# Overview

## Introduction

Spice Array Linking Technology, SALT, is a modeling interface that links huge data sets to the IsSpice4 simulator. It was built using the Intusoft Code Model Software Development Kit, CMSDK. The interface is composed of two parts; the SALT kernel, array.dll, and a set of OLE2 automation servers. The SALT kernel interfaces and synchronizes data arrays from external software or hardware. The OLE2 automation servers represent a set of pre-defined Spice models which dynamically link to IsSpice4 at run time. They also link to other OLE clients using a rendering server which is supplied with SALT. The rendering server is used to view simulation results while the simulation is running using a unique running object implementation. Intusoft provides several servers, along with source code. These servers can be used as provided or, with the CMSDK, you can develop servers which are tailored to your specialized applications.

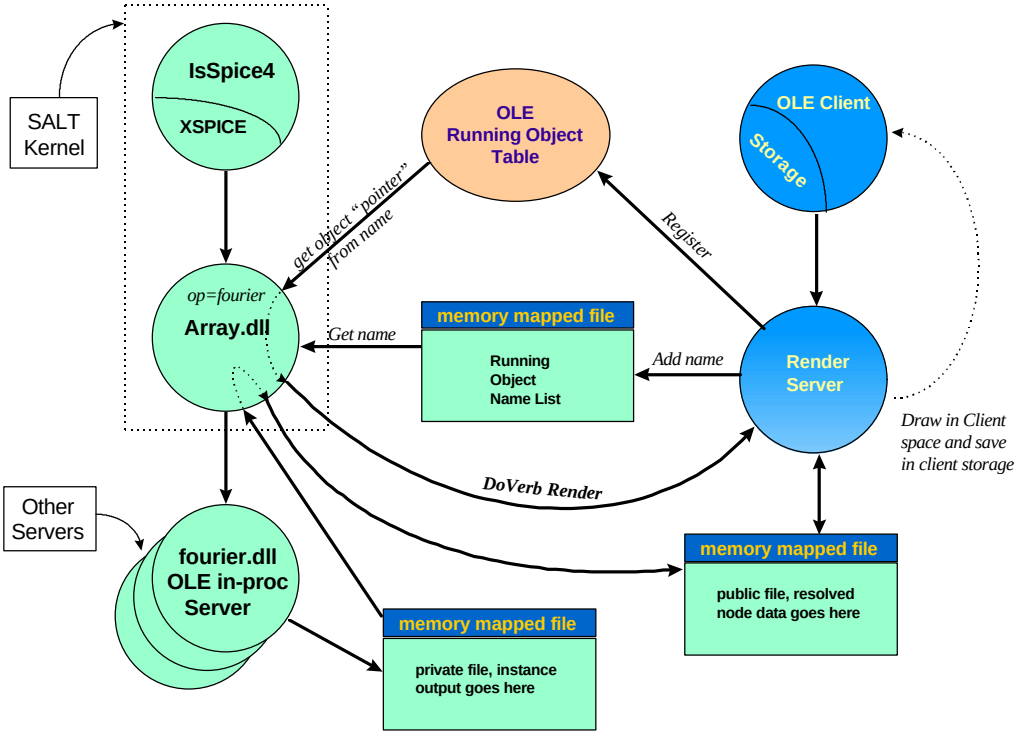
As the name implies, the type of data used by SALT is an array. The array is a matrix of time or frequency dependent data. It can be used to represent a sound, such as a WAV file, large sets of analog simulator data taken from IsSpice4, abstract data taken from hardware, or even links to other software. The possibilities are virtually endless.

SALT is ideal for the simulation of tracking and control problems which are common to robotics, guidance and navigation, image tracking, toys and household appliances. These problems span technologies from active and passive electronic components to signal and image processing and heuristic control algorithms. Linking the basic arrays and your own SALT servers, to the IsSpice4 simulation engine allows you to perform a comprehensive simulation of these problems. In some cases, the simulation can operate in real time and be used to collect data and test algorithm performance without first building prototype control and signal processing hardware.

## The SALT Interface

The SALT interface, shown below, takes advantage the power of Intusoft's CMSDK and Microsoft's OLE2 technology. The CMSDK created the SALT kernel which consists of a new node type, an Array node, and 5 array models; source, sink, array, mat_toa, and ato_mat. In the figure, the SALT kernel is represented by the IsSpice4 XSPICE/ array.dll path. This kernel uses two memory mapped files (MMFs), public and private, to process the array data as it is sent from model to model. The memory mapped file is named using a convention that makes it accessible to any other software that knows key features of the data.

OLE2 comprises the remainder of the SALT interface. Three of the models in the SALT kernel, source, sink and array, are OLE Automation Controllers. Depending on the model parameters provided, these OLE Automation Controllers make use of various OLE Automation Servers that are provided. This portion of the interface is represented by the array.dll/fourier.dll path. For the sake of simplicity only one sever is shown in the figure. Any number of servers may be present and, if present, would be placed along side the fourier.dll shown in the figure.




The SALT interface includes a viewing utility, render.dll, that is an OLE InProcess server, embeddable in any OLE client application. In the figure, this portion of the SALT interface is represented by the Render Server, OLE Client, the Running Object Name List MMF, the OLE Running Object Table, and the array.dll. In short, a Render object is placed in an OLE container application, such as Microsoft Word, or the Oclient sample OLE client application provided with SALT. The Render object registers itself in the OLE Running Object Table (ROT), then supplies a name using the name of the circuit and the node which is to be viewed in the Running Object Name List (RONL). As the simulation progresses the kernel, array.dll, queries the RONL for registered names. If any are found, data is sent to the object using that name using standard OLE2 interfaces. The Render server then draws the data into the client application. The data from the simulation is stored through the Render server in the client application.

The same node can have more than one instance of the Render server connected to it. This allows different presentations of the same data, or similar presentations in different Clients.

### OLE Automation

This is a brief description of OLE Automation for those readers who are not familiar with this technology. More detailed descriptions can be found in various Microsoft references. OLE Automation is a technology that allows one OLE application, an OLE Automation Controller, to take advantage of services provided by another OLE application, an OLE Automation Server. The service provided will depend on the server, but is always provided though a standard object interface. The interface is used to expose any capability through a cross-application, object-based interface.

Referring to the pervious figure, the array.dll contains the OLE Automation Controllers. On behalf of IsSpice4, array.dll will request the OLE automation servers, fourier.dll and others, to provide computing services to complete a simulation.

## The SALT Kernel

### The Array DLL

The SALT kernel, array.dll, consists of the Array Node and Array models. The Array node is responsible for calculating the correct array data which is passed from node to node. The array models within the array kernel are responsible for calculating the correct outputs for a given input. This requires that they call the OLE servers providing the calculation services.

The kernel is also responsible for updating all running objects listed in the running object name list. This process is covered in the Render chapter.

### The Array Node

IsSpice4 contains different types of data nodes to accommodate analog, digital, and sampled data. With array processing, we introduce another node type, the array node. An array node can be considered a collection of IsSpice4 vectors like a bus. Array nodes can represent giant data sets, such as movies or sound tracks, or a smaller collection of IsSpice4 analog or digital signals. To accommodate this wide variety of signals, a unique node data structure is used. The contents of this structure are given below:

| | |
|---|---|
| double state: | simulation time |
| int val: | an auxiliary value to force an iteration, increment until stable |
| int model: | used to decide the type cast of the instance |
| void * instance: | a cpp instance pointer to the object that<br>places data in the output node's memory mapped file |

Each node of an array model will have a copy of this structure associated with it. The C++ instance pointer (this) of the model is saved in the array node structure, along with its model type. The actual array data is stored in a MMF. When the kernel requires data to be updated, the array kernel calls the model which is pointed to by the instance pointer of the node. The model is then responsible for updating the data in the MMF.

The naming convention used for the MMFs is given in Appendix B. Any object that knows the naming convention may attach to the MMF.

## Array Models

Five basic array models have been constructed to operate on array node data. These models are;

| | |
|---|---|
| mat_src: | an array source that gets arrays from files |
| mat_snk: | an array sink that saves arrays in files or sound devices |
| array: | performs various matrix operations; convo lution, transforms, etc |
| ato_mat: | converts analog data to arrays |
| mat_toa: | converts arrays to analog data |

The first three models are OLE Automation Controllers, and perform the all of the manipulation and initialization of the array data. The last two models are used to interface the array data to the analog data.

The OLE Automation Controllers depend on two IsSpice4 model parameters to configure the OLE Server and request the correct service. These model parameters are “op” and “argname”. The op parameter is used to construct the OLE ID that will be used to query the registry. If the OLE Server is found, the registered object will run and initialize itself. The argname parameter will determine which service the running object will provide to the OLE Controller.

