i> = <i>'threshold</i>	If <i>mode</i> is set to <i>threshold</i> , the value of <i>theta</i> is taken as the threshold value.

**Note:** If *mode* is not specified, it will automatically be set to *percentage*.

### Example

```
export real dutycycleOut = dutycycle ( sig=V(out), theta=40 )
```

returns

```
dutycycleOut = 0.25436626860397216
```

## dutycycles

Returns the dutycycle of a nearly-periodic signal as a function of time.

### Syntax

**dutycycles** ( *sig*, *theta* )

**dutycycles** ( **sig**=*sig*, **theta**=*theta* )

### Arguments

*sig*                                      The signal.

*theta*                                      Percentage that defines the logic high of the signal. A threshold value is calculated as follows:  
 $yThresh = ((Ymax - Ymin) * theta * 0.01) + Ymin$   
The portion of the signal above *yThresh* is taken as high.  
Default value: 50.0

### Example 1

In Virtuoso Visualization and Analysis XL,

```
export real dutycyclesOut = dutycycles ( sig=V(out), theta=40 )
```

### Example 2

```
export real dutycycles_q[] = dutycycles ( sig=V(q), theta=40 )
```

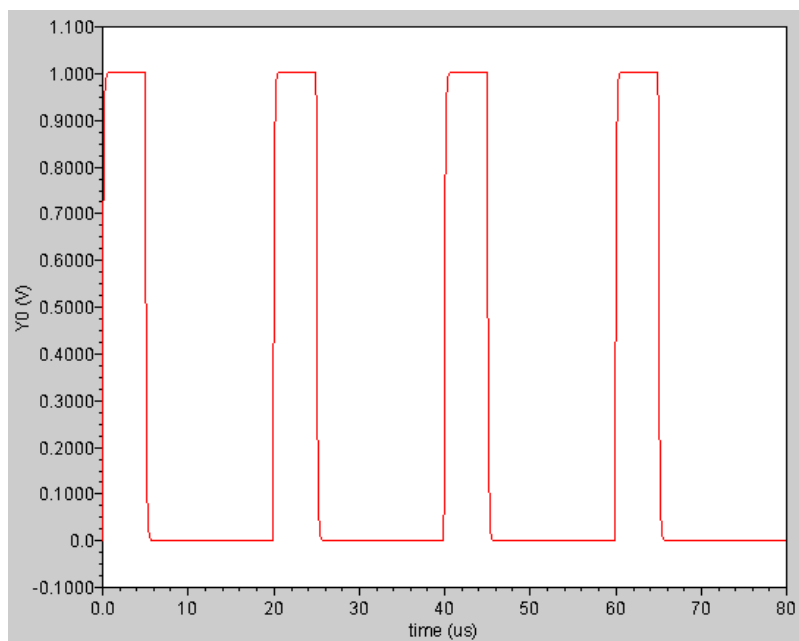
### returns

```
dutycycles_q[0] = 0.3877  
dutycycles_q[1] = 0.6897  
dutycycles_q[2] = 0.685696
```

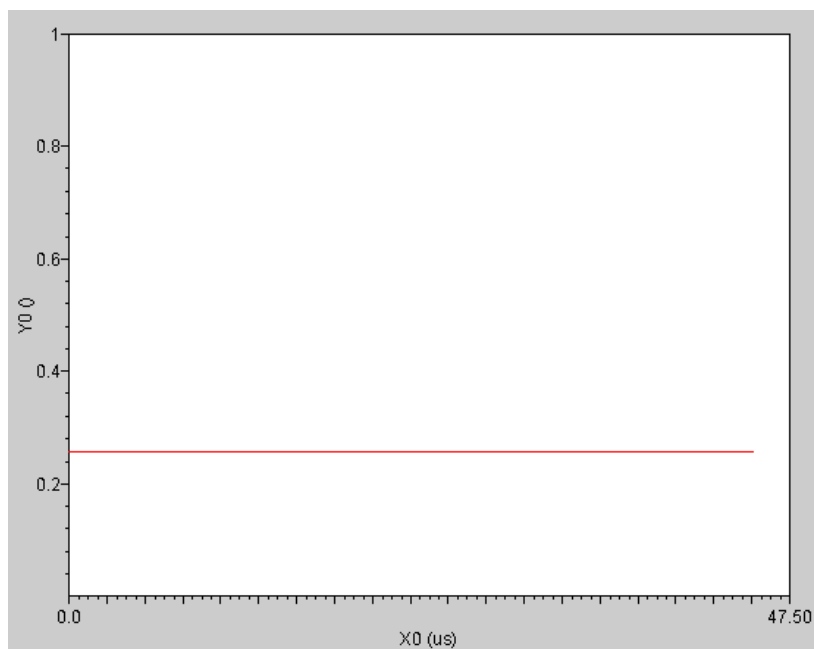
## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

transforms the following input signal



into the following output signal



### **exp**

Returns the  $e^x$  value of a signal.

### **Syntax**

```
exp( arg )  
exp( arg=arg )
```

### **Arguments**

<i>arg</i>	The scalar or signal.
------------	-----------------------

### **Example**

```
export real myexp = exp( 2 )
```

returns

```
myexp = 7.389
```

## falltime

Returns the fall time for a signal measured between percent high and percent low of the difference between the initial and final values. The measurement is always done with ordinate (Y-axis) values.

**Note:** You can use the `falltimes` function to obtain the fall time for all edges instead of a single edge that is returned by the `falltime` function.

## Syntax

```
falltime( sig[, initval[, finalval[, inittype[, finaltype[, theta1[, theta2[,  
          xtol[, ytol[, accuracy]]]]]]]] )  
  
falltime( sig=sig, initval=initval, finalval=finalval [, inittype=inittype] [,  
          finaltype=finaltype] [, theta1=theta1] [, theta2=theta2] [, xtol=xtol] [,  
          ytol=ytol] [, accuracy=accuracy] )
```

## Arguments

<i>sig</i>	The signal.
<i>initval</i>	The Y-axis value (if <i>inittype</i> is 'Y') or X-axis value at the specified X-axis point (if <i>inittype</i> is 'X') that starts the falltime interval.
<i>finalval</i>	The Y-axis value (if <i>inittype</i> is 'Y') or X-axis value at the specified X-axis point (if <i>inittype</i> is 'X') that ends the falltime interval.
<i>inittype</i>	When 'X', the initial value is an X value. When 'Y', the initial value is a Y value. Valid values: 'X', 'Y' Default: 'Y'
<i>finaltype</i>	When 'X', the final value is an X value. When 'Y', the final value is a Y value. Valid values: 'X', 'Y' Default: 'Y'
<i>theta1</i>	The threshold high expressed as a percentage of the difference between the initial and final values. Default: 90

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<i>theta2</i>	The threshold low expressed as a percentage of the difference between the initial and final values. Default: 10
<i>xtol</i>	The relative tolerance in percentage value in the X direction. Default: 1
<i>ytol</i>	The relative tolerance in percentage value in the Y direction. Default: 1
<i>accuracy</i>	Specifies whether the function should use interpolation, or use iteration controlled by the absolute tolerances to calculate the value. 'interp directs the function to use interpolation, and 'exact directs the function to consider the xtol and yval values. Data types: name for scalar Valid values: 'interp, 'exact Default: 'exact

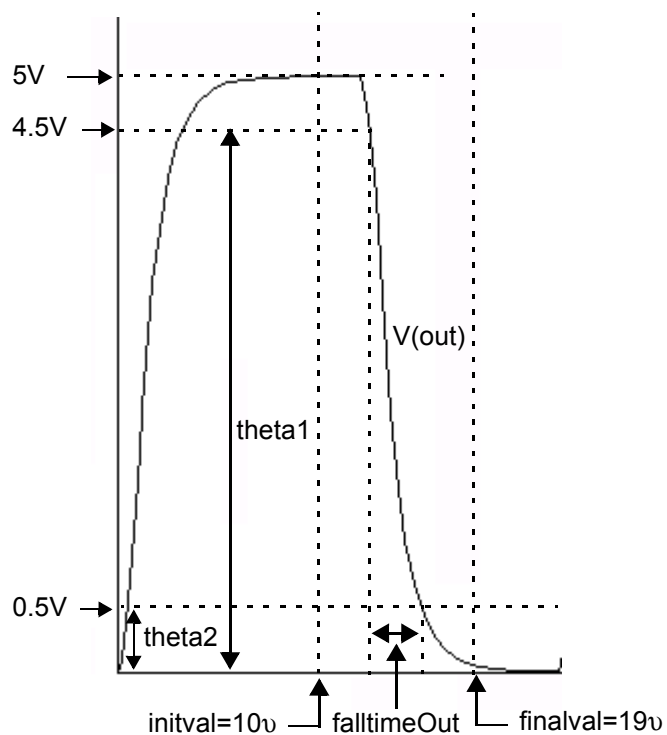
### Example

```
export real falltimeOut = falltime ( arg=V(out), initval=10u, initttype='x,  
finalval=19u, finaltype='x, theta1=90, theta2=10)
```

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

The following diagram illustrates how the result from the above example is determined.



## fft

Performs a Fast Fourier Transform on the signal and returns its frequency spectrum.

### Syntax

```
fft( sig, from, to, numPoints, window )
```

```
fft( sig=sig, from=from, to=to, numPoints=numPoints, window=window )
```

### Arguments

<i>sig</i>	The signal.
<i>from</i>	The starting X value.
<i>to</i>	The ending X value.
<i>numPoints</i>	The number of data points to be used for calculating the fft. If this number is not a power of 2, it is automatically raised to the next higher power of 2.
<i>window</i>	<p>The algorithm used for calculating the fft. For more information, see <a href="#">window</a>.</p> <p>Valid values: 'rectangular', 'bartlett', 'bartlettthann', 'blackman', 'blackmanharris', 'cosine2', 'cosine4', 'extcosbell', 'flattop', 'halfcyclesine', 'half3cyclesine', 'halfcyclesine3', 'half6cyclesine', 'halfcyclesine6', 'hamming', 'hanning', 'nuttall', 'parzen', 'triangular'</p> <p>Default: 'rectangular'</p>

### Example

In Virtuoso Visualization and Analysis XL,

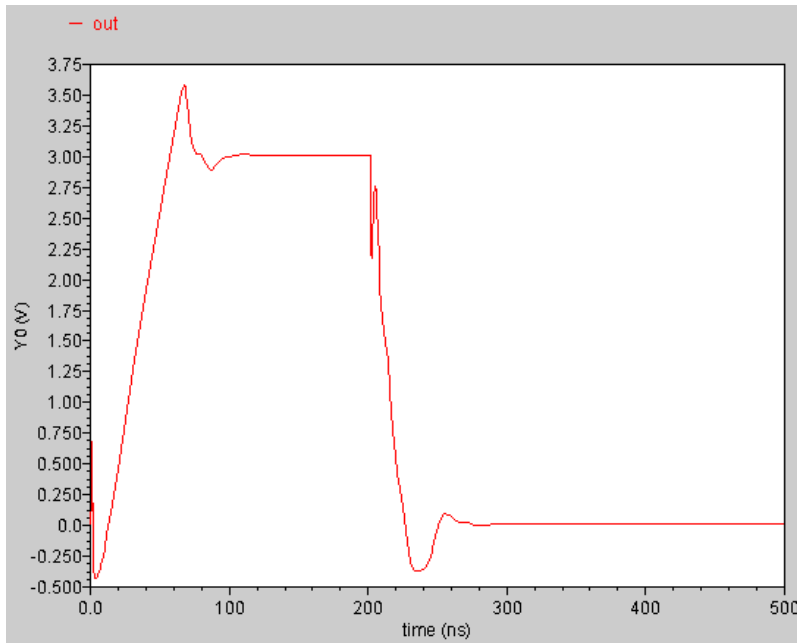
```
fft( sig=(V(out), from=1ns, to=200ns, numPoints=512, window='bartlett)
```



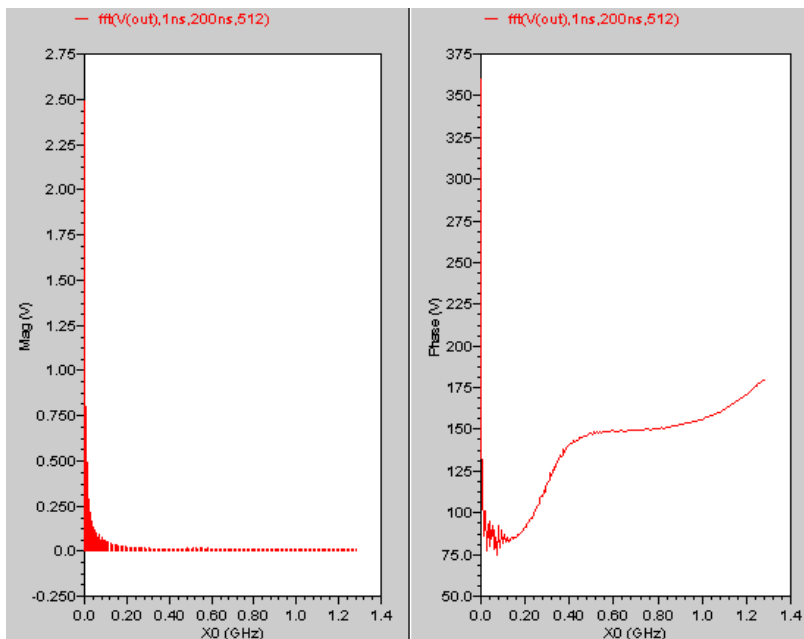
# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

transforms the following input signal



into the following output signal. The left subwindow shows the magnitude part of the spectrum and the right subwindow shows the phase part.



### flip

Returns a reversed version of a signal (rotates the signal along the Y-axis).

### Syntax

**flip** ( *sig* )

**flip** ( **sig**=*sig* )

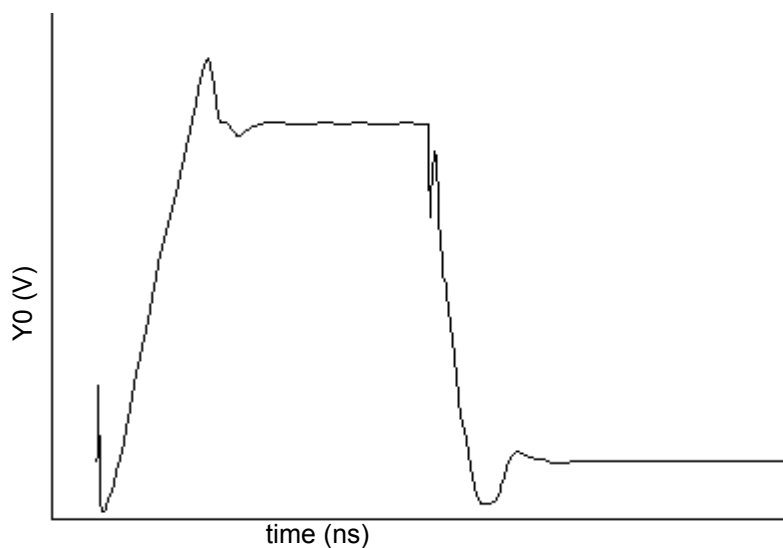
### Arguments

*sig*                                      The signal.

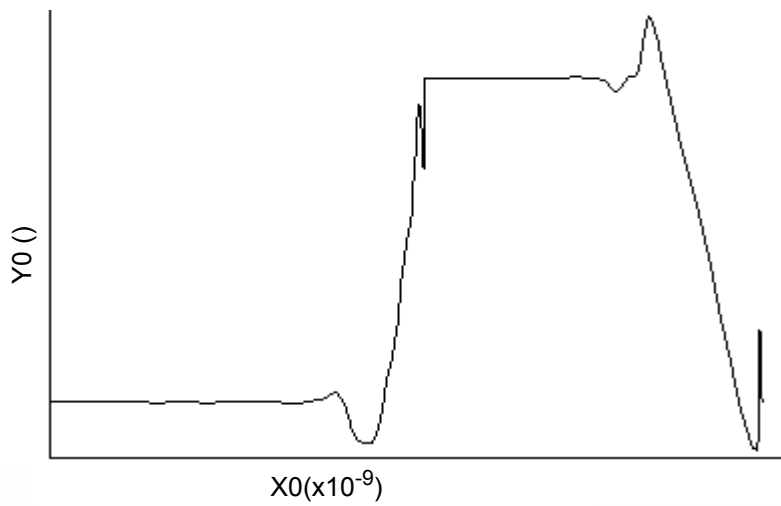
### Example

```
export real flipOut = flip( V(out) )
```

transforms the following input signal



into the following output signal.