As an example, let’s take a look at the model for a mat_src;

```
.MODEL mat_src_001  mat_src( op="source"  dim=[4096] period=.2 compress = 0
input=hello.wav  argname="sound_file wavelet daub20  nplay")
```

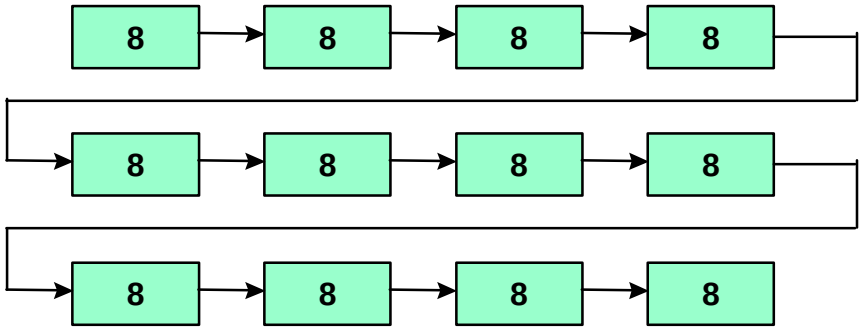
Notice that “op” is set to “source”. This will tell the kernel to query the registry for Array_Source. The response will be to load the server using that ID (in this case, source.dll). The “argname” parameter is parsed by the kernel and the first value is used as the service that is being requested. In this case, “sound_file” requests a service that reads data from a sound (.WAV) file. This wave file is provided as the argument to the “input” parameter. All of the other OLE Automation Controllers function in a similar manner.

The ato_mat and mat_toa models are specially designed models for interfacing the array node data to and from the analog data of IsSpice4. A key feature of the mat_toa and ato_mat models is their data ripping capability. The “loc=” model parameter is used to specify the location of the analog data in the array. See the Bridge.Cir example in the Examples Chapter for more details.

## The Array

SALT arrays are essentially a collection of IsSpice4 Vectors. (See Appendix A for more information about these vectors.) The IsSpice4 vectors in these arrays are almost always uniformly spaced in time or frequency. The vectors are stored in a multidimensional matrix, defined by the array model’s “dim” parameter. The first value in this dimension parameter is always the size of the vector. The second value defines the dimension of a set of vectors. The remaining values define higher order dimensions of the matrix. This is shown below.

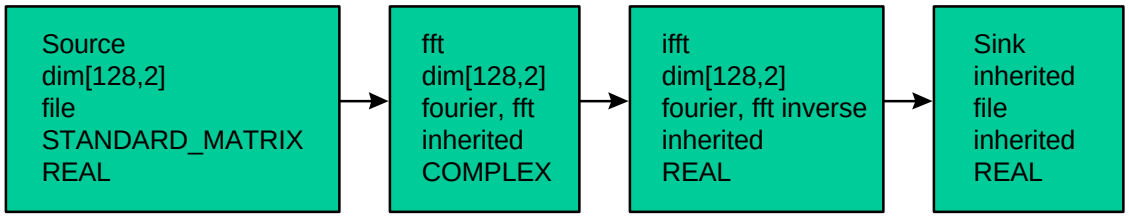
*low memory address*




*high memory address*

Matrix types are inherited from their inputs, therefore, the type of a matrix is established by the original input in the Source or ato_mat model. Two types of matrices exist, STANDARD and SPICE_PLOT. A SPICE_PLOT matrix has the first element of the second dimension filled with the default time vector. A STANDARD matrix does not contain this default time vector.

The following flow is an example of how matrix types are inherited. Data flow is from an input (source) to an output (sink) with a fft and ifft performed on the data. Each block consists of the server name (source), the array dimension (dim[128,2]), the type of operation (file), the matrix type (STANDARD_MATRIX), and the data type (REAL).




Data within the arrays can be REAL (usually time domain) or COMPLEX (usually frequency domain). Frequency domain data is always uniformly spaced, and its independent vector is implied. The independent time vector in a SPICE_PLOT is not transformed using any Fourier or Wavelet transforms; it's always a time series. Data compression is not used to reduce the heap storage, but instead, compression is used to test algorithms and reduce file storage size. This allows compression and decompression in-place; the array size does not need to be changed.

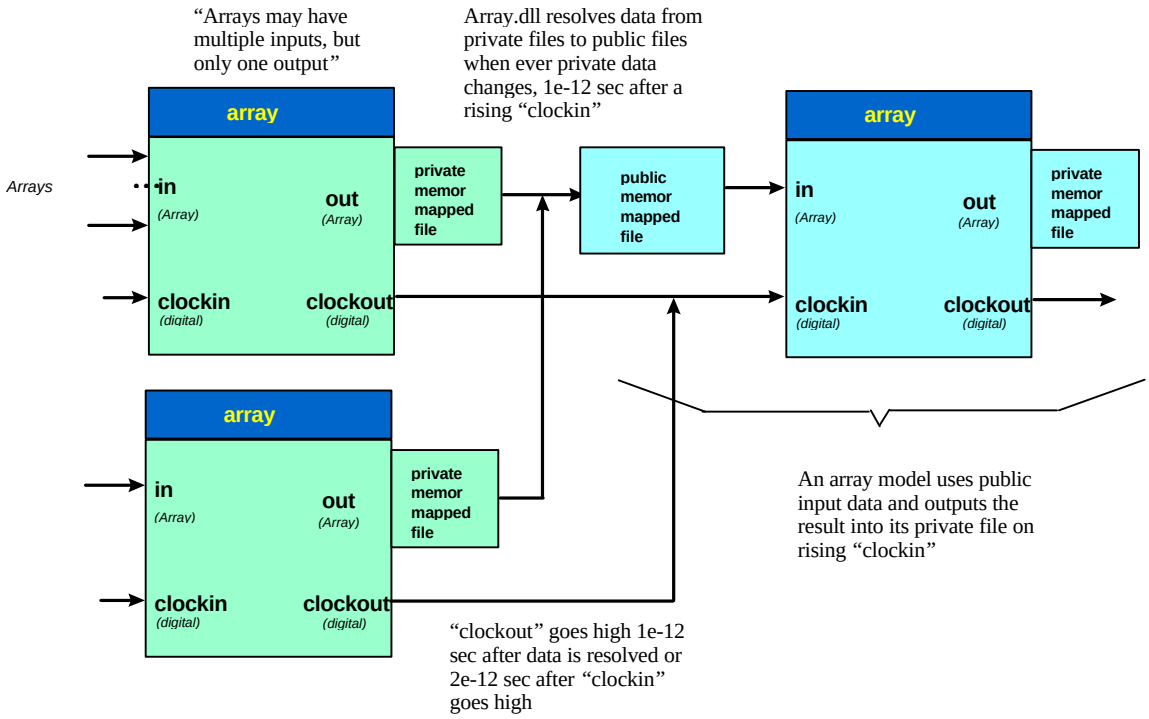
It turns out that the memory storage requirements for complex and real data are identical when transformed using the FFT. Complex data has 1/2 as many complex data points as real data, but has the same number of doubles because each component, real and imaginary use double storage; which allows the arrays to be identical in size. Storage of complex data is in successive pairs of real and imaginary points, so the real data pointer can be cast to a complex type after the transformation from time to frequency.

## Calculating A Node Value

The XSPICE event scheduler queues events for "C" code models whenever the state of an event driven node (digital, array, etc.) changes at the time delay requested by an API function call. The event queue will iterate the calls to objects that continue to change as a result of their evaluation. This iterative feature is used in the initialization of array models at time equal to 0. The order in which objects are evaluated can't be predicted, so an iterated solution is necessary to propagate an initial condition from the input to output of all the arrays. The general implementation is to

initialize each element in the IsSpice4 vectors to the initial input value. *(Another possibility you might try for your servers is to get the initial condition from a file which contains the value of the last simulation result.)* The iteration is forced to continue by incrementing the “val” member of the output state as defined in the array node data structure.

Each array output consists of two files, one called “public” and the other called “private”. The public parts of an array are accessible to the input ports of the array models to which they are connected. The private parts are used to hold the computations required by the array model. Computations take place under control of the "event queue" which is part of the IsSpice4 simulator. An event is processed by all array models connected to a node. The private parts are resolved or copied into the public file for that node when a solution is found. This generates an event for all models whose input ports connect to that node. This scheme is shown in the following figure.




When 2 array outputs are connected together, a process called resolution takes place. Resolution can be either by summation, replacement, or a combination of the two. The resolution method for the baseline array models is hard coded into the models, and is the basis for stripping out certain array channels for analog processing and their later re-insertion into the array data structure.

Before array data can be processed, it is necessary to fill the array with data from one of the following models

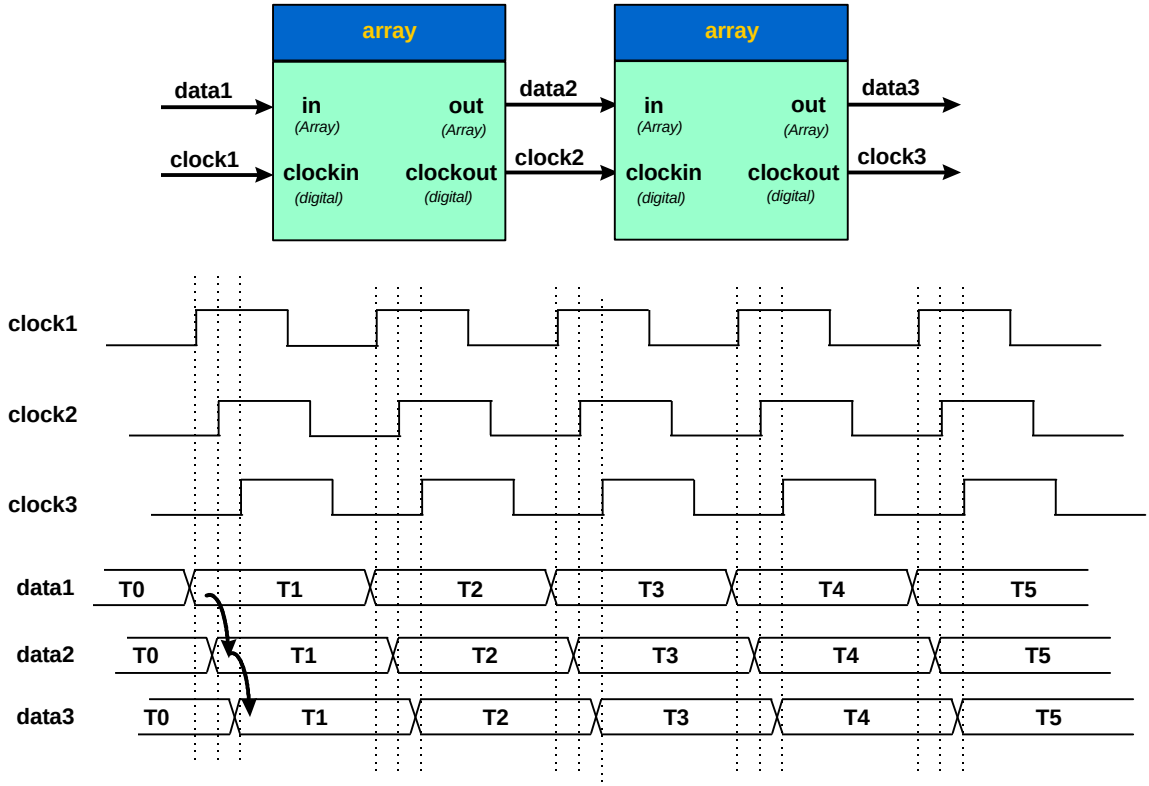
mat_src,
array ,
ato_mat .

These models all have clock outputs to signal completion of data processing for an array period. The “clockout” signals connect to the “clockin” nodes of mat_snk or array models. The “clockin” signal kicks off an event which signals the model to perform its respective computational function. The mat_toa model works slightly differently; it generates analog signals based on the data content of the array. The IsSpice4 time step will be reduced, if required, in order to preserve the “steptol” accuracy specified by the mat_toa model.

You can wire array outputs together and clockout signals together. When the combined clock goes high, all models connected together in this manner will have completed their computation, and the resulting data will have been transferred to the public data file for that node.

When a rising “clockin” is detected, the appropriate server function is executed and its output state is changed. The change in output state signals the event simulator to resolve array data from the instance private file to the node public file. Upon completion of this operation within array.dll, any running objects connected to this node are updated.

If you cascade “clockout” to “clockin”, then the second array will complete its computation just after the first. The simulation delay is set to 1e-12 seconds, and the clockout delay is 2e-12 seconds. If, instead, all clock inputs are connected to a single clock source, the computations will march down a pipeline like a synchronous shift register, each delayed by the clock period. Clock periods should generally be set by mat_src models. These clocks will drop low 1/2 period after going high.




# SALT Models

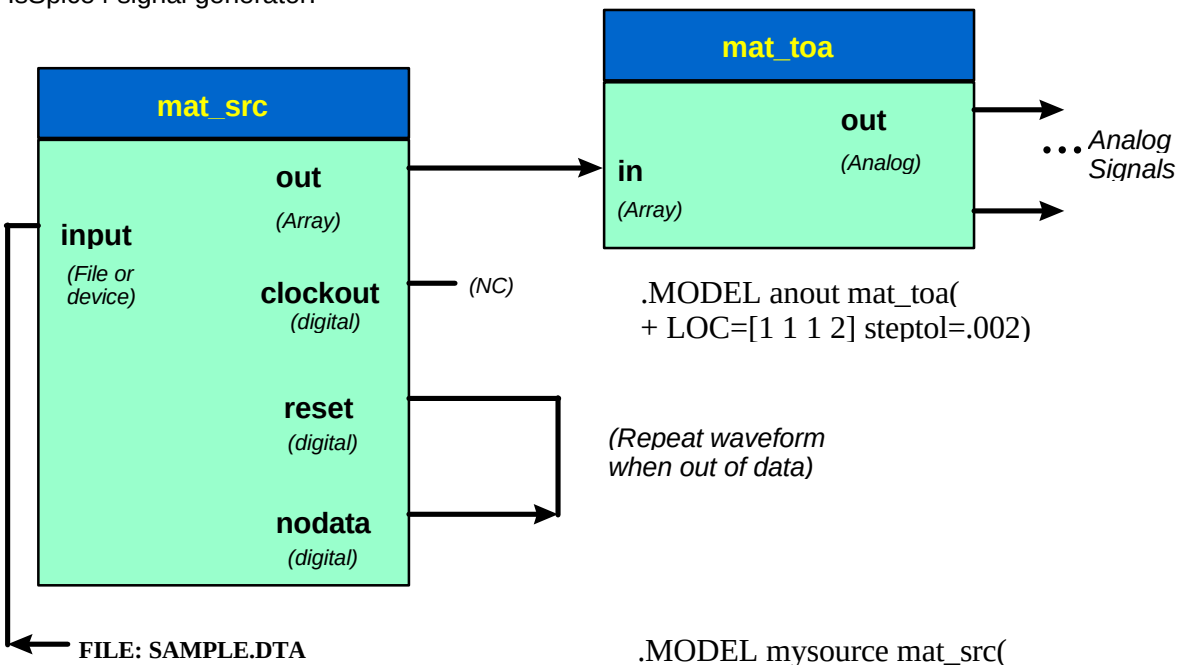
## mat_src

**Example:**

```
.MODEL mat_src_001  mat_src( op="source" ; use the  Array_sound server
+ argname="sound_file  wavelet daub20  nplay" dim=[4096] ;  max array size
+ period=.2 compress = 0 ; percentage
+ input=hello.wav)
```

“Mat_src”, source servers, take input from files, devices or other programs. The output of a source is a single array, with a digital “clockout” signal that goes high when a period of data has been read into its array. Another digital output, “nodata”, signals that there is no more data to

read, even though the simulation is not complete. The source can be made to start over by connecting its “nodata” output to its “reset” input. Thus the source will repeat the output. The “clockout” high state has resistive strength so that array output nodes and their associated clocks can be connected together. The file data is read in again, each time the model is reset. This allows the data stored in the file to interact dynamically with a simulation. (i.e. The simulation modifies the data in the file.) The following figure illustrates the use of a mat_src server as a IsSpice4 signal generator.




.MODEL anout mat_toa(
+ LOC=[1 1 1 2] steptol=.002)

**FILE: SAMPLE.DTA**
SCALE 1
DATA REAL
varstep
0 0.670238 1.22239
1e-009 0.679271 1.20273
2e-009 0.688473 1.10607
3e-009 0.693092 0.967208
…
9.9e-008 0.670085 1.23446
END

.MODEL mysource mat_src(
OP="SOURCE"
+ ARGNAME="MATRIX_FILE "
+ DIM=[128 3]
+ PERIOD=99N
+ COMPRESS = 0
+ INPUT=SAMPLE.DTA )

The server for mat_src is source.dll. The server is accessed by setting the op model parameter to `source`. (op=”`source`” ) At this time this is the only server available for the mat_src model.