### **floor**

Rounds a real number down to the closest integer value.

### **Syntax**

```
floor( arg )  
floor( arg=arg )
```

### **Arguments**

<i>arg</i>	The real number.
------------	------------------

### **Example**

```
export real myfloor = floor( 1.6 )
```

returns

```
myfloor = 1
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

## fmt

Provides formatting capability to turn MDL datatypes into a string representation.

### Syntax

```
fmt( "format", varargs )  
fmt( format="format", varargs=varargs )
```

### Arguments

*format*                      The percent code format string. In addition to the standard percent codes in C ( %s, %S, %g, %G, %e, %E, %f, %d, %i, %u, %o, %x, %X), the following percent codes are also available:  
%V - the value of the *varargs* argument.  
The following “\” qualifiers (constant escape sequences) are also supported:  
\n - newline  
\t - tab  
In addition to these, full precision qualifier support exists for all supported percent codes.

*varargs*                      Arguments to the *fmt* function used to fill in the percent code values in the *format* string.

### Example

```
alias measurement printmeas {  
    input string out="myfile.out"  
    print fmt("Header is %s\n", out) to=out  
    print fmt("%s\t%s\t\t%s\t%s\t%s\t%s\n",  
        "%d", "%f", "%o", "%x", "%X", "%u") addto=out  
    print fmt("%d\t%f\t%o\t%x\t%X\t%u\n", 10, 10, 10, 10, 10, 10) addto=out  
}  
run printmeas (out="test.dat")
```

The simulator writes the following results to the *test.dat* file:

```
Header is test.dat  
%d      %f      %o      %x      %X      %u  
10      10.000000  12      a      A      10
```

## freq

Returns an array of frequencies defined by the given threshold crossing and direction for a signal.

### Syntax

```
freq( sig, thresh, dir )
```

```
freq( sig=sig, thresh=thresh, dir=direction )
```

### Arguments

<i>sig</i>	The signal.
<i>thresh</i>	The threshold Y-axis value to be crossed.
<i>dir</i>	The direction of the crossing event. Valid values: 'rise', 'fall'

### Example

In Virtuoso Visualization and Analysis XL,

```
export real freqOut = freq ( sig=V(out), thresh=0.5, dir='rise )
```

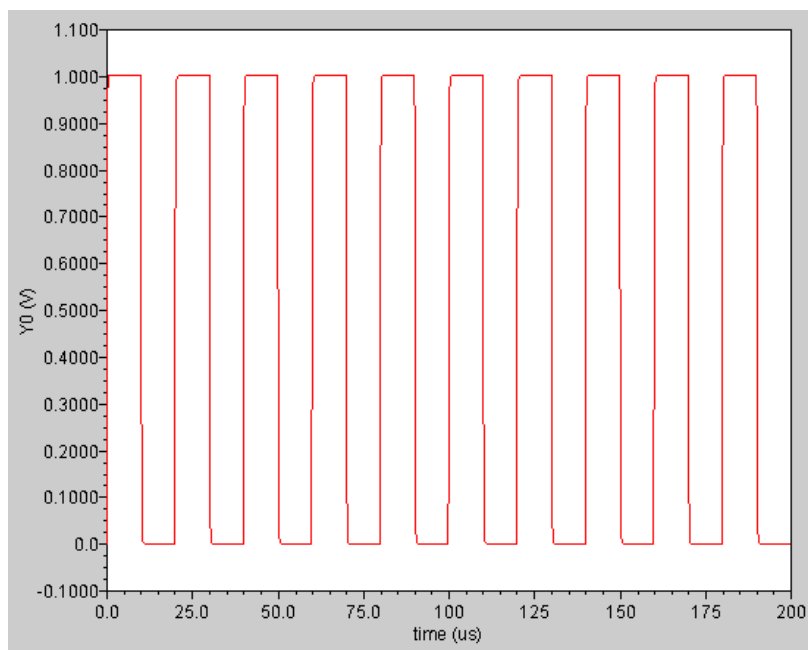
#### returns

```
freqOut[0] = 5.0001e+04  
freqOut[1] = 5e+04
```

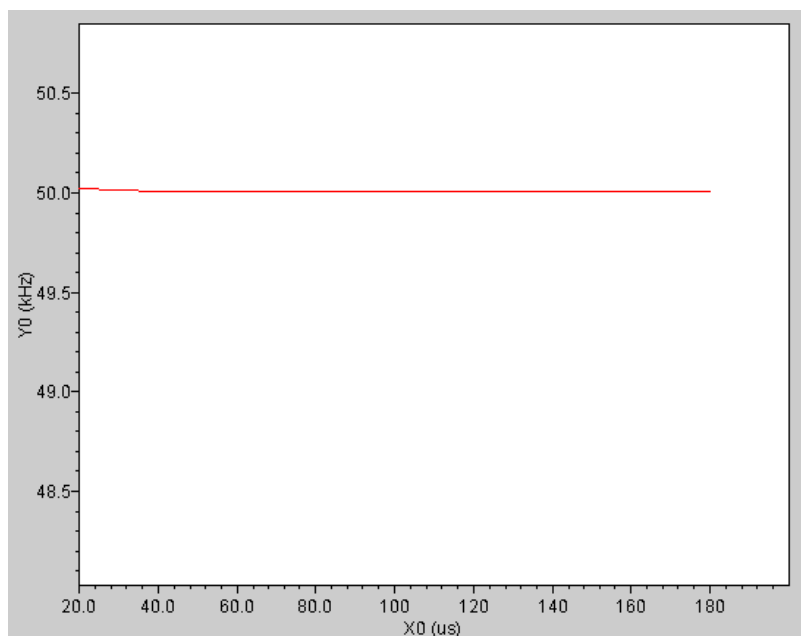
## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

the following input signal



is converted to the following output signal



## freq\_jitter

Returns a waveform representing the deviation from the average frequency.

### Syntax

**freq\_jitter** ( *sig*, *thresh*, *dir*, *binsize* )

**freq\_jitter** ( **sig**=*sig*, **thresh**=*thresh*, **dir**=*direction*, **binsize**=*binsize* )

### Arguments

<i>sig</i>	The signal.
<i>thresh</i>	The threshold Y-axis value to be crossed.
<i>dir</i>	The direction of the crossing event. Valid values: 'rise', 'fall'
<i>binsize</i>	Integer used to calculate the average frequency of the signal. If binsize=0, all frequencies are used to calculate the average. If binsize= <i>N</i> , the last <i>N</i> frequencies are used to calculate the average. Default value=0

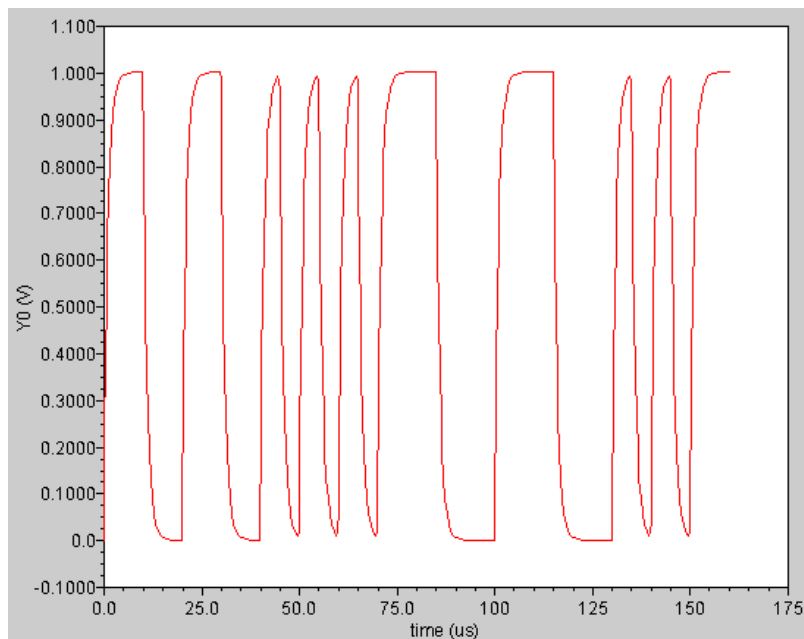
### Example

In Virtuoso Visualization and Analysis XL,

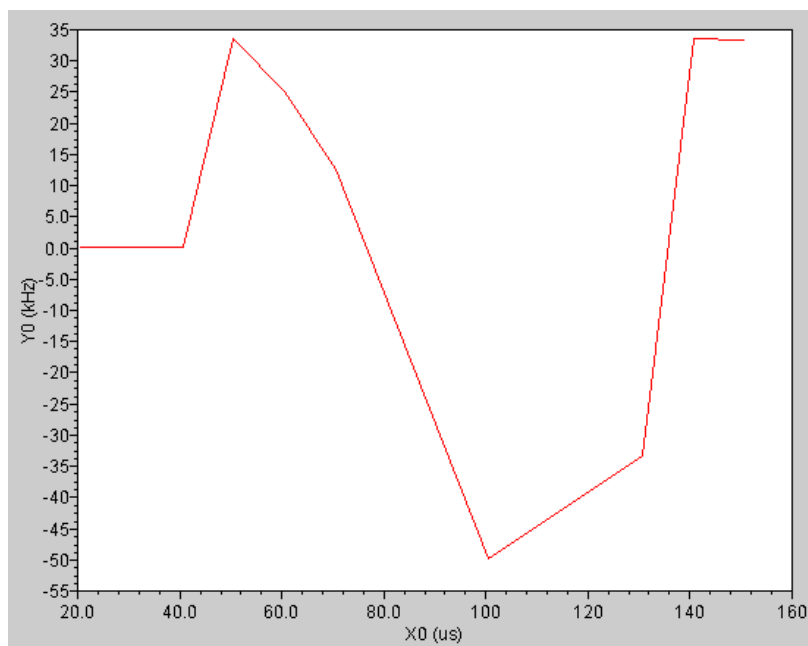
```
export real freq_jitterOut = freq_jitter ( sig=V(out), thresh=0.5, dir='rise',  
binsize=4 )
```



the following input signal



is converted to the following output signal



### gainBwProd

Returns the product of DC gain and upper cutoff frequency for a low-pass type filter or amplifier.

#### Syntax

**gainBwProd**( *sig* )

**gainBwProd**( **sig**=*sig* )

#### Arguments

<i>sig</i>	The signal. It can represent the magnitude of the gain or a frequency response.
------------	---

#### Example

```
export real gainBwProdOut = gainBwProd ( sig=mag(out) )
```

#### returns

```
gainBwProdOut = 1804641.158689868
```

### gainmargin

Computes the gain margin of the loop gain of an amplifier.

The gain margin is calculated as the magnitude (in dB) of the gain at f0. The frequency f0 is the smallest frequency in which the phase of the gain provided is -180 degrees. For stability, the gain margin must be positive.

### Syntax

```
gainmargin( sig )  
gainmargin( sig=sig )
```

### Arguments

<i>sig</i>	The loop gain of interest over a sufficiently large frequency range.
------------	--

### Example

```
export real gainmar=gainmargin(vout)
```

## getinfo

Returns information related to the simulator, such as version, subversion, and command information.

### Syntax

```
getinfo( type )
```

```
getinfo( type=type )
```

### Arguments

<i>type</i>	Type of information to be displayed. Valid values are 'simulator', 'version', 'subversion', and 'cmd'.
-------------	--

### Example

```
string simulator=getinfo('simulator')
string version=getinfo('version')
string subversion=getinfo('subversion')
string cmdline=getinfo('cmd')
```

## groupdelay

Calculates the rate of change of phase with respect to frequency in a frequency response measurement.

`groupdelay=d(phase)/dw`

where  $w$ =angular frequency in  $\text{rad/s}=2*\text{PI}*f$

## Syntax

`groupdelay ( sig )`

`groupdelay ( sig=sig )`

## Arguments

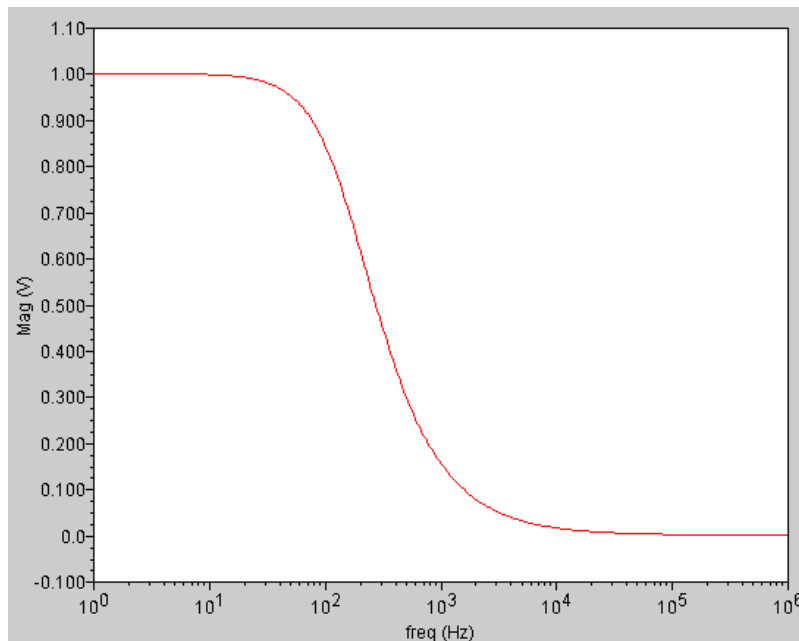
*sig*                                      The signal. It should represent frequency response.

## Example

In Virtuoso Visualization and Analysis XL,

```
export real groupdelayOut = groupdelay ( sig=out )
```

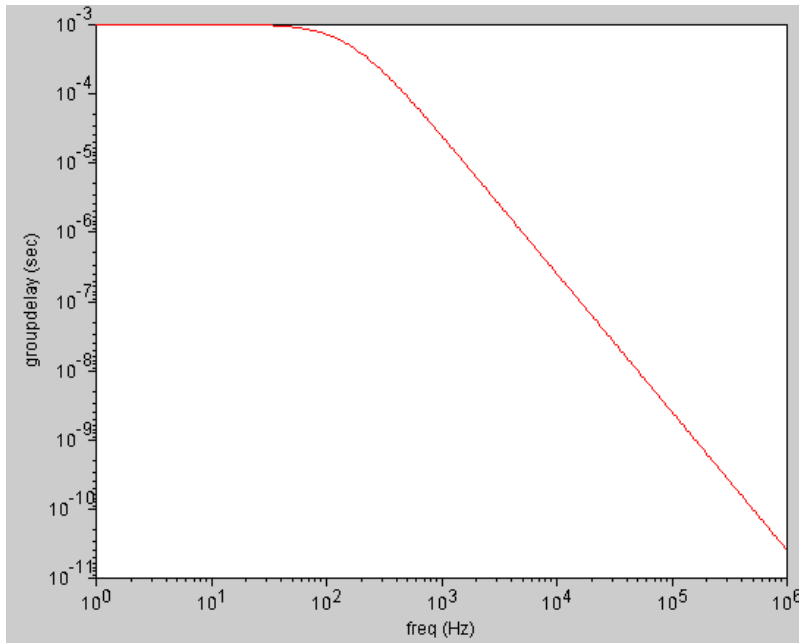
the following input signal



## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

is converted to the following output signal



## histo

Creates a histogram from a signal.

The histo function is available from the calculator. It is not supported within a Spectre MDL control file since it returns a scalar and not a waveform.

## Syntax