The remaining parameters for the mat_src model depend on the function that the server is to provide. The function is specified as the first argument in the `argname` argument list. The value assigned to `argname` is a list of parameters enclosed in quotes. The following functions are available:

| Function Name | Description |
|---|---|
| sound_file | read WAV files |
| matrix_file | read external IsSpice4 piece-wise linear data point files |
| random | output random values |
| atod | hardware interface for A/D converter |

The following sections describe each of these functions and the parameters that are allowed for each. Parameters that are not followed by an equal sign, "=", are expected to appear on the argname argument list. All other parameters appear independently in the mat_src model statement. In all cases the parameter appears first, followed by an explanation. An example is provided for each function type.

**sound_file:**
The current implementation takes data from Microsoft WAV files with the following options:

| | |
|---|---|
| wavelet | Performs wavelet compression. This parameter requires a mother function to be specified. The percentage compression is given by the compress parameter. |
| daub20 \| daub12 \| daub4 | The wavelet mother function. |
| play | Plays the input sound file. The sound file is played before any compression is performed. |
| dim = | The dimension of the array that stores the data. |
| period = | The time that one array represents |
| compress = | if 0, no compression takes place, otherwise, the value entered is the percent of the wavelet data that is saved. Compression is by amplitude. The wave table index is saved at the end of the array data. Data amplitudes greater than the compression percentage are saved and their position is determined from their index, up to the number calculated from the compression percentage. All other data is assumed to be zero. |
| input = | The filename of the WAV file to play. |

Examples:
This model represents a mat_src model that will read and play the file hello.wav into an array that has a size of 4096. The array will represent .2 seconds of the sound contained in hello.wav.

```
.MODEL modelname mat_src( op="source"  argname="sound_file play"
+ dim=[4096] period=.2  input=hello.wav)
```

This is the same as the previous example except that wavelet compression is imposed on the data read using the daub20 mother function with a 25% compression. The sound from hello.wav is not played in this example.

```
.MODEL modelname mat_src( op="source"  argname="sound_file wavelet daub20"
+ dim=[4096] period=.2 compress=25   input=hello.wav)
```

**matrix_file:**
Takes data from a file. The IsSpice4 format is given in the Data File Syntax section. The following model parameters are used:

| | |
|---|---|
| dim = | The dimension of the array that stores the data. |
| period = | The time that one array represents |
| input = | The filename of the IsSpice4 PWL file to read. |

Example
This model represents a mat_src model that will read the IsSpice4 PWL file, Data.Dta into an array. The array represents 3 time vectors each 128 elements in length. (See the Array section,

or the Bridge.Cir example.) Each of the time vectors will represent 99ns of data from the PWL file.

```
.MODEL modelname mat_src( op="source"  argname="MATRIX_FILE"  dim=[128 3]
+ period=99N  input=Data.DTA)
```

**random:**

Generates a normally distributed "Gaussian" signal with the following options:

| | |
|---|---|
| `val` | The rms noise value. |
| `offset` | The DC offset of the noise. |
| `seed` | The noise seed value. |
| `dim =` | The dimension of the array that stores the data. |
| `period =` | The time that one array represents |

A new value is calculated for each ( period / dim[0] ) time step. (Dim[0] represents the first dimension, time vector, value of the dim parameter.) The seed is used to make data repeatable (on the same platform) and allow more than one generator with a different sequence. All values are IsSpice4 floating point numbers.

Example

This model represents a mat_src model that generates a random signal with an rms value of 2 and an offset of .75 using a seed of 167. The signal represents 99ns for each array of data.

```
.MODEL modelname mat_arc( op="source"  argname="random val=2 offset=.75
+ seed=167.00"  dim=[128] period=99n)
```

**atod:**

The current implementation interfaces the National Instruments AT-2150 A-to-D converter. The following parameters are used:

| | |
|---|---|
| `national` | The manufacturer of the A/D converter. |
| `dim =` | The dimension of the array that stores the data. |
| `period =` | The time that one array represents |
| `input =` | The configuration file for the A/D board |

An auxiliary configuration file defines the parameters for the National Instruments device as follows:

BOARD=AT-2150; the board number; this is the only board supported at this time.
GROUP=1
DEVICE=1
RATE=10K
CHANNEL=0,1,2,3
ITERATIONS=0
GAIN=.9E-4
OFFSET=0
END ; file won't be searched past this point.

These parameters are used to fill in the NIDAQ API function arguments. Please refer to your National Instruments User manuals to see how data in this file controls the board. The code that drives the board is provided in adc.cpp in the Source server subdirectory.

Example

This model represents a mat_src model that reads data from a National Instruments AT-2150 A/D board. Fills the array with data representing 99ns. The configuration file, NIATOD.CFG is used to configure the software for the AT-2150.

```
.MODEL modelname mat_src( op="source"  argname="ATOD NATIONAL"  dim=[128 3]
```

```
+ period=99n  input=NIATOD.CFG )
```

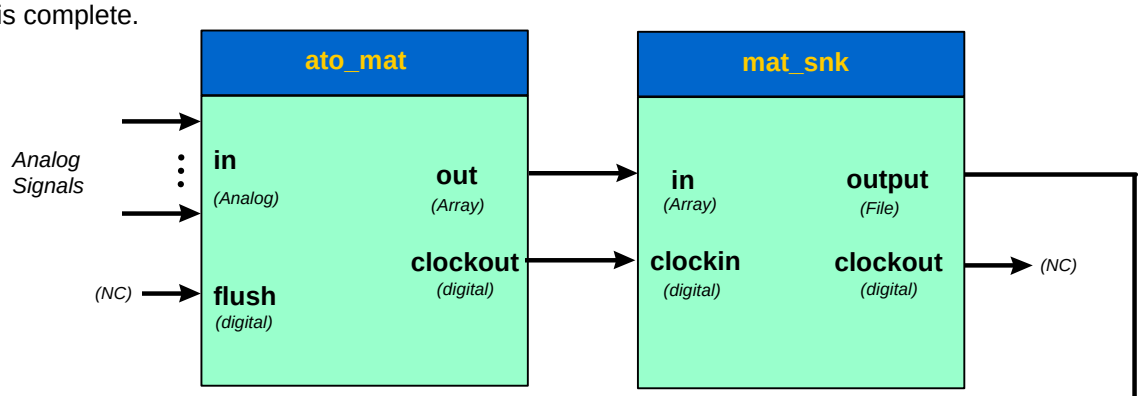
Intusoft will update and expand the capabilities of source.dll so you should create a different server if you wish to add functionality to the mat_src model. (Do not add new argument keys to source.cpp.)

## mat_snk

**Example**

```
.MODEL mat_snk_001  mat_snk( op="sink" ; use the  Array_sink server
+ argname="sound_device  wavelet daub20  play" output=hello2.wav  )
```

"mat_snk" servers are used to output array data to files, devices, IsSpice4 vectors or other programs. The sink will transfer data from its input on the rising edge of its "clockin" signal. A "clockout" signal is provided so that a mat_src model can be reset when a series of computations is complete.




```
.MODEL getmat ato_mat(
+ dim=[100 2] loc=[1 0 1 1]
+ period = 99n)
```

```
.MODEL asink mat_snk(op="sink"
+                argname="matrix_file delay=99n"
+                output=sample.dta  )
```

**FILE: SAMPLE.DTA**

```
Project: ARRAY_B
Instance: a2
SCALE 1
DATA REAL
varstep
0 0.670238  1.22239
1e-009 0.679271  1.20273
2e-009 0.688473  1.10607
3e-009 0.693092  0.967208
…
9.9e-008 0.670085 1.23446
END
```

The server for mat_snk is sink.dll. The server is accessed by setting the op model parameter to `sink`. (op="`source`" ) At this time this is the only server available for the mat_snk model.

The remaining parameters for the mat_snk model depend on the function that the server is to provide. The function is specified as the first argument in the `argname` argument list. The value assigned to `argname` is a list of parameters enclosed in quotes. The following functions are available:

| Function Name | Description |
|---|---|
| sound_file | save in a WAV file |
| sound_device | speaker output |
| matrix_file | save in IsSpice4 piece-wise linear data point files |
| vector | save array data in IsSpice4 vectors |
| dac | hardware interface for D/A converters |

**sound_file**
The current implementation saves data in Microsoft WAV files with the following options:

| | |
|---|---|
| play | plays the input sound file, after de-compression. Note: nplay is undefined, and is used as a simple method to turn play off; moreover, there is no provision to handle cases of multiple accesses to the WAV playing driver, so if your simulation is running quickly, you won't be able to hear both input and output sounds. |
| output = | names the file in which the sound will be stored. All sound is stored in WAV format. |

Example
This model statement plays the sound being processed by the simulation and also writes it to Hello2.Wav.

```
.MODEL modelname mat_snk( op="sink" argname="SOUND_FILE play"
+ output=Hello2.Wav )
```

**sound_device**
The current implementation plays a Microsoft WAV file to the sound hardware installed on your computer.

Example
This model statement plays the sound being processed by the simulation.

```
.MODEL modelname mat_snk( op="sink" argname="SOUND_DEVICE")
```

**matrix_file:**
Saves the matrix in an IsSpice4 PWL file. The following parameters are defined for a matrix_file operation.

| | |
|---|---|
| delay | Causes the output time series to be offset by the value specified. |
| tol | Interpolation tolerance. |
| output = | Names the file to which the PWL data will be stored. The format is discussed in the Data File Syntax Section. |

The delay entry is used to compensate for the period delay built into the array models. It eliminates the first "dead" period of output. The tol entry assures that the data can be reconstructed using linear interpolation to the absolute tolerance specified.