```
histo( sig, nbins, min, max )  
histo( sig=sig, nbins=nbins, min=min, max=max )
```

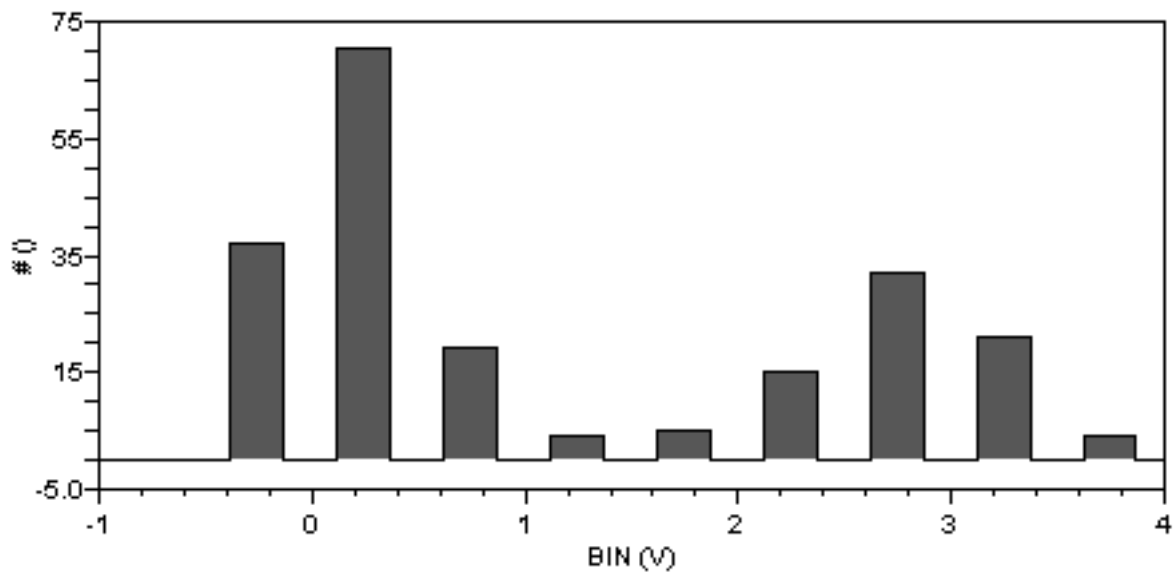
## Arguments

<i>sig</i>	The waveform.
<i>nbins</i>	The number of bins to be created.
<i>min</i>	The value that specifies the smaller end point of the range of values included in the histogram.
<i>max</i>	The value that specifies the larger end point of the range of values of values included in the histogram.

## Example

```
histo(V(out),nbins=10, min=-1.0, max=4.0)
```

creates a display with 10 bins that might look like this when the leftmost bin is empty.





## I

Current probe function.

### Syntax

```
I(devname)                //equivalent to devname:0
I(devname:term)            //term can be either terminal name or terminal index
I(Instname:term)           //term can be either terminal name or terminal index
```

The `I` probe function does not support current access by node name, nor does it support current difference between two `devname:term(s)`. In other words, it is illegal to apply the `I` probe to a node or a pair of nodes.

### Arguments

<i>devname</i>	The N-terminal device instance name.
<i>Instname</i>	The N-terminal subcircuit instance name.
<i>term</i>	The terminal name or terminal index of a device or a subcircuit .

### Examples

```
I(Rload:1)    // Returns the current through terminal Rload:1
I(I0.mp1:d)   // Returns the current through device terminal name d
I(I0:vdd1)    // Returns the current through subcircuit terminal name vdd1
```

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

## ifft

Performs an inverse Fast Fourier Transform on a frequency spectrum and returns the time domain representation of the spectrum.

## Syntax

```
ifft ( sig )
```

```
ifft( sig=sig )
```

## Arguments

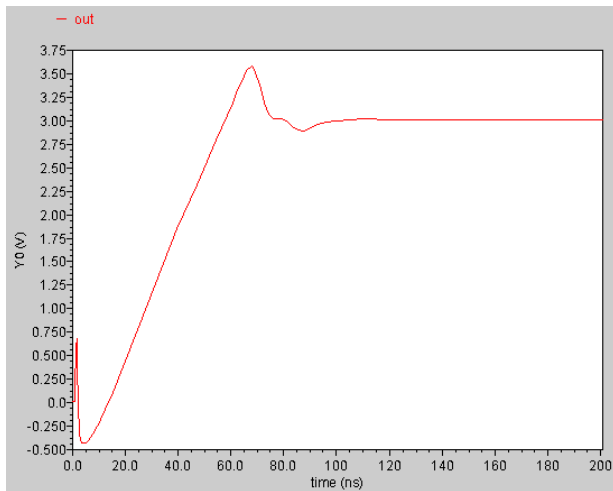
*sig*                                      The frequency spectrum.

## Example

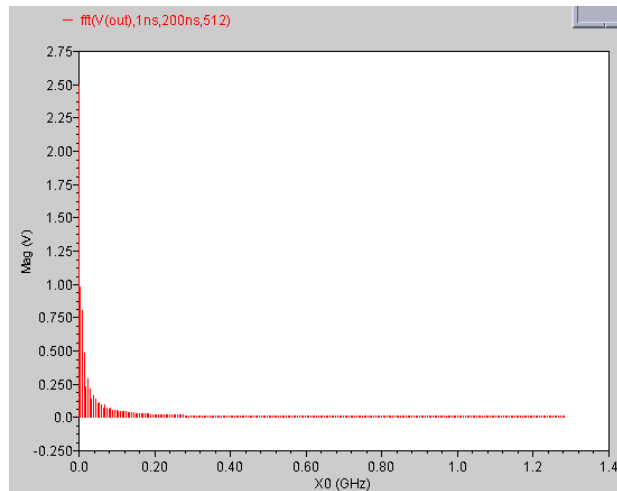
```
fft( sig=V(out), from=1ns, to=200ns, npoints=512)
```

results in the graph on the right side.

The signal out



Fast fourier transform of the signal out



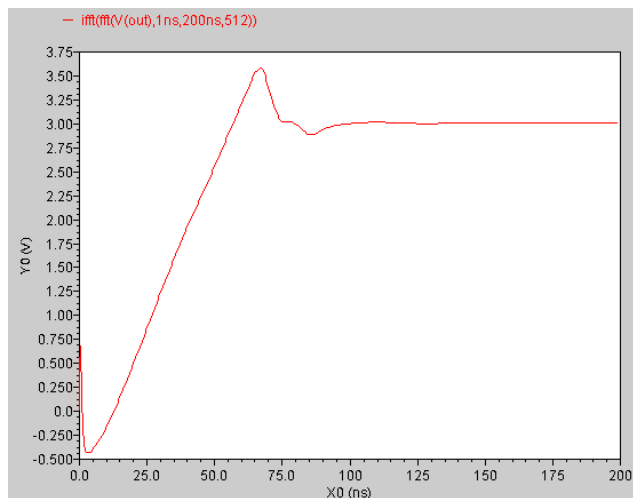
Now if I perform an `ifft` on the above expression,

```
ifft( fft( sig=V(out), from=1ns, to=200ns, npoints=512) )
```

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

The result is the same as the original signal (`out`) – from 1ns to 200ns.



### iinteg

Returns the incremental area under the waveform.

#### Syntax

```
iinteg( sig )  
iinteg( sig=sig )
```

#### Arguments

*sig*                                      The signal.

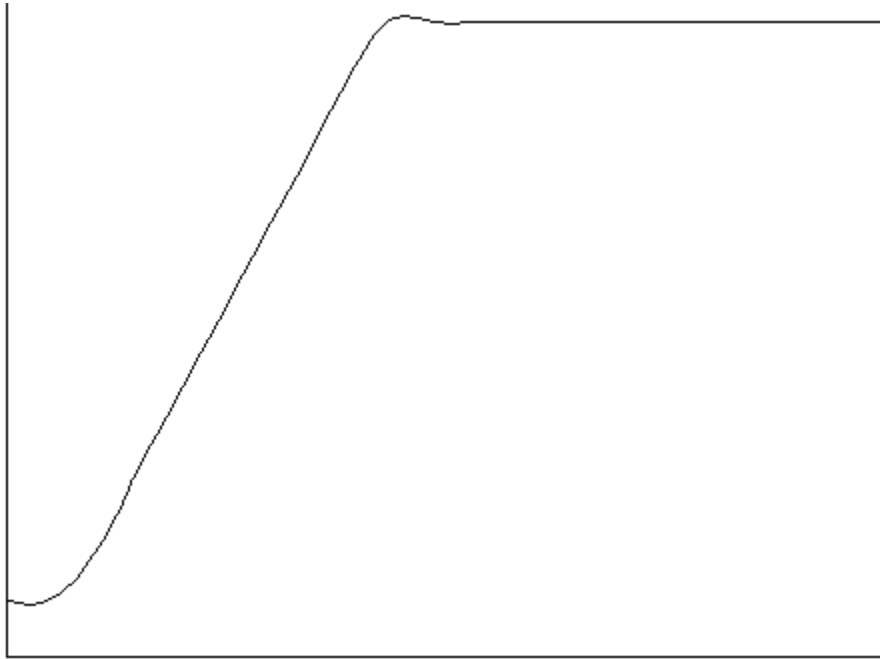
#### Example 1

```
export real iintegOut = iinteg( V(out) )
```

transforms the following input signal



into the following output signal



Each X value on the output trace is equal to the area under the input trace from start till that particular X-value.

## **im**

Returns the imaginary part of a complex number.

### **Syntax**

```
im( arg )
```

```
im( arg=arg )
```

### **Arguments**

*arg*                                      The complex number.

### **Examples**

```
export real myim = im( cplx(1,2) )
```

**returns**

```
myim = 2
```

```
export real im_s11 = im( s(1,1) )
```

**returns**

```
im_s11 = 0.670029
```

### **int**

Returns the integer portion of a real value.

#### **Syntax**

```
int( arg )
```

```
int( arg=arg )
```

#### **Arguments**

<i>arg</i>	The real number whose integer portion is to be returned.
------------	--

#### **Example**

```
int(4.998)
```

returns the value

4

## integ

Returns the area bounded under the curve.

### Syntax

```
integ( sig )  
integ( sig=sig )
```

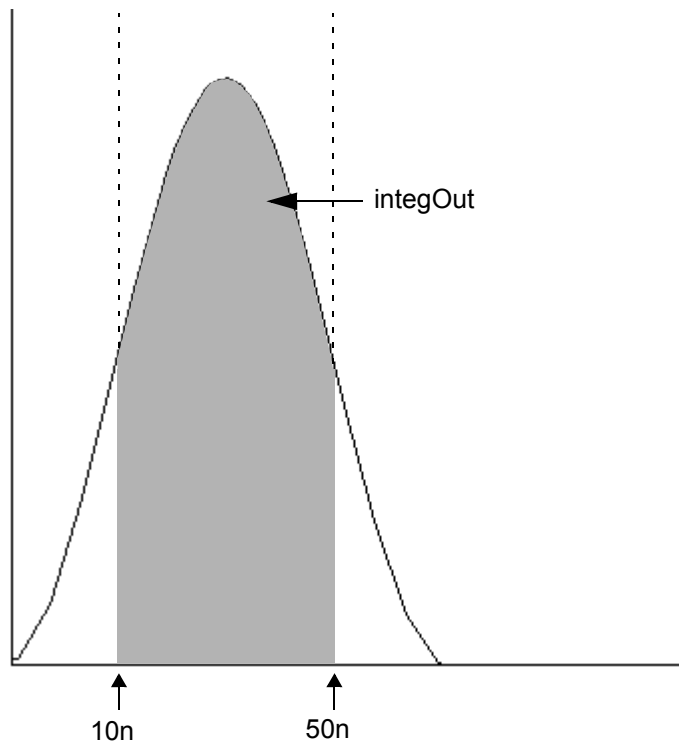
### Arguments

*sig*                      The signal.

### Example 1

```
export real integOut = integ( trim( sig=V(sinewave), from=10n, to=50n ) )
```

The following diagram illustrates how the result from the above example is determined. The result is equal to the shaded area in the graph.





## ln

Returns the natural logarithm of a signal or a number. If no specific point of a signal is specified, MDL returns value for the last simulation point of the signal.

### Syntax

```
ln( arg )  
ln( arg=arg )
```

### Arguments

*arg*                                      The scalar or signal.

### Examples

```
export real mylog = ln ( 10 )
```

returns

mylog = 2.3

```
export real myln = ln ( v(q) )  
export real myln_ons = ln ( v(q) @0 )
```

returns

```
myln = -0.223144  
myln_ons = -21.9773
```

## log10

Returns the base 10 logarithm of a signal or a number. If no specific point of a signal is specified, MDL returns value for the last simulation point of the signal.

### Syntax

```
log10( arg )  
log10( arg=arg )
```

### Arguments

*arg*                                      The scalar or signal.

### Example

```
export real mylog10 = log10( 10 )
```

returns

```
mylog10 = 1
```

### **mag**

Returns the magnitude of a signal or complex number.

### **Syntax**

```
mag( arg )
```

```
mag( arg=arg )
```

### **Arguments**

<i>arg</i>	The scalar or signal.
------------	-----------------------

### **Example**

```
export real mymag = mag( cplx(1,2) )
```

returns

```
mymag = 2.236
```

## max

Returns the maximum value of a signal, maximum value of two real values, or the maximum value of a signal and a real value

### Syntax

```
max ( arg )  
max ( arg=arg )
```

### Arguments

*arg*                                      The scalar or signal.

### Example 1

```
export real maxOut1 = max ( V(out) )
```

### Example 2

```
export real maxOut2 = max ( V(out)@100n, V(out)@200n )
```

This returns the value of `out` at 100n or 200n – whichever is greater.

### Example 3

```
export real maxq=max(trim(q, from=0, to=100n))
```

This returns the maximum value of `out` over the range of `t=0ns` to `t=100ns`.

### Example 4

```
export real maxOut4 = max( I(IP1)@ 1.0 , 1e-15 );
```

### **min**

Returns the minimum value of a signal or the minimum value of two real values.

### **Syntax**

```
min( arg )  
min( arg=arg )
```

### **Arguments**

*arg*                                      The scalar or signal.

### **Example**

```
export real minOut1 = min( V(out) )
```

### **Example 2**

```
export real minOut2 = min ( V(out)@100n, V(out)@200n )
```

This returns the value of `out` at 100n or 200n – whichever is smaller.

### mod

Returns the floating point remainder of the dividend divided by the divisor. The divisor cannot be zero.

### Syntax

```
mod( dividend, divisor )  
mod( dividend=dividend, divisor=divisor )
```

### Arguments

<i>dividend</i>	The scalar dividend.
<i>divisor</i>	The scalar divisor.

### Example

```
export real mymod = mod( 546, 324 )
```

### returns

```
mymod = 222
```

## movingavg

Calculates the moving average for the specified signal.

### Syntax

```
movingavg( sig[, n ])
```

```
movingavg( sig=sig[, n=n ] )
```

### Arguments

<i>sig</i>	The signal.
------------	-------------

<i>n</i>	Number of points specifying the bin size. Default: 1
----------	---

## overshoot

Returns the overshoot/undershoot of a signal as a percentage of the difference between initial and final values.

### Syntax

```
overshoot( sig[, initval[, finalval[, inittype[, finaltype]]]] )  
overshoot( sig=Sig, initval=initval, finalval=finalval [, inittype=inittype]  
          [, finaltype=finaltype] )
```

### Arguments

<i>sig</i>	The signal.
<i>initval</i>	The initial value. To calculate the undershoot of a signal, the initval should be higher than finalval.
<i>finalval</i>	The final value.
<i>inittype</i>	When 'x', the initial value is a time value. When 'y', the initial value is a current or voltage value. Valid values: 'x', 'y' Default: 'y'
<i>finaltype</i>	When 'x', the final value is a time value. When 'y', the final value is a current or voltage value. Valid values: 'x', 'y' Default: 'y'

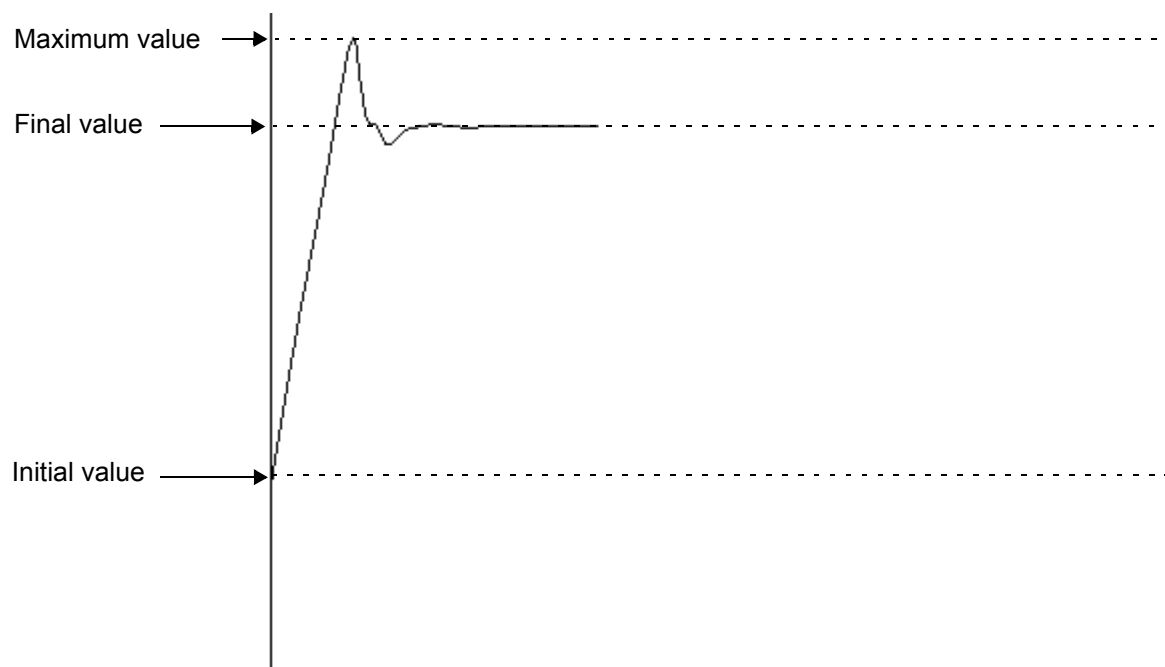


## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

### Example

```
export real overshootOut = overshoot ( sig=V(out), initval=1, finalval=3,  
inittype='y', finaltype='y' )
```



OvershootOut is given by the following formula:

$$\text{OvershootOut} = \frac{\text{MaximumValue} - \text{FinalValue}}{\text{FinalValue} - \text{InitialValue}}$$

## period\_jitter

Returns a waveform representing the deviation from the average period.

### Syntax

**period\_jitter** ( *sig*, *thresh*, *dir*, *binsize* )

**period\_jitter** ( **sig**=*sig*, **thresh**=*thresh*, **dir**=*direction*, **binsize**=*binsize* )

### Arguments

<i>sig</i>	The signal.
<i>thresh</i>	The threshold Y-axis value defining the period/frequency of the signal.
<i>dir</i>	The direction of the crossing event. Valid values: 'rise', 'fall Default value: 'rise
<i>binsize</i>	Integer used to calculate the average frequency of the signal. If binsize=0, all periods are used to calculate the average. If binsize= <i>N</i> , the last <i>N</i> periods are used to calculate the average. Default value=0

### Example

```
export real period_jitterOut = period_jitter ( sig=V(out), thresh=0.5, dir='rise',  
binsize=4 )
```

### ph

Returns the phase of a signal in radians.

#### Syntax

```
ph( arg[, wrap=<value> ])  
ph( arg=arg, wrap=value )
```

#### Arguments

<i>arg</i>	The signal.
<i>value</i>	Wraps the phase. The phase is wrapped around +/- PI. Possible values are <code>yes</code> (default) and <code>no</code> . The value can be scalar.

#### Example

```
ph( v(out), wrap='no' )
```

### phasemargin

Computes the phase margin of the loop gain of an amplifier. The phase margin is calculated as the difference between the phase of the gain in degrees at  $f_0$  and at -180 degrees. The frequency  $f_0$  is the smallest frequency where the gain is 1. For stability, the phase margin must be positive. The value is returned in degrees.

#### Syntax

```
phasemargin( sig )  
phasemargin( sig=sig )
```

#### Arguments

<i>sig</i>	The loop gain of interest over a sufficiently large frequency range.
------------	--

#### Example

```
export real phasemar=phasemargin(vout)
```

### pow

Returns the value of base raised to the power of exponent ( $base^{exponent}$ ).

### Syntax

```
pow( base, exponent )
```

```
pow( base=base, exponent=exponent )
```

### Arguments

*base*                                      The base argument.

*exponent*                                The exponent argument.

### Example

```
export real mypow = pow( 2,2 )
```

### returns

```
mypow = 4
```

## pp (peak-to-peak)

Returns the difference between the highest and lowest values of a signal.

### Syntax

```
pp( sig )  
pp( sig=sig )
```

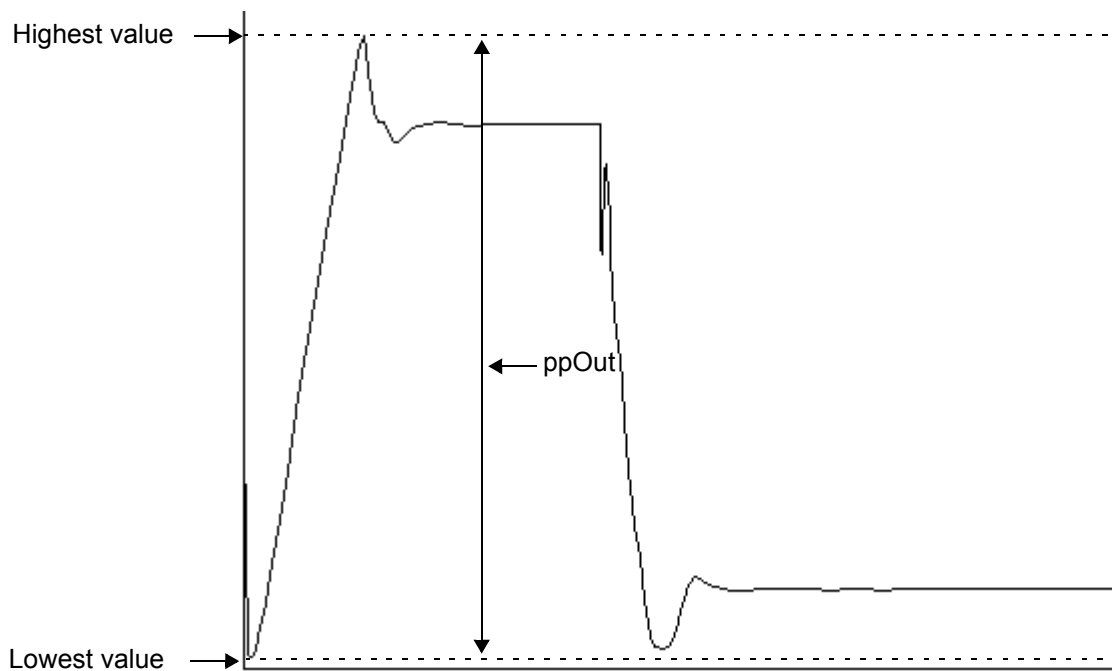
### Arguments

*sig*                      The signal.

### Example 1

```
export real ppOut = pp( V(out) )
```

The following diagram illustrates how the result from the above example is determined.



### pzbode

Calculates and plots the transfer function for a circuit from pole zero simulation data. This function is available only in the MDL mode.

### Syntax

```
pzbode( poles, zeroes, c, minfreq, maxfreq, npoints)
pzbode( poles=poles , zeroes=zeroes , c=c , minfreq=minfreq , maxfreq=maxfreq
      , npoints=npoints )
```

### Arguments

<i>poles</i>	The poles.
<i>zeroes</i>	The zeroes.
<i>c</i>	The transfer gain constant.
<i>minfreq</i>	The minimum frequency for the bode plot.
<i>maxfreq</i>	The maximum frequency for the bode plot.
<i>npoints</i>	The frequency interval for the bode plot, in points per decade.

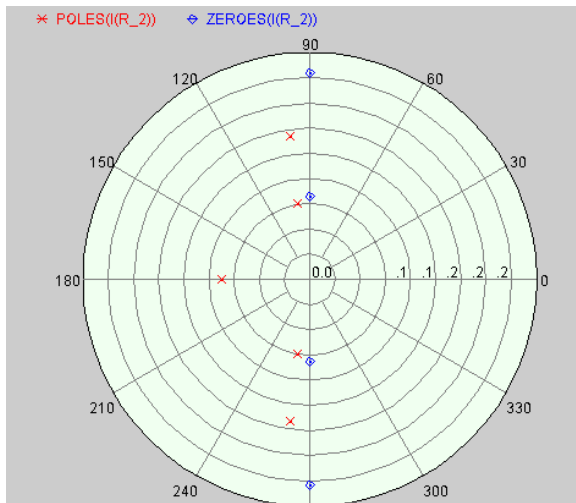
# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

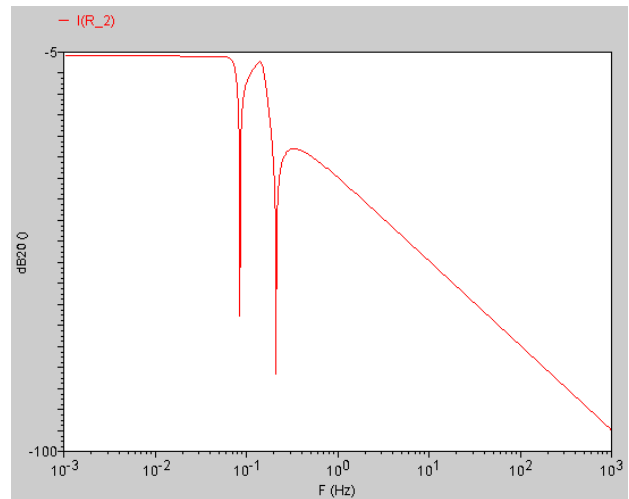
## Example

The following diagram illustrates how the result with the values  $poles=POLES<I<R_1>>$ ,  $zeroes=ZEROES<I<R_1>>$ ,  $c=I<R_1>\backslash[K\backslash]$ ,  $minfreq=1e-3$ ,  $maxfreq=1e3$ , and  $npoints=1000$  is determined.

Polar Plot



Corresponding bode plot





## pzfilter

Filters the poles and zeroes according to the specified criteria. The `pzfilter` function works only on pole zero simulation data. This function is available only in the MDL mode.

### Syntax

```
pzfilter( poles, zeroes, maxfreq, reldist, absdist, minq)
pzfilter( poles=poles , zeroes=zeroes , maxfreq=maxfreq , reldist=reldist ,
          absdist=absdist , minq=minq )
```

### Arguments

<i>poles</i>	The poles.
<i>zeroes</i>	The zeroes.
<i>maxfreq</i>	The frequency up to which the poles and zeroes are plotted.
<i>reldist</i>	The relative distance between the pole and zero. Pole-zero pairs with a relative distance lower than the specified value are not plotted.
<i>absdist</i>	The absolute distance between the pole and zero. Pole-zero pairs with an absolute distance lower than the specified value are not plotted.
<i>minq</i>	The minimum Q-factor. Pole-zero pairs with a Q-factor less than the specified value are not cancelled. The equations that define the Q-factor of a complex pole or zero are described in the section below.

**Note:** If you do not specify *maxfreq*, *reldist*, *absdist*, or *minq*, `pzfilter` filters out the poles and zeroes with a frequency higher than 10 GHz (default value of *maxfreq*).

## Equations Defining the Q-Factor of a Complex Pole or Zero

$$\operatorname{Re}(X) < 0.0 \quad Q = 0.5 \times \sqrt{[\operatorname{Im}(X) / \operatorname{Re}(X)]^2 + 1}$$

$$\operatorname{Re}(X) = 0 \quad \text{UNDEFINED}$$

$$\operatorname{Re}(X) > 0.0 \quad Q = -0.5 \times \sqrt{[\operatorname{Im}(X) / \operatorname{Re}(X)]^2 + 1}$$

## Filtration Rules

- Real poles can be cancelled only by real zeroes. A real pole  $P$  is cancelled by a real zero  $Z$  if the following equation is satisfied:

$$|P - Z| < \text{absdist} + \frac{|P + Z|}{2} \times \text{reldist}$$

- Complex poles and zeroes always occur in conjugated pairs. A pair of conjugated poles can only be canceled by a pair of conjugated zeroes. A pole pair  $P1=a+jb$ ,  $P2=a-jb$  is cancelled by a zero pair  $Z1=c+jd$ ,  $Z2=c-jd$ , if the following equation is satisfied:

$$|P1 - Z1| = |P2 - Z2| = \sqrt{(a-c)^2 + (b-d)^2} < \text{absdist} + \frac{|a+c|}{2} \times \text{reldist}$$

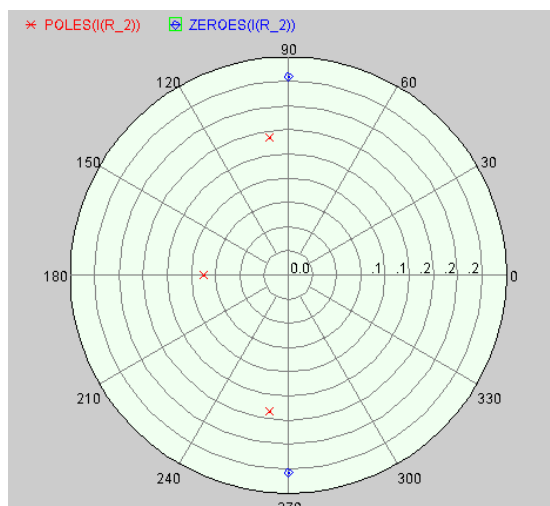
- Poles in the right-half plane are never cancelled because they show the instability of the circuit.

## Example

The values  $\text{poles}=\text{POLES}\langle \text{I}\langle \text{R}_2 \rangle \rangle$ ,  $\text{zeroes}=\text{ZEROES}\langle \text{I}\langle \text{R}_2 \rangle \rangle$ ,  $\text{absdist}=0.05$ , and  $\text{minq}=10000$  filters pole-zero pairs with a relative distance of less than 0.05 Hz from the plot

on the left side. In the filtered plot shown on the right side, two pole-zero pairs have been filtered out.

### Filtered polar plot



### **r2d (radians-to-degrees)**

Converts a scalar or waveform expressed in radians to degrees.

#### **Syntax**

```
r2d( arg )  
r2d( arg=arg )
```

#### **Arguments**

<i>arg</i>	The signal.
------------	-------------

#### **Example**

```
export real myr2d = r2d( 3.14 )
```

returns

```
myr2d = 179.909
```

## re

Returns the real portion of a complex number.

### Syntax

```
re( arg )  
re( arg=arg ) ?
```

### Arguments

*arg*                                      The complex number.

### Examples

```
export real myre = re( cplx(1,2) )
```

returns

```
myre = 1
```

```
export real real_s11 = re( s(1,1) )
```

returns

```
real_s11 = 0.682203
```

### **real**

Creates a real number from an integer number.

### **Syntax**

```
real( arg )
```

```
real( arg=arg )
```

### **Arguments**

<i>arg</i>	The integer.
------------	--------------

## risetime

Returns the rise time for a signal measured between percent low and percent high of the difference between the initial and final value.

**Note:** You can use the `risetimes` function to obtain the rise time for all edges instead of a single edge returned by the `risetime` function.

## Syntax