Example
This model statement produces an IsSpice4 PWL file with an absolute interpolation tolerance of 10u delayed by 99ns. The PWL file is written to Data.Dta.

```
.MODEL modelname mat_snk( op="sink" argname="MATRIX_FILE delay=99N
+ tol=10U" output=Data.Dta)
```

**vector:**
Saves the array data in a IsSpice4 vector. The following parameters are used:

| | |
|---|---|
| output = | names the file in which the vector data will be stored. See the Data File Syntax section for a description of the file format. |

Vectors are named using the following convention:

| **Array type** | **example** |
|---|---|
| Real | |
| independent variable “instance name”_time; | a2_time |
| vectors “instance_name_“vectorNumber” | a2_3 |
| Complex | |
| independent variable “instance name”_freq; | a3_freq |
| real part | |
| vectors “instance_name_re_“vectorNumber” | a3_re_4 |
| imaginary part | |
| vectors “instance_name_im_“vectorNumber” | a3_im_4 |

The output real-time vectors should be plotted using IntuScope using the IsSpice WFMs… function as x-y plots. Use the independent vector provided for the array because the default simulation vector will have a different set of time points.

Example
This statement generates real-time vector output.

```
.MODEL modelname mat_snk( op="sink" argname="VECTOR")
```

The statement generates real-time vector output and also writes the data to temp.Dta.

```
.MODEL modelname mat_snk( op="sink" output=Temp.Dta argname="VECTOR")
```

**dac:**
The current implementation interfaces the National Instruments AT-AO-6/10 Analog output board. The following parameters are used:

| | |
|---|---|
| `initfile` | ? |
| `output =` | Names the configuration file used to configure the softwar e interface. |

```
.MODEL HARDSNK MAT_SNK(OP="SINK"
+      ARGNAME="DAC NATIONAL INITFILE=AFILE.DTA"
+      OUTPUT=NIDAC2.CFG  )
```

An auxiliary configuration file defines the parameters for the National Instruments device as follows:

```
BOARD=AT-AO-6/10
GROUP=1
DEVICE=1
RATE=10k
CHANNEL=1,2,3
ITERATIONS=0
GAIN=200
OFFSET=0
END ;  anything below hear is ignored
```

These parameters are used to fill in the NIDAQ API function arguments. Please refer to your National Instruments User manuals to see how data in this file controls the board. The code that drives the board is provided in dac.cpp in the Sink server subdirectory.

Intusoft will update and expand the capabilities of sink.dll so that you may select a different name for any mat_snk server you wish to add. Do not add new argument keys to sink.cpp.

## array

**Example**

```
.MODEL array1 array( dim=[20 5 6]  op="Copy" )
.MODEL array2 array(  dim=[128 1]  op="FOURIER"
+ argname="fft  freqToTime")
```

There are currently three servers available for array; acopy.dll, matrix.dll and fourier.dll. These servers are accessed by setting the op model parameter to desired server name. (op=″copy″, op=″matrix″, op=″fourier″)

The remaining parameters for the array model depend on the function that the selected server is to provide. The function is specified as the first argument in the argname argument list. The value assigned to argname is a list of parameters enclosed in quotes. The Copy server copies the input to its output. It is primarily used as a template to make custom array servers, however, it can be used as a delay or ideal transmission line. The Fourier server has several functions available based on the Fast Fourier Transform (FFT). The Matrix server contains several matrix math functions. These functions are listed below.

| Function Name | Description |
|---|---|
| Fourier | Server Name |
| fft | Performs a complex transform in the “forward” direction. |
| fft inverse | Performs the inverse complex FFT. |
| fft timetofreq | Transforms real, time domain data, to the frequency domain. |
| fft freqtotime | Transforms complex, frequency domain data to the time domain. |
| Convolution | Performs a convolution filter based on an external parameter file. |
| Matrix | Server Name |
| math | Performs matrix math |

All of the Fourier and Matrix server functions require the dim model parameter.

dim = The dimension of the array that stores the data.

The first element of the dim model parameter determines the FFT size, while its scaling is determined by the period parameter. The first element of the dim model parameter must be a binary radix. (an integer power of 2)

**fft**

Performs a complex FFT.

```
.MODEL modelname array( dim=[128 1]  op="FOURIER"  argname="fft " )
```

**fft inverse**

Performs an inverse complex FFT.

```
.MODEL modelname array( dim=[128 1]  op="FOURIER"  argname="fft inverse" )
```

**fft timetofreq**

Produces the complex FFT for the time series input.

```
.MODEL modelname array( dim=[128 1]  op="FOURIER"
+ argname="fft  timetofreq")
```

**fft freqtotime**
Produces the time series for the complex FFT provided as input.

```
.MODEL modelname array( dim=[128 1]  op="FOURIER"
+ argname="fft  freqtotime")
```

**Convolution**
The following model parameter is expected:

`paramfile =` The filename containing the coeficients of convolution.

The input data is REAL, and will be converted from time to frequency, multiplied by the frequency domain data from the `paramfile` , and the result will be converted back to time data. This is essentially a FIR filter with coefficients defined by the first element of the dim model parameter. The `paramfile` can be either time (REAL) or frequency (COMPLEX) data. The `paramfile` describes the impulse response of a filter, either as a time series or a COMPLEX frequency response. The filter response is taken over the "period" parameter. If the CIRCULAR keyword is not present, a frequency domain response will be constructed from an impulse response that is 2 times the period in length, with the second time period having all zeros. If, on the other hand, the CIRCULAR keyword appears in the paramfile, the paramfile data will be used to construct the time series for the second period. For non CIRCULAR convolution, it will be necessary for the model to convert frequency domain paramfile data to time in order to null the second period. CIRCULAR convolution will produce valid results if the time series is periodic about "period". The non-CIRCULAR filter produces valid results if the impulse response is zero for the second period. The paramfile is read in for each simulation so that it can be modified by the simulation to test adaptive algorithms using an ICL script.

```
.MODEL modelname array(dim=[128 3] op="FOURIER" argname="convolution"
+ paramfile="Test1.Fir")
```

**math**
The following parameters are used to perform matrix math functions:

| | |
|---|---|
| `mul` | performs multiplication |
| `div` | performs division |
| `sub` | performs addition |
| `add` | performs addition |
| `conj` | performs conjugation |

The parameters are placed on the `argname` line in the order that the arithmetic is to be performed. For example, `argname="math  mul  conj"` will multiply the inputs then take the conjugate of the answer.

The transform of frequency to time and its inverse can be handled more efficiently for special cases, due to symmetry; that is, time has no imaginary component and frequency is symmetric about the Nyquist frequency. The first frequency cell holds the DC or average value in its real part and the Nyquist component in its imaginary part, both quantities in the first cell are real, their imaginary parts are zero.

## ato_mat

```
.MODEL ato_mat_001  ato_mat( dim=[20 5 6]  loc=[1 3 2 5 3  3] period = 1)
```

The dim parameter is used to specify the dimensions of the array used to store data. The loc parameter specifies the location within the array where the data is to be placed. (See The Array section for details about the array storage.)

## mat_toa

```
.MODEL mat_toa_001  mat_toa( dim=[20 5 6]  loc=[1 3 2 5 3  3] )
```

The dim parameter is used for error checking since the actual dimension is inherited from the device to which the mat_toa is connected. The loc parameter specifies which section of the input array will be ripped into analog signals. (See The Array section for details about the array storage.)

## Data File Syntax

The Data File is created using the Sink OLE Automation Server using the keyword `matrix_file` on the `argname` parameter. The file is a standard ASCII text file with the structure shown below. If necessary you can create or edit these files with a standard ASCII text editor.

Header:
- Line 1: The source project if the data is generated using a SALT sink.
- Line 2: The instance of the sink that generated the data

Key Items:
- SCALE [multiplier | AUTO]
- [MATRIX_TYPE=] [STANDARD] [SPICE_MATRIX] makes the default data part of the array
- DATA [CIRCULAR] [REAL | COMPLEX]
- [varstep | fixedstep]
- < …. data items …. >
- END

The Header lines are optional, anything can be placed before the key words; however, once the first key word is entered, the remaining data must be in the order shown. You can also place anything after the END keyword. Placing additional text after the END keyword is a convenient way to save several variations of data. In fact, this technique can be used either before the SCALE key word or after the END keyword.