```
risetime( sig[, initval[, finalval [, inittype[, finaltype[, theta1[,  
    theta2[, xtol[, ytol[, accuracy]]]]]]]] )  
  
risetime( sig=sig, initval=initval, finalval=finalval [, inittype=inittype] [,  
    finaltype=finaltype] [, theta1=theta1] [, theta2=theta2] [, xtol=xtol] [,  
    ytol=ytol] [, accuracy=accuracy] )
```

## Arguments

<i>sig</i>	The signal.
<i>initval</i>	The X value (if inittype is 'x') or Y value (if inittype is 'y') that starts the rise time interval. The measurement is always done in ordinate values.
<i>finalval</i>	The X value (if inittype is 'x') or Y value (if inittype is 'y') that ends the rise time interval.
<i>inittype</i>	When 'x', the initial value is an X value. When 'y', the initial value is a Y value. Valid values: 'x', 'y' Default: 'y'
<i>finaltype</i>	When 'x', the final value is an X value. When 'y', the final value is a Y value. Valid values: 'x', 'y' Default: 'y'
<i>theta1</i>	The percent low. Default: 10
<i>theta2</i>	The percent high. Default: 90

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

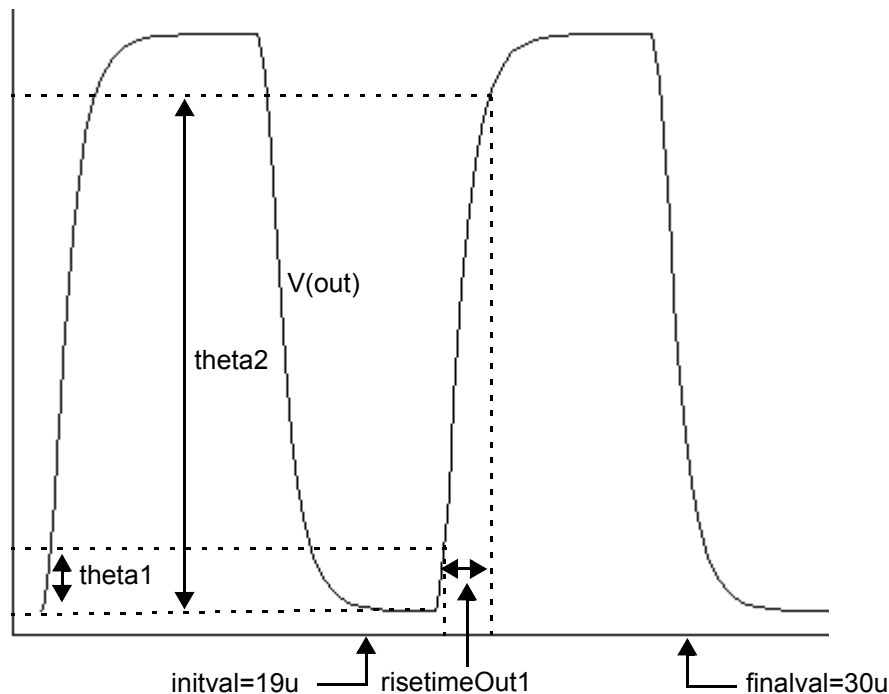
---

<i>xtol</i>	The relative tolerance in percentage value in the X direction. Default: 1
<i>ytol</i>	The relative tolerance in percentage value in the Y direction. Default: 1
<i>accuracy</i>	Specifies whether the function should use interpolation, or use iteration controlled by the absolute tolerances to calculate the value. 'interp' directs the function to use interpolation, and 'exact' directs the function to consider the xtol and yval values. Data types: name for scalar Valid values: 'interp', 'exact' Default: 'exact'

### Example 1

```
export real risetimeOut1 = risetime( sig=V(out), initval=19u, finalval=30u,  
inittype='x', finaltype='x', theta1=10, theta2=90)
```

The following diagram illustrates how the result from the above example is determined.

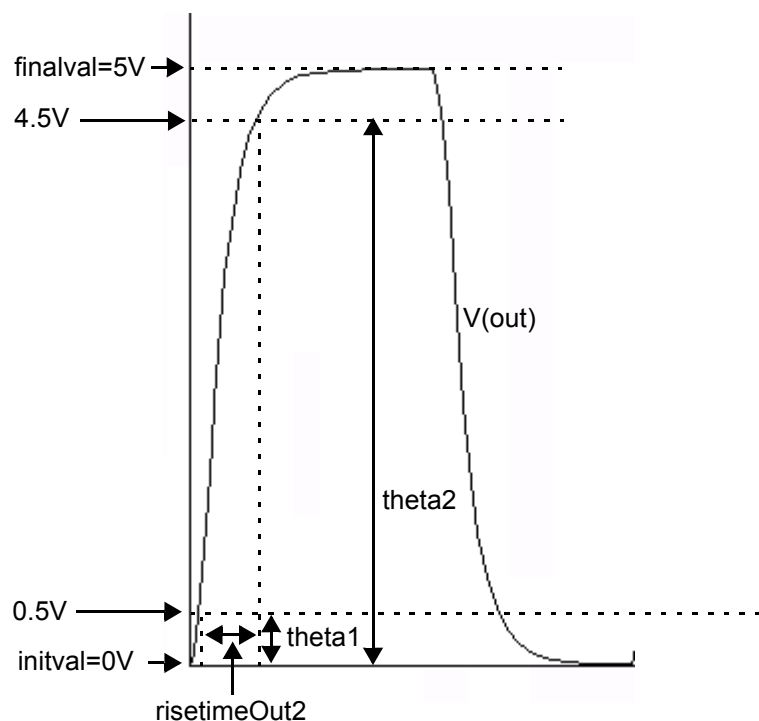




### Example 2

```
export real risetimeOut2 = risetime( sig=V(out), initval=0V, finalval=5V,  
inittype='y', finaltype='y', theta1=10, theta2=90)
```

The following diagram illustrates how the result from the above example is determined.



## rmsnoise

Returns the root mean square noise of a signal. The root mean square is defined as:

$$\text{rmsnoise} = \sqrt{\text{integral}[\text{Sig}(t) * \text{Sig}(t)]}$$

### Syntax

```
rmsnoise( sig:param )
```

```
rmsnoise( sig=sig:param )
```

### Arguments

*sig*                                      The signal.

*param*                                      The parameter that refers to the noise to be provided. Possible values are `out` (output noise), `in` (input noise), `F` (noise factor), `NF` (noise figure) and `gain` (circuit gain).

### Example

```
export real total_noise = rmsnoise ( myNoise:out )
```

SpectreMDL returns the total output referred noise from the pre-defined noise analysis `myNoise`.

### **rms (root-mean-square)**

Returns the root mean square of a signal.

#### **Syntax**

```
rms ( sig )  
rms ( sig=sig )
```

#### **Arguments**

<i>sig</i>	The signal.
------------	-------------

#### **Example**

```
export real rmsOut = rms( V(out))
```

## round

Rounds a number to the closest integer value.

### Syntax

```
round( arg )  
round( arg=arg )
```

### Arguments

<i>arg</i>	The number.
------------	-------------

### Example

```
export real myround = round( 1.234 )
```

returns

```
myround = 1
```

### S

Returns the complex value of Scattering (S) parameter of a network. Only available from sp analysis results.

#### Syntax

```
s( rowindex, colindex )  
s(rowIndex=rowIndex, colIndex=colIndex )
```

#### Arguments

*rowindex*                      The scattering matrix row index.

*colindex*                      The scattering matrix column index.

In general, the 2-port network S-parameter definitions are:

s(1,1)      input port voltage reflection coefficient

s(1,2)      reverse voltage gain

s(2,1)      forward voltage gain

s(2,2)      output port voltage reflection coefficient

If used with the functions like db, angle, re or im, the real number value is returned:

db(s(1,1))    returns the db of s(1,1)

angle(s(1,1)) returns the phase of s(1,1) in degrees

ph(s(1,1)) returns the phase of s(1,1) in radians

re(s(1,1))    returns the real part of s(1,1)

im(s(1,1))    returns the image part of s(1,1)

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

### Example

```
export real ft = cross( sig = ( db(s(2,1) ) ), dir = 'cross, n=1 )
```

returns

```
ft = 3.68369e+09
```

## sample

Returns a waveform or an array representing a sample of the signal based on step size or points per decade.

### Syntax

```
sample( sig, from, to, by, type )
```

```
sample( sig=sig, from=from, to=to, by=by, type=type )
```

### Arguments

<i>sig</i>	The signal.
<i>from</i>	The X-axis value at which the sampling begins.
<i>to</i>	The X-axis value at which the sampling stops.
<i>type</i>	Specifies whether the sample should be linear or logarithmic. Valid values: 'linear', 'log' Default value: 'linear'
<i>by</i>	If type is 'linear', specifies the step size for the sample. If type is 'log', specifies the points per decade.

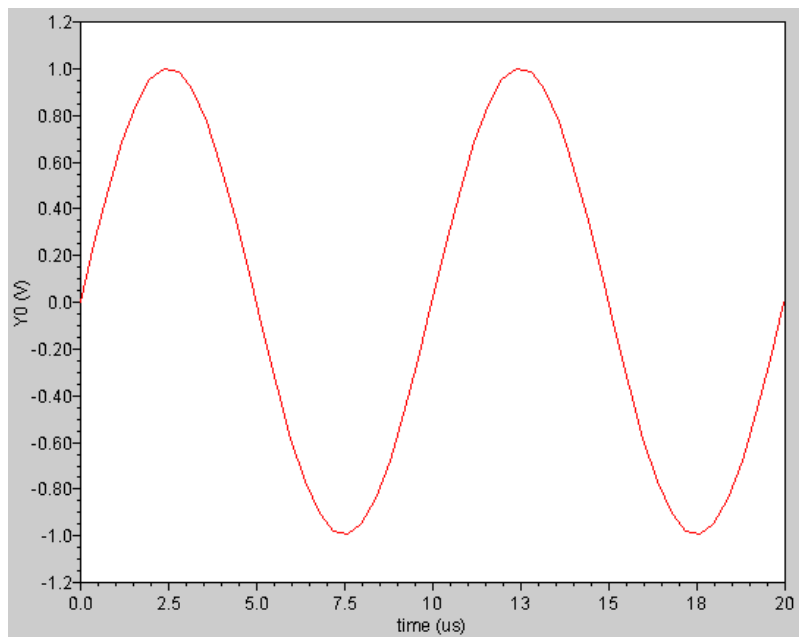
### Example 1

```
export real sampleOut = sample(sig=V(2), from=7.5us, to=18us, by=5us, type='linear')
```

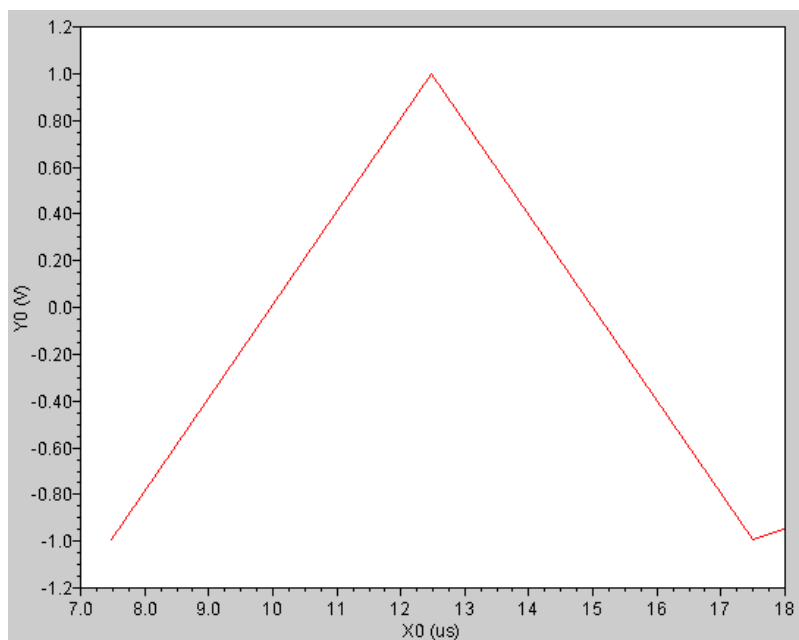
## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

transforms the following input signal



into the following output signal





## Example 2

```
export real v2smp1[] = sample(sig=V(2), from=10n, to=40n, by=0.1n)
```

The above example samples signal  $V(2)$  into an array as shown below:

```
v2smp1[0] = 1.08957e-10  
v2smp1[1] = 1.21644e-08  
v2smp1[2] = 1.8  
v2smp1[3] = 2.39729e-07  
v2smp1[4] = 1.8  
...
```

## settlingtime

Calculates the time required by a signal to settle at a final value within a specified limit.

### Syntax

```
settlingtime( sig, initval, finalval, inittype, finaltype, theta)
settlingtime( sig=sig, initval=initval, finalval=finalval, inittype=inittype,
              finaltype=finaltype, theta=theta )
```

### Arguments

<i>sig</i>	The signal.
<i>initval</i>	The starting value for the measurement.
<i>finalval</i>	The final value for the measurement.
<i>inittype</i>	Specifies whether <i>initval</i> is an X-axis or Y-axis value. If it is 'x', <i>initval</i> is the X-axis value. If it is 'y', <i>initval</i> is the Y-axis value.
<i>finaltype</i>	Specifies whether <i>finalval</i> is an X-axis or Y-axis value. If it is 'x', <i>finalval</i> is the X-axis value. If it is 'y', <i>finalval</i> is the Y-axis value and the signal settles at <i>finalval</i> until the end.
<i>theta</i>	Percentage of ( <i>finalval</i> - <i>initval</i> ) within which the signal has to settle.

### Example

```
export real settlingTimeOut = settlingtime( sig=V(out), initval=0, finalval=1.0,
inittype='y, finaltype='x, theta=5 )
```

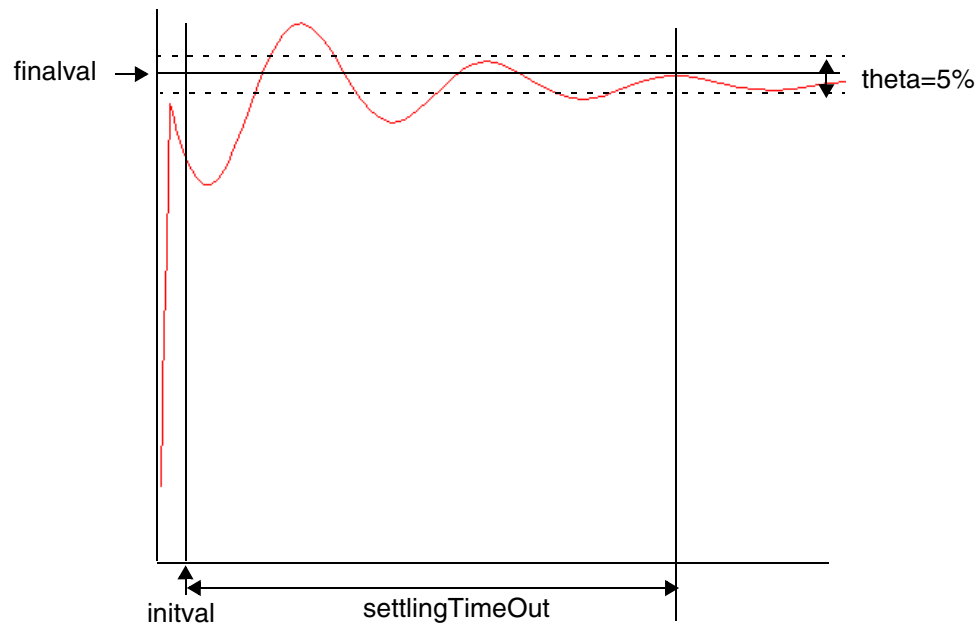
### returns

```
settlingTimeOut = 3.7185180980334184E-5sec
```

The following diagram illustrates how the result from the above example is determined.

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---



### sign

Returns a value that corresponds to the sign of a number.

#### Syntax

```
sign( arg )  
sign( arg=arg )
```

#### Arguments

<i>arg</i>	The number whose sign is to be returned. If the number is greater than zero, the <code>sign</code> function returns 1.0; if the number is equal to zero, the <code>sign</code> function returns 0.0; if the number is less than zero, the <code>sign</code> function returns -1.0.
------------	--

#### Example

```
sign( -17.3)
```

returns

```
-1.0
```

## **sin**

Returns the sine of a signal.

### **Syntax**

```
sin( arg )  
sin( arg=arg )
```

### **Arguments**

<i>arg</i>	The signal.
------------	-------------

### **Example**

```
export real mysin = sin( 1 )
```

returns

```
mysin = 0.84
```

## **sinh**

Returns the hyperbolic sine of a signal.

### **Syntax**

```
sinh( arg )  
sinh( arg=arg )
```

### **Arguments**

<i>arg</i>	The signal.
------------	-------------

### **Example**

```
export real mysinh = sinh( 1 )
```

returns

```
mysinh = 1.18
```

## size

Returns the size of an array or the number of points in a waveform.

### Syntax

```
size( arg, [, from [, to ] ] )  
size( arg=arg [, from=from] [, to=to] )
```

### Arguments

<i>arg</i>	The signal or the array.
<i>from</i>	The starting abscissa.
<i>to</i>	The ending abscissa

### Example 1

```
run tran( step=1e-09, pstep=1e-09, stop=9e-02 )  
export real signalNum = size( V(R1), 8.9e-022, 9e-02)
```

#### returns

```
signalNum = 108018
```

### Example 2

```
export real cro = crosses(sig=(V(R1))-(1/ 2),dir='cross,n=int(1))  
export real num = size(cro)
```

#### returns

```
cro[0]      = 8.33333e-07  
cro[1]      = 4.16583e-06  
cro[2]      = 1.08334e-05  
cro[3]      = 1.41666e-05  
num         = 4
```

### Example 3

```
export real arr [] = {1.1, 2.2}  
export real num = size(arr)
```

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

returns

```
arr[0]    = 1.1
arr[1]    = 2.2
num       = 2
```



## slewrates

Computes the average rate at which the buffer expression changes from percent low to percent high of the difference between the initial value and the final value.

### Syntax

```
slewrates( sig[, initval[, finalval [, inittype[, finaltype[, theta1[,  
            theta2[, xtol[, ytol[, accuracy]]]]]]]] )  
  
slewrates( sig=sig, initval=initval, finalval=finalval [, inittype=inittype] [,  
            finaltype=finaltype] [, theta1=theta1] [, theta2=theta2] [, xtol=xtol] [,  
            ytol=ytol] [, accuracy=accuracy] )
```

### Arguments

<i>sig</i>	The signal.
<i>initval</i>	The X value (if inittype is 'x') or Y value (if inittype is 'y') that starts the rise time interval.
<i>finalval</i>	The X value (if inittype is 'x') or Y value (if inittype is 'y') that ends the rise time interval.
<i>inittype</i>	When 'x', the initial value is an X value When 'y', the initial value is a Y value. Valid values: 'x', 'y' Default: 'y'
<i>finaltype</i>	When 'x', the final value is an X value. When 'y', the final value is a Y value. Default: 'y'
<i>theta1</i>	The percent low. Default: 10
<i>theta2</i>	The percent high. Default: 90
<i>xtol</i>	The relative tolerance in percentage value in the X direction. Default: 1

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

<i>ytol</i>	The relative tolerance in percentage value in the Y direction. Default: 1
<i>accuracy</i>	Specifies whether the function should use interpolation, or use iteration controlled by the absolute tolerances to calculate the value. 'interp directs the function to use interpolation, and 'exact directs the function to consider the xtol and yval values. Data types: name for scalar Valid values: 'interp, 'exact Default: 'exact

### Example

A statement like

```
export real slewrate1 = slewrate( V(out), 20ns, 60ns )
```

produces a result similar to

```
6.337662406448401E7V/s
```

## slice

Returns the slice of an array.

### Syntax

```
slice( arg, from, to, step )
```

```
slice( arg=arg, from= from, to = to, step =step)
```

### Arguments

array	A user-defined array, or an array that comes from the built-in function.
from	Array starting subscript.
to	Array ending subscript.
step	Array step.

### Example

```
real arr[]={1.0,2.0,3.0,4.0,5.0,6.0,7.0}  
export real myslice1=slice(arr,from=2,to=5,step=1)  
export real myslice2=slice(arr,from=2,to=5,step=2)
```

### returns

```
myslice1[0]    =  2  
myslice1[1]    =  3  
myslice1[2]    =  4  
myslice1[3]    =  5
```

```
myslice2[0]    =  2  
myslice2[1]    =  4
```

### snr

Calculates the signal to noise ratio from a complex frequency based signal.

### Syntax

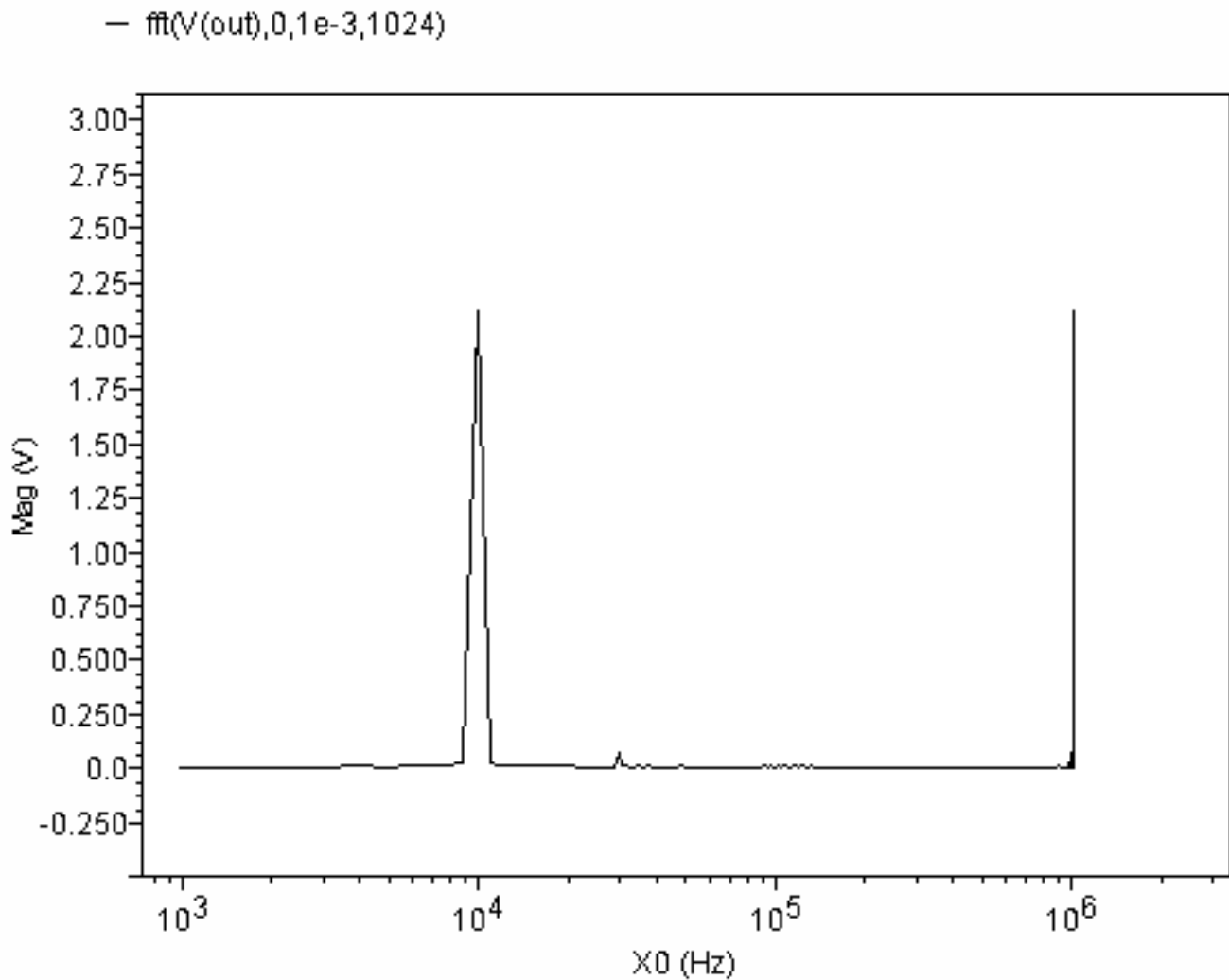
```
snr( sig, sig_from, sig_to, noise_from, noise_to )  
snr( sig=sig, sig_from=sig_from, sig_to=sig_to, noise_from=noise_from,  
      noise_to=noise_to )
```

### Arguments

<i>sig</i>	The signal.
<i>sig_from</i>	The left window border of the signal. The <i>sig_from</i> value must be greater than or equal to <i>noise_from</i> .
<i>sig_to</i>	The right window border of the signal. The <i>sig_to</i> value must be less than or equal to <i>noise_to</i> .
<i>noise_from</i>	The left window border of the noise.
<i>noise_to</i>	The right window border of the noise.

### Example

You have the following frequency plot.



To determine the signal-to-noise ratio, you use the statement

```
export real snr(fft(V(out),0,1e-3,1024),9e3,11e3,1,500e3)
```

which, in this case, returns

```
29.268026738835342dB
```

## **sqrt**

Returns the square root of a signal.

### **Syntax**

```
sqrt( arg )  
sqrt( arg=arg )
```

### **Arguments**

<i>arg</i>	The signal.
------------	-------------

### **Example**

```
export real mysqrt = sqrt( 4 )
```

returns

```
mysqrt = 2
```

## stathisto

Creates a histogram from a signal.

The stathisto function is available from the calculator. It is not supported within a Spectre MDL control file since it returns a scalar and not a waveform.

### Syntax

```
stathisto( sig [, nbins][, min][, max][, innerswpval] )  
stathisto(sig=sig [, nbins=nbins] [, min=min] [, max=max]  
          [, innerswpval=inner swpval])
```

### Arguments

<i>sig</i>	The waveform.
<i>nbins</i>	The number of bins to be created.
<i>min</i>	The value that specifies the smaller end point of the range of values included in the histogram.
<i>max</i>	The value that specifies the larger end point of the range of values included in the histogram.
<i>innerswpval</i>	The inner-most sweep parameter in the dataset. You use this parameter to slice through parametric waveforms to extract the data for the histogram. Default: The first available value of time in the dataset.

### Example

Assume that you have the results of running a Monte Carlo analysis on top of a transient analysis, so that the inner-most swept variable is time. Now, for the particular value of time specified by the *innerswpval* argument specification, the *stathisto* function creates a histogram by analyzing all the Monte Carlo iterations and extracting from each one the value of the signal at the specified time.

For example, to create a histogram for the time 100ns, you might use the following statement.

```
stathisto(I(V10\:p), innerswpval=100e-9)
```

To create a histogram for the time 650ps, you might use the following statement.

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
stathisto(I(V10\:p), innerswpval=.65e-9)
```



## stddev

Returns the standard deviation of a signal. Standard deviation is defined as follows:

```
sqrt( variance(N) )
```

## Syntax

```
stddev( arg )
```

```
stddev( arg=arg )
```

## Arguments

<i>arg</i>	The signal.
------------	-------------

## sum

Returns the sum value of an array.

### Syntax

```
sum( arg )  
sum( arg=arg )
```

### Arguments

<i>arg</i>	A user-defined array, or an array that comes from the built-in function.
------------	--

### Example

```
real arr[ ] = {1.0, 2.0, 3.0}  
export real mysum=sum(arr)
```

### returns

```
mysum=6.0
```

## system

Returns a string, which is the output of *command* executed by shell.

### Syntax

**system** ( *command* )

**system** ( **command=***command* )

### Arguments

*command*                      A user-specified shell command.

### Example

```
string d1=system( "date +\"%y%m%d%H%M\"" );  
print fmt("%s", d1) addto="aa.data"
```

### returns

1302130702

**Note:** The function should only be used at the top-level MDL file, or before the `run` command in an alias measurement, and not during analysis.

## tan

Returns the tangent of a signal.

### Syntax

```
tan( X )
```

```
tan( X=X )
```

### Arguments

$X$	The scalar or signal.
-----	-----------------------

### Example

```
export real mytan = tan( 1 )
```

returns

```
mytan = 1.56
```

## **tanh**

Returns the hyperbolic tangent of a signal.

### **Syntax**

```
tanh( arg )  
tanh( arg=arg )
```

### **Arguments**

*arg*                                      The scalar or signal.

### **Example**

```
export real mytanh = tanh( 1 )
```

returns

```
mytanh = 0.76
```

## trim

Returns the portion of a signal between two points along the abscissa.

### Syntax

```
trim( sig[, from[, to]] )  
trim( sig=sig [, from=from] [, to=to] )
```

### Arguments

<i>sig</i>	The signal. Data types: real for scalar
<i>from</i>	The starting abscissa. Data types: real for scalar
<i>to</i>	The ending abscissa. Data types: real for scalar

### Example 1

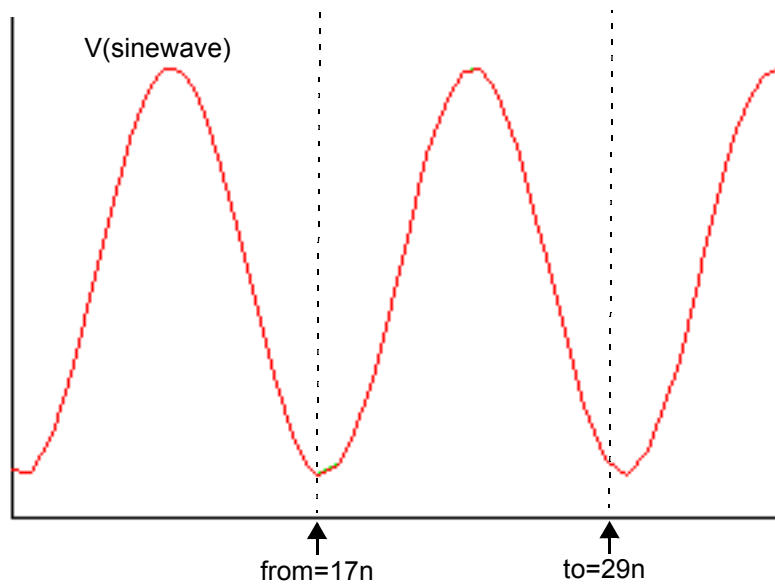
The following example works in an MDL control file.

```
export real trimOut = max ( trim( sig=V(sinewave), from=17n, to=29n ) )
```

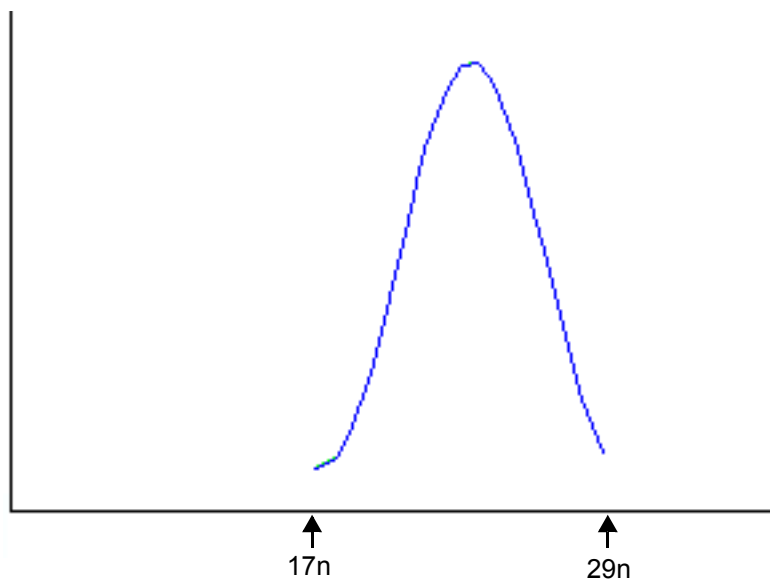
In Virtuoso Visualization and Analysis XL,

```
trim ( sig=V(sinewave), from=17n, to=29n )
```

transforms the following input signal



into the following output signal



## V

Returns the voltage of a net.

### Syntax

`V(node)`

`V(node, node)`

`V(Instname:term)`

`V(Instname:term, Instname:term)`

`V(devname)` //which outputs voltage value between positive and negative terminals of a 2-terminal device.

`V(node)` takes precedence over `V(devname)`. It is illegal to apply the `V` probe function to a `devname` and a `node`, or a pair of `devnames`. `V` can be uppercase or lowercase.

### Arguments

<i>node</i>	The net name with or without the hierarchical path.
<i>Instname</i>	The instance name. It can be a N-terminal device instance or a N-terminal subcircuit instance.
<i>devname</i>	The 2-terminal device name (not including the 2-terminal subcircuit instance).
<i>term</i>	The terminal name or terminal index of a device or a subcircuit instance.

### Examples

```
V(p,n)           // Returns the voltage between nodes p and n.
V(Rload:1)        // Returns the voltage from terminal Rload:1 to ground.
V(I0:q)           // Returns the voltage from terminal I0:q to ground.
V(I0:q,I1:y)      //Returns the voltage between terminal I0:q and terminal I1:y.
```



## variance

Returns the statistical variance of a signal. The variance is defined as follows:

$$1/(N-1) * ( (X1 - \text{mean})^2 + (X2 - \text{mean})^2 + \dots + (XN - \text{mean})^2 ) ,$$

where  $N$  is the total number of samples.

## Syntax

**variance**( *arg* )

**variance**( **arg=***arg* )

## Arguments

*arg*                                      The scalar or signal.

window

Applies the specified window to a signal.