Data is saved in a row column format. Each row represent the values of the matrix items for the value of the independent variable given in the first column. SALT will interpolate the data as required to compute values needed at other times or frequencies, using linear interpolation. All items for a dimension are entered in a row. Each additional dimension is given another row. The following table shows a piece-wise linear data point file.

```
Project: Data
Instance: a56:x2
SCALE 1
MATRIX_TYPE=STANDARD
DATA REAL
varstep
0 1.45871  0.659409
7.80315e-010 1.4505  0.667121
1.56063e-009 1.39972  0.67796
2.34094e-009 1.31225  0.685685
…
END
```

Each line of the file contains all of the PWL data for a single time or frequency point. The first entry is the time or frequency default if the “varstep” keyword is present, otherwise if “fixedstep” the time or frequency is determined from the period parameter and size of the first dimension. “Varstep” data is considered to be joined by a straight lines. The matrix is filled using linear interpolation of the default data. If MATRIX_TYPE = SPICE_MATRIX, the default time or

frequency data will be read into the first vector of the array data matrix. Subsequent data fills the dimensioned arrays in the order read. (See The Array section for more information.) AUTO scaling causes the independent variable to be scaled using the "period" parameter, otherwise, it's multiplied by the SCALE value. For example, SCALE 2 will multiply each default scale by 2.0. Auto scale makes it easy to build filters or data sets that can apply to data that has any period. The CIRCULAR key is used for convolution filters to indicate that the replica's second half is not to be zero filled (see the description of the Fourier array model for more details).

The following table shows how to fill in data describing positions of 4 objects in 3 dimensional space (x,y,x) for a time period of 0 to 10 seconds (t).

```
DATA REAL
varstep
0 15 5 30          target 1, t=0, x=15, y=5, z=30
0 5 0 20           target 2, t=0
0 15 0 20          target 3, t=0
0 -30 0 20         target 4, t=0
10 0 5 30          target 1, t=10, x=0, y=5, z=30
10 5 -10 40        target 2, t=10
10 5 -10 40        target 3, t=10
10 30 30 -30       target 4, t=10
END
```

Note: The *italic text* is not part of the SALT data file.

Notice that the position of each target is given at each time. In this case, the targets represent independent points. You could just as easily model them as polygon vertices, making this a description of a 3D object flying through space. This data can be read into an array, dimensioned as dim=[4096,4,4]. The data would then be processed into the array, depending on the simulation time and array clock_in period. The space scaling; feet, meters etc. will be interpreted by the model using this data table. This file is used as part of the Sensor.cir example.

# Render

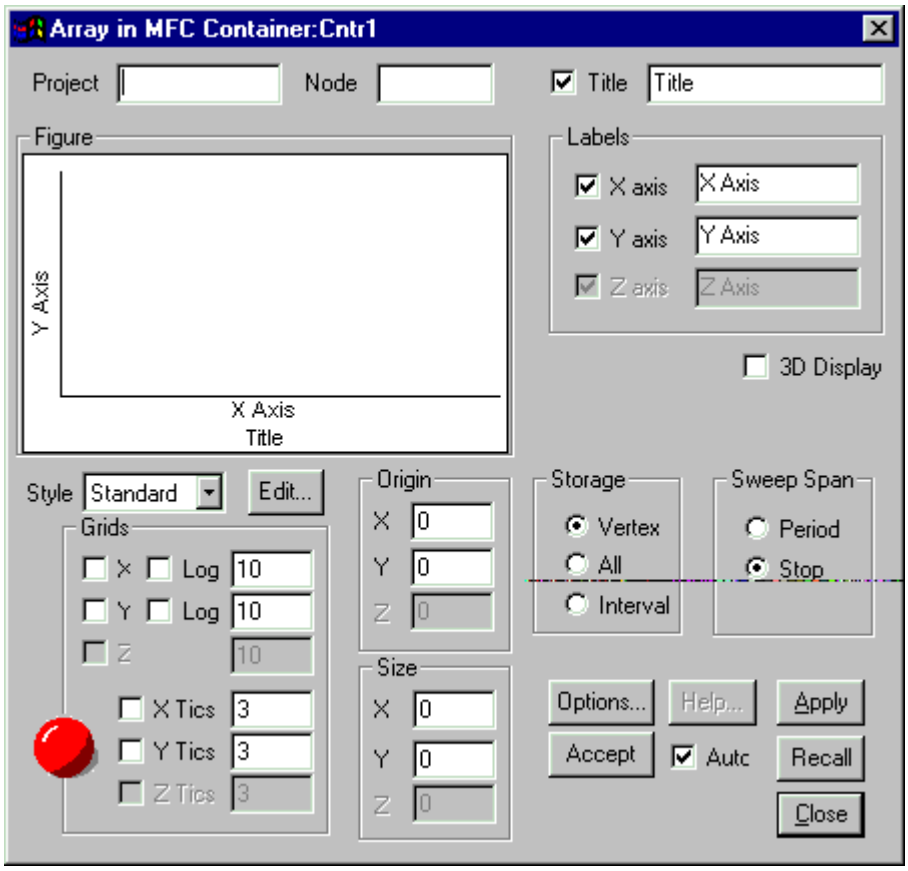
## Overview

Render is a embeddable 32-bit InProcess OLE server. The DLL is registered using the .REG file provided with the SALT package. It is used to view the data for any REAL, time domain, array node.

Render must be started from a client application such as Microsoft Word or the Oclient application supplied with SALT. All client applications will have an Insert Object command that allows playable, embeddable, servers to be placed in the clients document. Check the manuals for your application for the exact menu function and steps.

## The Render Dialog

The following sections will discuss the various portions of the Render dialog. Grayed items will be activated when appropriate. For instance, the Z axis label will be active when the 3D Display option is checked.




**Project**
The circuit filename is entered here. The dialog is case sensitive. Therefore it will be necessary to know how the file is represented in the File manager, or Explorer before entering the name.

**Node**
The array node you want to view.

**Title and Labels**
The Title and Labels of the graph can be entered in these fields. The check boxes determine if the label or title will be displayed on the screen. There is a 128 character limitation on the strings entered into these fields. A practical limitation will be set by the viewing area you select for the graph.

**Figure**
This section is a rough view of the placed image before the Apply button is selected. Any changed made to the dialog will be previewed in this area. This allows you to view the output before committing it to the client site.

**3D Display**
This activates the 3D display mode. This topic is discussed in more detail in the 3D View section.

**Storage**
Data collected by Render is stored in one of three different modes. The intent of these modes is to reduce the amount of data held in memory. Vertex storage saves data proportional to the viewing area for the entire simulation. The larger the placed image the more data that is stored. The All storage mode saves all data in the simulation. The Interval storage mode saves only the data within the X and Y display axis regardless of the simulation parameters. The All storage mode stores the most data since all data points are stored at any given time. The Interval

storage mode has the potential for storing the least amount of data since only those points that fall within the defined X axis will be stored. For the most part, Vertex storage mode will produce the most useful results as far as memory use is concerned.

**Sweep Span**
The options here allow you to alter how data is displayed. Period sweep is generally used for 3D views. It will display a new set of data for each period. The period is determined from the data collection size which is determined by the model parameters of the simulation. When Period is selected for a 2D data view, the plot is erased after each period. The Stop option is generally used for 2D data views and will cause the data to be displayed continuously until the simulation stops.

**Style**
The Style list box, and the Edit button next to it, are used to save 3D View styles. In some cases you may wish to view data from various angles. By saving a view style you can easily recall the view angle. The 3D View section covers this option in more detail.