Syntax

```
window( arg[, window] )  
fft( arg=arg[, window=window] )
```

Arguments

<i>arg</i>	The signal.
<i>window</i>	<p>The window to be applied.</p> <p>Valid values: 'rectangular', 'bartlett', 'bartlettthann', 'blackman', 'blackmanharris', 'cosine2', 'cosine4', 'extcosbell', 'flattop', 'halfcyclesine', 'half3cyclesine', 'halfcyclesine3', 'half6cyclesine', 'halfcyclesine6', 'hamming', 'hanning', 'nuttall', 'parzen', 'triangular'</p> <p>Default: 'rectangular'</p>

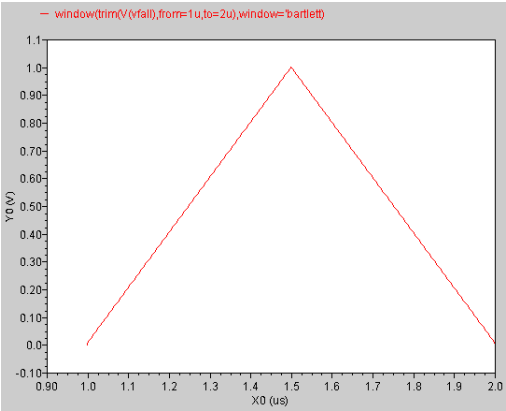
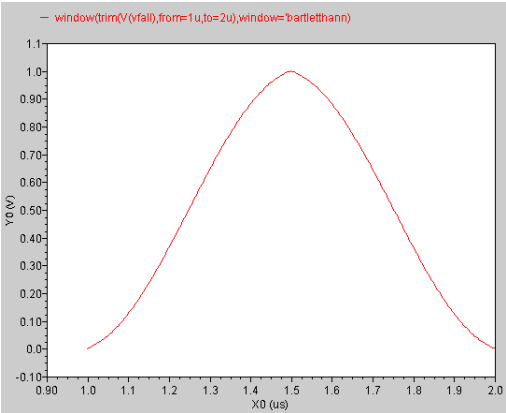
Equations and Examples

This section describes the equations used by each type of window and then shows an example. In the equations:

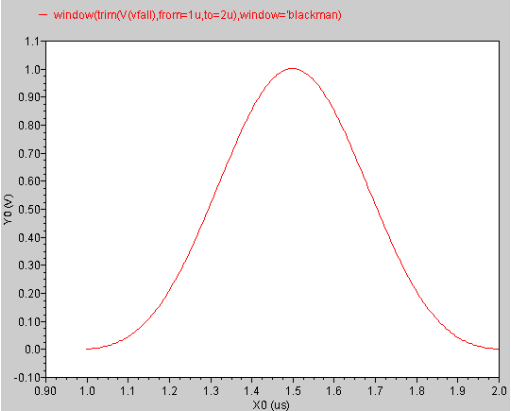
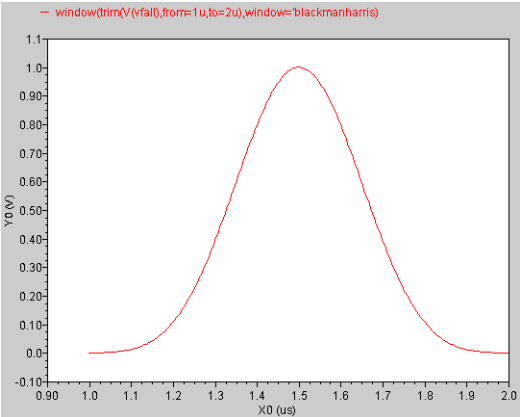
- N* = total number of waveform points
- n* = current waveform point

Window	Equation and Example	Where
'rectangular	$w(n) = 1$	

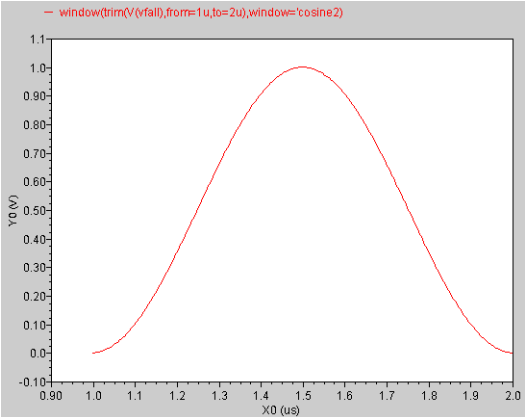
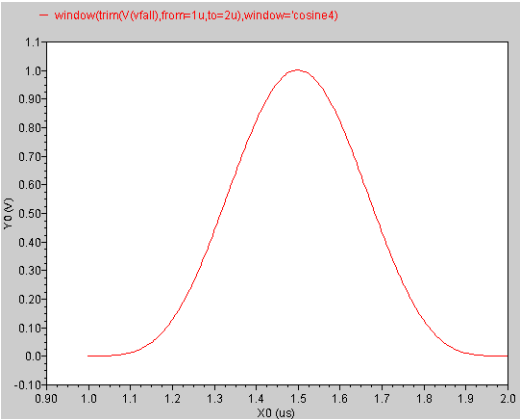
# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

Window	Equation and Example	Where
'bartlett	$w(n) = 1 - abs\left(2 \times \frac{n}{N} - 1\right)$ $w(n) = 0$ 	$0 \leq n \leq N$  otherwise
'bartlettthann	$w(n) = 0.62 - 0.48 \times abs\left(\frac{n}{N} - 0.5\right) + 0.38$ $\times \cos\left(2 \times \pi \times \left(\frac{n}{N} - 0.5\right)\right)$ $w(n) = 0$ 	$0 \leq n \leq N$  otherwise

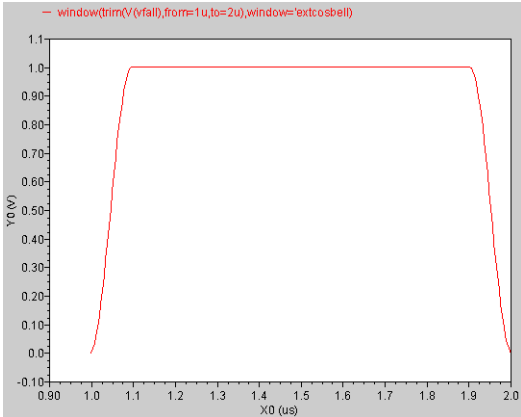
# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

Window	Equation and Example	Where
'blackman	$w(n) = 0.42 - 0.50 \times \cos\left(2 \times \pi \times \frac{n}{N}\right) + 0.08 \times \cos\left(4 \times \pi \times \frac{n}{N}\right)$ $w(n) = 0$ 	$0 \leq n < N$  otherwise
'blackmanharris	$w(n) = 0.35875 - 0.48829 \times \cos\left(2 \times \pi \times \frac{n}{N}\right) + 0.14128$ $\times \cos\left(4 \times \pi \times \frac{n}{N}\right) - 0.01168 \times \cos\left(6 \times \pi \times \frac{n}{N}\right)$ $w(n) = 0$ 	$0 \leq n < N$  otherwise

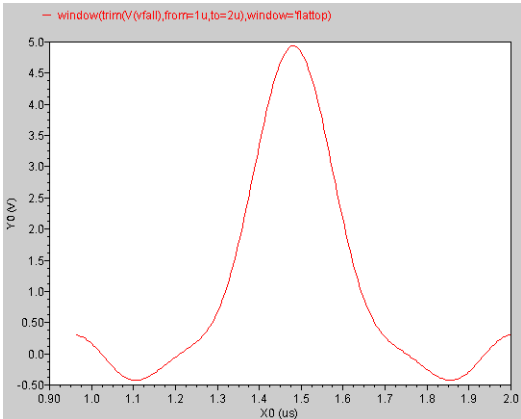
Spectre Circuit Simulator Measurement Description Language User Guide and Reference

Window	Equation and Example	Where
'cosine2	$v(n) = 0.5 - 0.5 \times \cos\left(2 \times \pi \times \frac{n}{N}\right)$ $w(n) = 0$ 	$0 \leq n < N$  otherwise
'cosine4	$v(n) = \left(0.5 - 0.5 \times \cos\left(2 \times \pi \times \frac{n}{N}\right)\right)^2$ $w(n) = 0$ 	$0 \leq n < N$  otherwise

Window	Equation and Example	Where
'extcosbell	$v(n) = 0.5 - 0.5 \times \cos\left(10 \times \pi \times \frac{n}{N}\right)$ $w(n) = 1$	$\text{abs}(n/N - 0.5) > 0.4$ <p>otherwise</p>

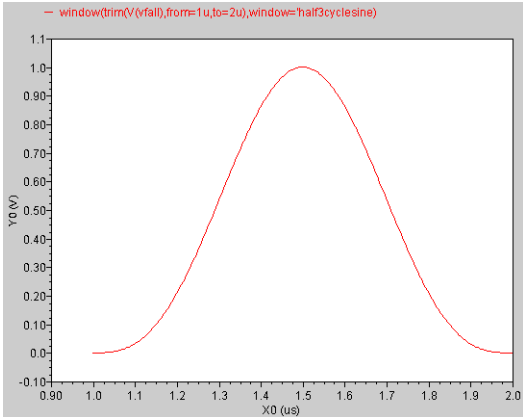
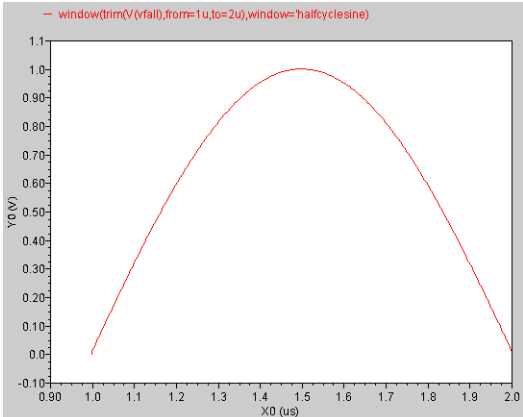


'flattop	$v(n) = 1 - 1.93 \times \cos\left(2 \times \pi \times \frac{n}{N}\right) + 1.29 \times \cos\left(4 \times \pi \times \frac{n}{N}\right) - 0.38 \times \cos\left(6 \times \pi \times \frac{n}{N}\right) + 0.322 \times \cos\left(8 \times \pi \times \frac{n}{N}\right)$ $w(n) = 0$	$0 \leq n \leq N$ <p>otherwise</p>
----------	--	------------------------------------

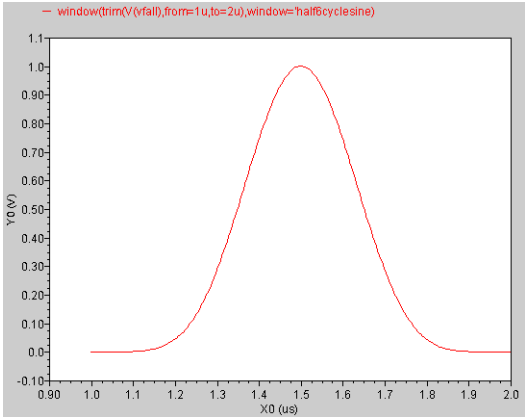


# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

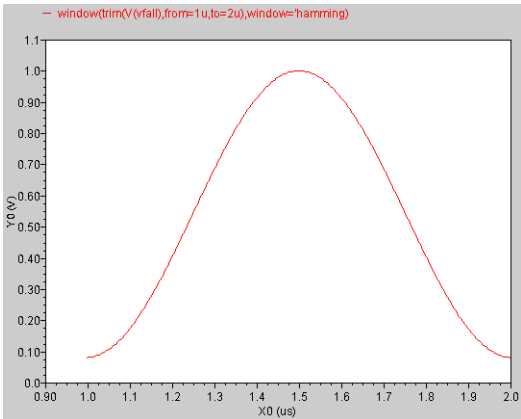
Window	Equation and Example	Where
'halfcyclesine	$w(n) = \sin\left('pi \times \frac{n}{N}\right)$ $w(n) = 0$	<div>0 ≤ n</div> <div>otherwise</div>
<div>'half3cyclesine</div> <div>and</div> <div>'halfcyclesine3</div>	$w(n) = \left(\sin\left('pi \times \frac{n}{N}\right)\right)^3$ $w(n) = 0$	<div>0 ≤ n</div> <div>otherwise</div>



Window	Equation and Example	Where
'half6cyclesine and 'halfcyclesine6	$w(n) = \left( \sin\left( \pi \times \frac{n}{N} \right) \right)^6$ $w(n) = 0$	0 ≤ n ≤ N  otherwise

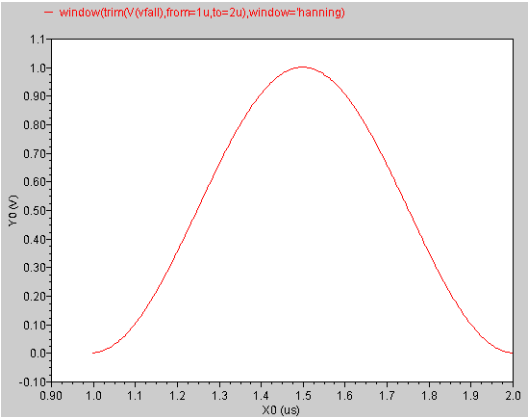
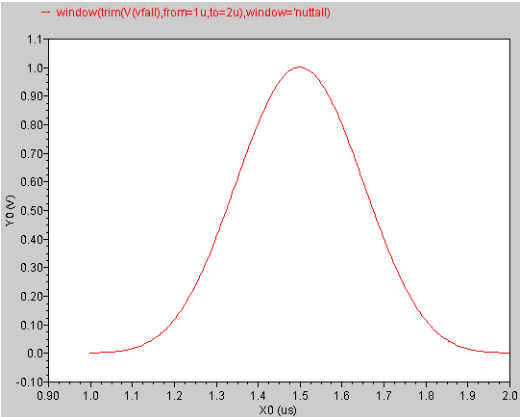


'hamming	$w(n) = 0.54 - 0.46 \times \cos\left( 2 \times \pi \times \frac{n}{N} \right)$ $w(n) = 0$	0 ≤ n ≤ N  otherwise
----------	---	----------------------------

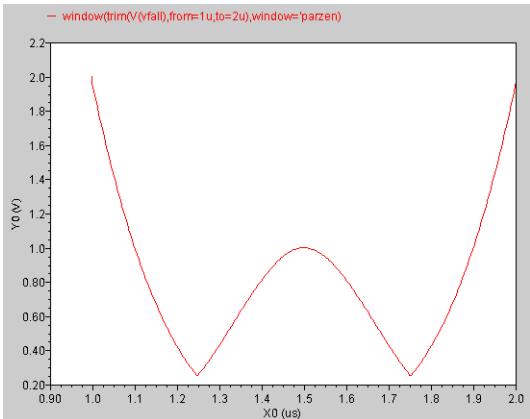




# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

Window	Equation and Example	Where
'hanning	$w(n) = 0.5 - 0.5 \times \cos\left(2 \times 'pi \times \frac{n}{N}\right)$ $w(n) = 0$ 	$0 \leq n < N$  otherwise
'nuttall	$w(n) = 0.3635819 - 0.4891775 \times \cos\left(2 \times 'pi \times \frac{n}{N}\right) + 0.1365995$ $\times \cos\left(4 \times 'pi \times \frac{n}{N}\right) - 0.0106411 \times \cos\left(6 \times 'pi \times \frac{n}{N}\right)$ $w(n) = 0$ 	$0 \leq n < N$  otherwise

Window	Equation and Example	Where
'parzen	$w(n) = 1 - 6 \times \text{abs}\left(2 \times \frac{n}{N} - 1\right) + 6 \times \text{abs}\left(2 \times \frac{n}{N} - 1\right)^3$ $w(n) = 2 \times \text{abs}\left(2 \times \frac{n}{N} - 1\right)$	$\text{abs}(2 \times n/N - 1) \leq 0.5$ otherwise



'triangular	Same as bartlett. For more information, see <a href="#">'bartlett</a> on page 219.
-------------	--

## xval

Returns the vector consisting of the abscissas of the points in the signal.