**Grids**
This section of the dialog displays the type of grids that are available.

**Log Axes**
When Log is selected the edit field for the selected grid becomes the number of cycles.

**Origin**
The numbers entered here specify the origins for the x, y, and z axes.

**Size**
The numbers entered here determine the maximum x, y, or z scale. The maximum is the origin plus the size.

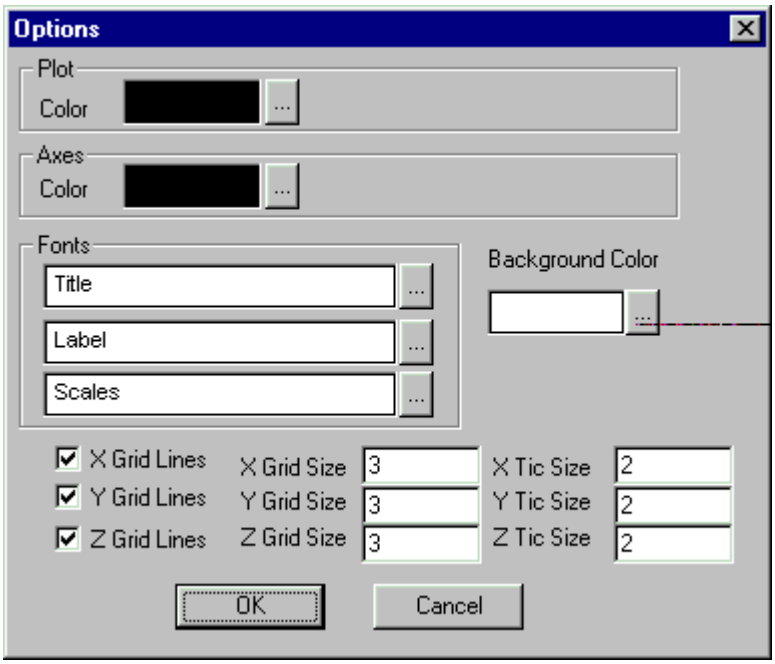
**Auto**
The auto check box is used to initialize the dialog from the MMF supplied by the simulation. It is recommended that this option be left on (checked) until a simulation is run. After the dialog has been initialized with the simulation data the option can be turned off. Once off, the settings of the dialog can only be changed manualy.

**Accept**
This button will force a redraw of the previewed image. It has no connection to the image placed in the client. If you change an option and wish to see its effects, click the Accept button. The figure area of the dialog will be updated with the latest changes.

**Apply**
Click this button to apply any changes to the client image. Once this button is clicked the changes cannot be undone with the Recall button.

**Recall**
This button functions as an undo. It will only undo the actions up to the last Apply.

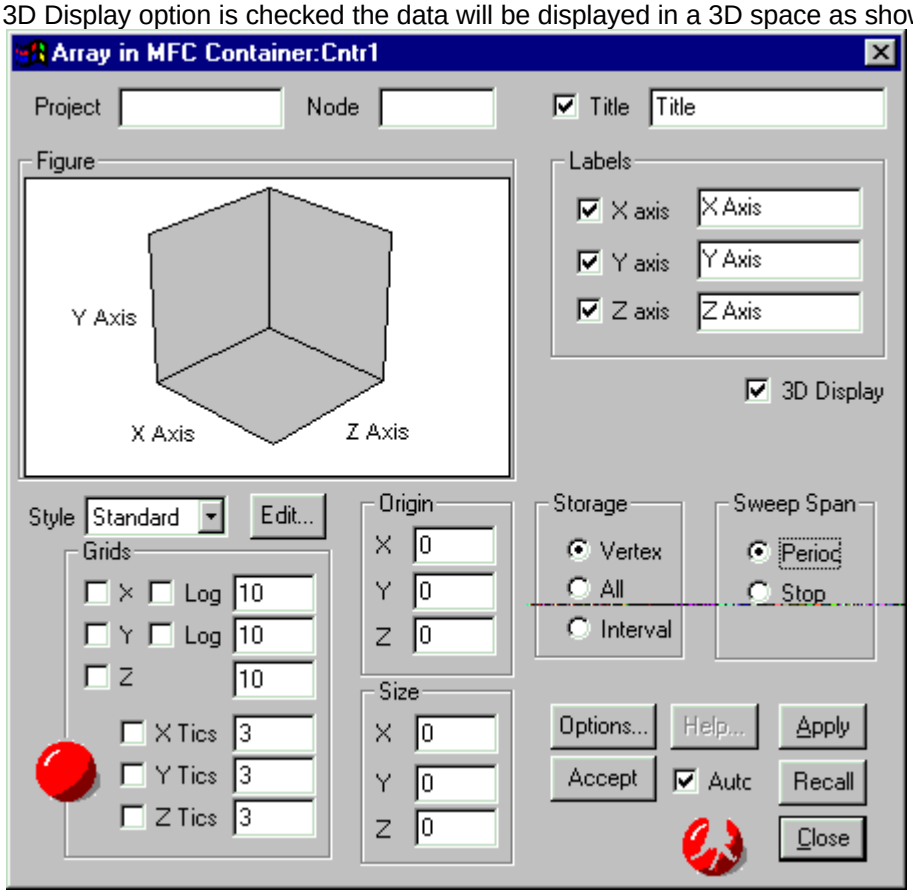
**Options…**
This button activates the Options dialog.




This dialog controls the appearance of the client image.

## 3D View

When the 3D Display option is checked the data will be displayed in a 3D space as shown below.



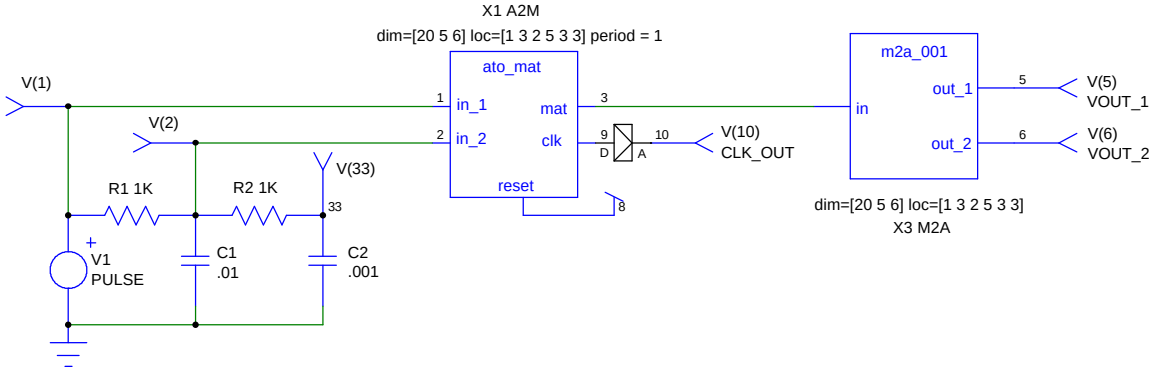
The 3D view orientation is controlled by placing the mouse in the figure area, holding the left mouse button down, and dragging the mouse. This will cause the 3D view box to rotate about the origin, the back corner where the axes meet. If the Shift key is held down while dragging, the image will translate in the direction the mouse is moved. If the Control key is pressed at the same time the Shift key is pressed the 3D view box will enlarge or shrink. If the Control key is pressed by itself the perspective will change allowing you to zoom into the 3D view box. The following table can describes the key combinations and the effect they produce.

| | |
|---|---|
| Shift: | Translate in x and y. |
| Ctrl: | left-right movement to scale Z axis; up-down to fly in or out of the image, changing your observational position. |
| Ctrl + Shift: | Scale x and y. |

# Examples

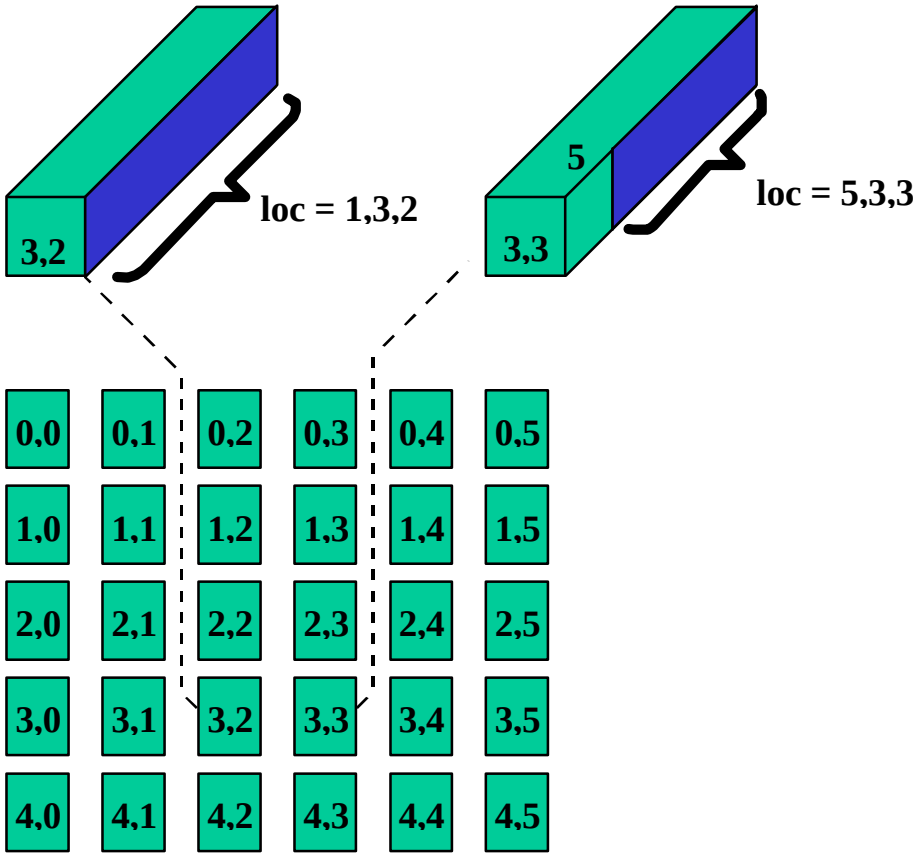
## Bridge.Cir

The Analog-to-Matrix and Matrix-to-Analog node bridges are an integral part to the operation of SALT. This example will use a simple circuit that takes an analog signal, converts it to a Matrix element, then converts the matrix element back to an analog signal.




```
.MODEL mat_toa_001  mat_toa( dim=[20 5 6]  loc=[1 3 2 5 3  3])
.MODEL ato_mat_001  ato_mat( dim=[20 5 6]  loc=[1 3 2 5 3  3] period = 1)
```