### Syntax

```
xval( arg )
```

```
xval( arg=arg )
```

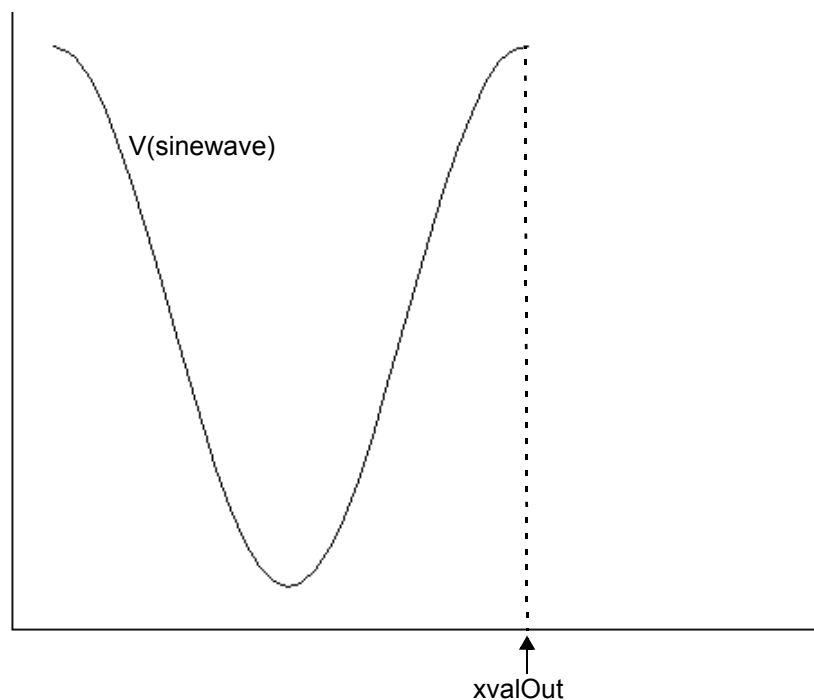
### Arguments

*arg*                                      The signal.

### Example 1

```
export real xvalOut = max ( xval( V(out) ) )
```

Returns the maximum X-axis value for `V(out)`.



### Example 2

```
export real xvalMax=xval(max(V(out)))
```

Returns the X-axis value of the point where  $V(out)$  is at its maximum voltage value.

# Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

## Y

Returns the complex value of the Admittance (Y) parameter of a network.

### Syntax

```
y( rowindex, colindex )  
y( rowIndex=rowIndex, colIndex=colIndex )
```

### Arguments

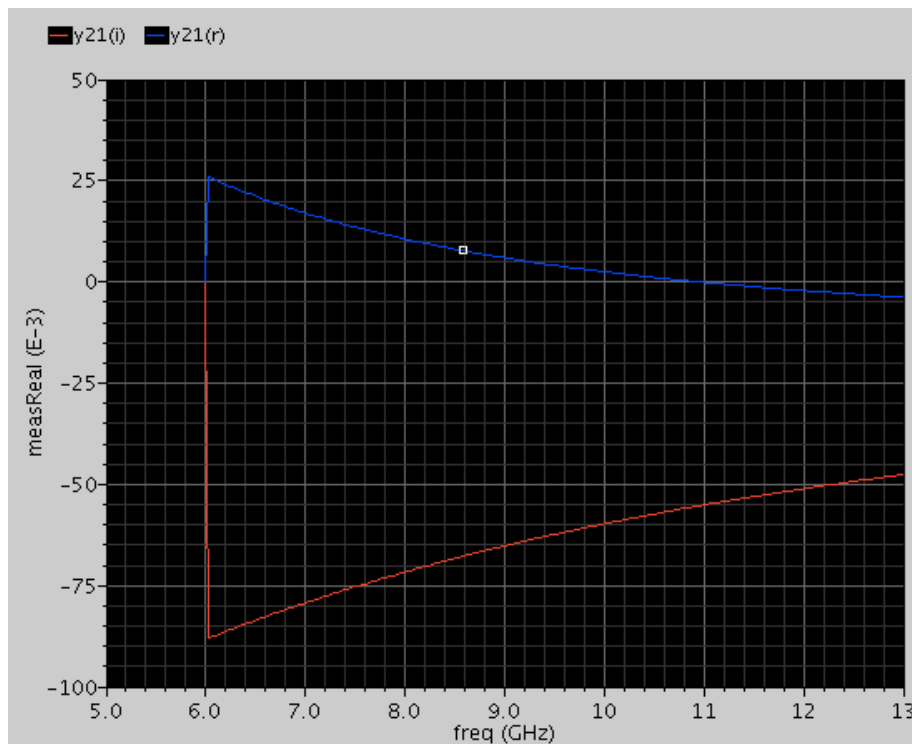
*rowindex*                      The admittance matrix row index. The value can be scalar.

*colindex*                      The admittance matrix column index. The value can be scalar.

### Example

```
real __mdlvar_13=_hprobe( "y21(r)", re(y(2,1)) )  
real __mdlvar_14=_hprobe( "y21(i)", im(y(2,1)) )
```

The output looks like below.



## yval

Returns a vector consisting of the ordinates of the points in the signal. This function can also calculate the ordinate value at a specified abscissa value.

### Syntax

```
yval( arg )  
yval( arg=arg )
```

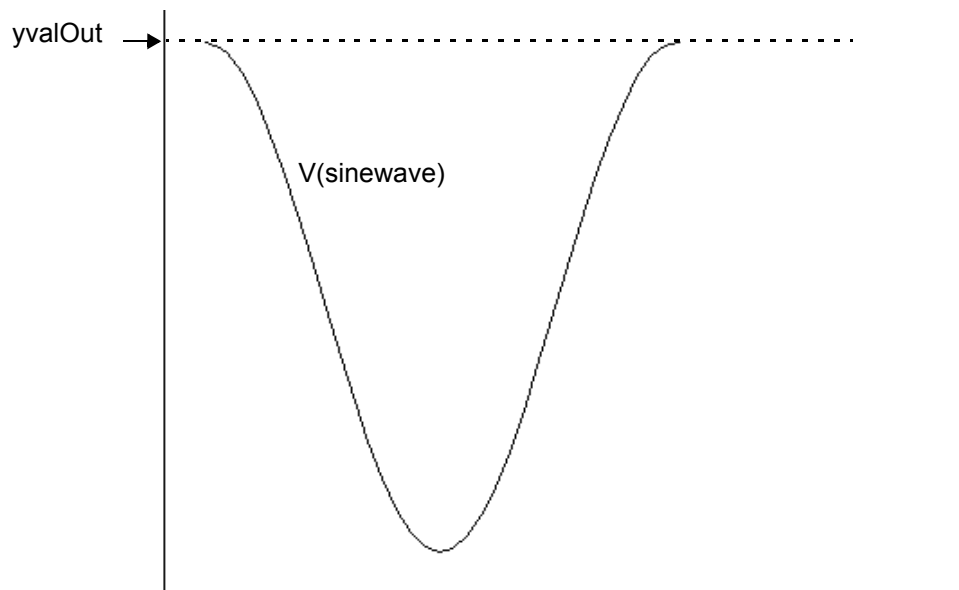
### Arguments

*arg*                                      The signal.

### Example 1

```
export real yvalOut = max ( yval( V(out) ) )
```

Returns the maximum Y-axis value for  $V(out)$ .



### Example 2

```
export real yvalOut1 = yval ( V(out)@ 100ns )
```

returns

```
3.467928474540306
```

## Z

Returns the complex value of Impedance (Z) parameter of a network.

### Syntax

```
z( rowindex, colindex )  
z(rowIndex=rowIndex, colIndex=colIndex )
```

### Arguments

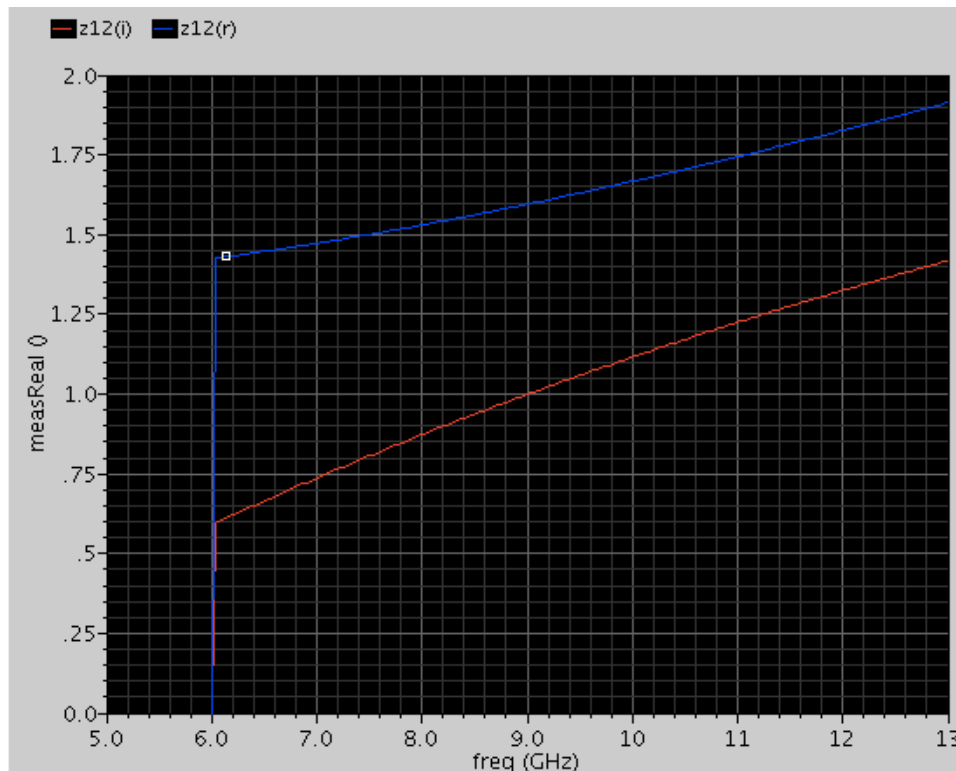
*rowindex*                      The impedance matrix row index. The value can be scalar.

*colindex*                      The impedance matrix column index. The value can be scalar.

### Example

```
real __mdlvar_7=_hprobe( "z22(r)", re(z(2,2)) )  
real __mdlvar_8=_hprobe( "z22(i)", im(z(2,2)) )
```

The output looks like below.





# SPICE Compatibility for Analyses

MDL supports the SPICE `.tran`, `.ac`, `.dc`, and `.op` analyses as described below.

- 1. MDL supports SPICE `.tran`, `.ac`, `.dc`, and `.op` analyses defined in the netlist.
- 2. Each `.tran`, `.ac`, `.dc`, or `.op` is mapped to a Spectre `tran`, `ac`, `dc`, or `op` analysis. Multiple SPICE analyses per analysis type are supported as well. Here is a list of Spectre mapped names:

SPICE Analysis Type	Spectre Mapped Name	
	First Run Name	Subsequent Run Names
.TRAN	timeSweep	tran2, tran3, ...,tran $n$
.AC	frequencySweep	ac2, ac3, ..., ac $n$
.DC	srcSweep	dc2, dc3, ..., dc $n$
.OP	opBegin	op2, op3, ..., op $n$

Therefore, the following statements are necessary in an MDL control file when running MDL on a netlist in SPICE format:

```
run opBegin           // runs first defined .OP analysis
run op2               // runs second defined .OP analysis
run srcSweep          // runs first defined .DC sweep analysis
run dc2               // runs second defined .DC sweep analysis
run frequencySweep    // runs first defined .AC analysis
run ac2               // runs second defined .AC analysis
run timeSweep         // runs first defined .TRAN analysis
run tran2             // runs second defined .TRAN analysis
```

- 3. When running MDL, the SPICE `.measure/.probe/.print` statements defined in the netlist are ignored. In other words, the MDL control file supersedes the SPICE `.measure/.probe/.print` statements defined in the netlist. However, if running Spectre but not MDL (for instance, using command line `spectre`

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

`spice.ckt`), the `.measure/.probe/.print` statements defined in the netlist supersede the `spice .mdl` control file.

4. The analyses can be parameterized in the MDL control file without modifying the netlist (see [Example 2](#) on page 234).
5. Mixed syntax netlists (containing both Spice and Spectre syntax analyses statements) are also supported in MDL (see [Example 3](#) on page 235).
6. Multi-level sweeps of `.tran/.dc/.ac` are not supported in MDL. For example, the following SPICE `.ac` statement is not supported.

```
.ac dec 20 1k 100k SWEEP V1 1 3 2
```

You can use MDL `foreach` statement to sweep `V1:mag` and run AC analysis inside the alias measurement (see [Example 4](#) on page 235). If you want to do a DC sweep, you need to sweep `V1:dc` and run a DC analysis.

### Example 1

For the SPICE analyses below:

\*in Netlist

```
.TRAN 1ns 5us
.TRAN 1ns 10us START=8us
.ac dec 10 1 10M
.ac dec 10 100M 1G
```

The equivalent MDL statements are:

//in MDL control file

```
run timeSweep           // runs first .tran with stop time of 5us
run tran2 (start=8us)    // runs second .tran with stop time of 10us
run frequencySweep       // runs first .ac sweeping from 1Hz to 10MHz
run ac2                  // runs second .ac sweeping from 100MHz to 1GMHz
```

### Example 2

The following example shows how to set a new value for the parameters of built-in analyses without modifying the netlist:

For the following SPICE analysis:

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

\*in Netlist

```
.TRAN 1ns 5us
```

The following statement tells the simulator to run transient analysis with the new stop time 100us:

//----in MDL control file -----

```
run timeSweep(stop=100u)           // set stop time to new value 100us
```

### Example 3

For mixed syntax statements like the following:

\*in Netlist

```
.op
.dc V1 0 1 0.1
simulator lang=spectre
mytran tran stop=1ms
```

The equivalent MDL statements are:

//in MDL control file

```
run opBegin           // runs .op pre-defined in SPICE syntax
run srcSweep          // runs .dc pre-defined in SPICE syntax
run mytran            // runs tran analysis pre-defined in SPECTRE syntax
```

### Example 4

To implement the following SPICE-like sweeps:

```
.ac dec 20 1k 100k SWEEP V1 1 3 2
```

You can use the following statements:

\*in Netlist

```
.ac dec 20 1k 100k
```

//in MDL control file

## Spectre Circuit Simulator Measurement Description Language User Guide and Reference

---

```
alias measurement acmeas {  
    run frequencySweep  
    <export variable block>  
}  
  
foreach V1:dc from swp (start=1, stop=3, step=2) {  
    run acmeas  
}
```

---

## SPICE Compatibility for options supported by MDL

---

### Support the SPICE option `.option co=<number>`

Spectre supports the SPICE option `.option co=<number>` as described below.

1. Spectre supports the SPICE option `co` to control the number of columns in the `.print` file by mapping it to the Spectre option `colslog`.
2. When `<number>` is defined between  $(n*15)$  and  $((n+1)*15)$ , then the  $(n+1)$  columns with 15 bit per column are printed to the `.print` file, where  $n$  columns is for defined variables and 1 column is for independent variable such as time for transient, frequency for AC analysis and swept parameter for DC analysis.
3. The default value is 80, printing 6 column data.
4. The minimum value is 31, printing 2 column data. If it is less than 31, Spectre uses the default value (80) and prints 6 column data.
5. The `co` option is not supported in `.alter` block.

#### Example 1

```
.option co=132
```

Spectre outputs 9 columns on a single line in `.print` file.

#### Example 2

```
Simulator lang=spectre  
Opt1 options colslog=60
```

Spectre outputs 5columns on a single line in `.print` file.

## Support equal interval output for `.print`

1. Spectre can print transient results to `.print` file in equal step (that are defined in `.tran` statement) by taking advantage of the spectre option `printstep`.
2. The value of `printstep` can be 1 | 0, true | false or yes | no. When `printstep=1/true/yes`, Spectre prints transient results in equal step as specified in `.tran` statement. When `printstep=0/false/no` (default), Spectre prints transient results in non-equal solver time.
3. The `printstep` option is not supported in `.alter` block.

### Example 1

```
.option printstep=1
.tran 1ns 20ns
.print tran v(1)
```

Spectre prints transient results in an equal interval of 1ns in `.print` file.

### Example 2

```
Simulator lang=spectre
Opt1 options printstep=yes
Simulator lang=spice
.tran 1u 5m
.print tran v(1)
```

Spectre prints transient results in an equal interval of 1us in `.print` file.