The Array-to-Matrix bridge, X1, uses a 20x5x6 matrix as the array node. This matrix is established by the `dim` model parameter shown in the ato_mat model. The matrix is shown in the following figure.



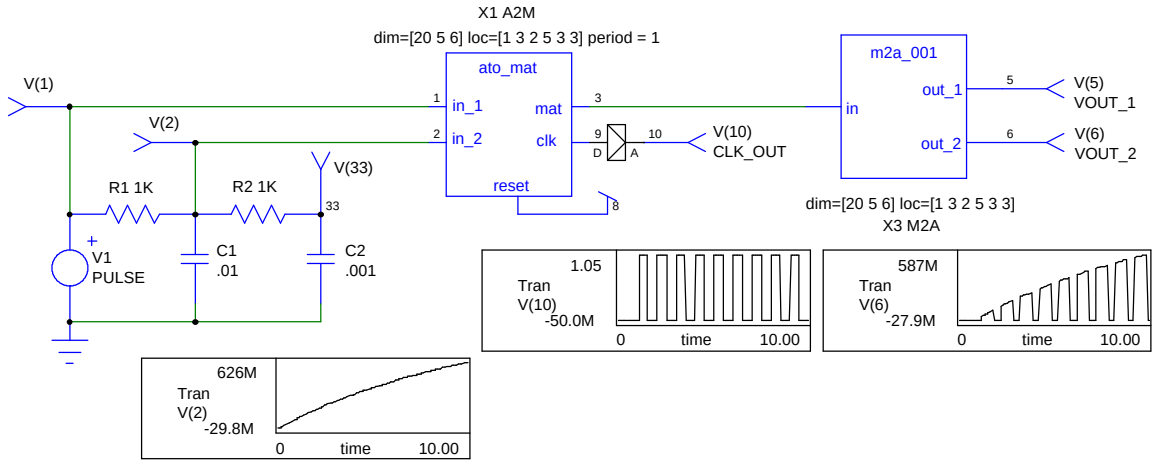

The first dimension of the `dim` parameter, 20, establishes the time vector size. This is the depth of each element shown in the figure. The following two dimensions provide the number of rows and columns in the array node matrix. In all, we will have 30 time vectors each having a size of 20.

Also shown in the figure is the meaning of the `loc` model parameter. The `loc` parameter is the position in the array matrix where the analog data will be inserted. Here we have a two input Analog-to-Matrix bridge that has a dimension of 3 hence we will require 6 values to describe the location for the two analog signals (loc = [1 3 2 5 3 3]). The first analog signal is inserted into the 1st location of the time vector located at the row,column index of 3,2. The second analog signal is inserted in the 5th location of the time vector located at the row, column index 3,3. One thing to be aware of is that the indexing is arbitrary. In that we mean the index can be row,column, or column,row. The choice is up to you. The only gotcha is that you must be consistent throughout the use of the array as each element uses it. The examples provided will use a row,column index.

The final parameter to be aware of is the `period` parameter. This parameter describes the meaning of the time vector. In other words, each 20 element time vector will represent onr `period` of data. In this example each time vector will hold 1 second worth of data as defined in the ato_mat model statement.

During the initial operating point of the circuit each Array element propagates the operating point voltage through the circuit. Once the simulation begins each periods worth of data is propagated through the circuit in response to a CLK signal from the Analog-to-Matrix node bridge. The clock goes high when a periods worth of data has been processed by the bridge. This array data is then passed to all array models attached to the node. The recipient of the data will know there is new data when its clk_in goes high. For the case of the Matrix-to-Analog bridge we process any data present in the array from each time point. For the first period there is no array data. For this

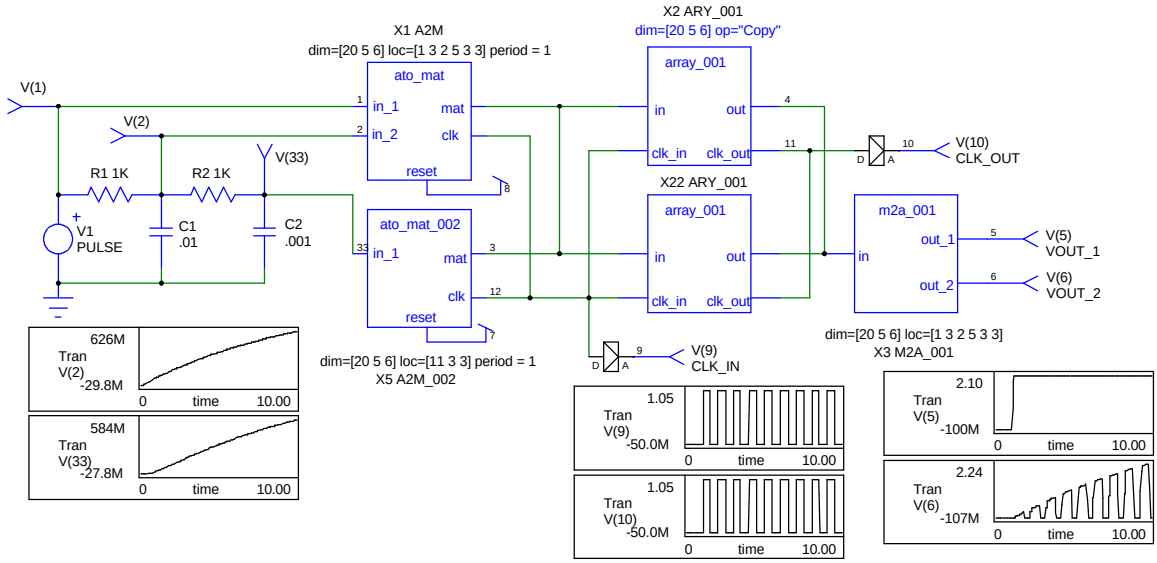
case the array at the input of the Matrix-to-Array bridge will initialize to the operating point of the circuit. Hence, from the figure provided we can see that the output will be a period sampled version of the input delayed by one period.




Each new period will generate new data that is placed in the array matrix established for the circuit. There are no new array matrices created during the simulation. Each set of data replaces the previous.

## Copy.Cir

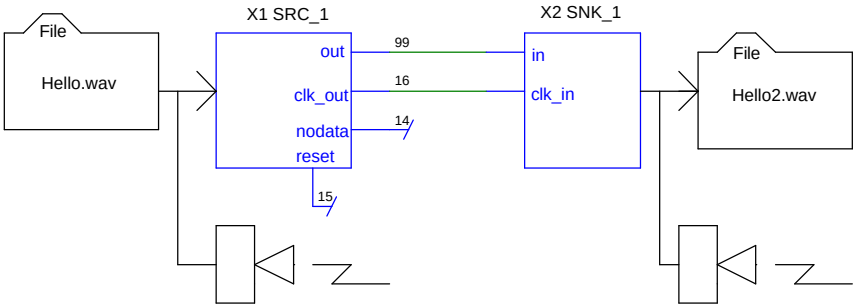
The Array Copy function copies the input to the output whenever the input clock signal goes high. It acts as a one period delay. In this example we connect two Copy models in parallel to demonstrate how array node connections are resolved. As shown in the figure below we have a connection, node 4, where two array nodes connect. At this node the Array node resolve function will determine the correct array value to pass to the next array model.




This circuit will look very similar to the previous example. The only difference is the resolution that takes place at node 4. To resolve the parallel connection the two arrays are summed to create the output that is passed to the next array model.

## Hello.Cir

This example introduces the Source and Sink Array models. These Array models provide a way to input and output the Array matrix values. These matrix values can be sent to or taken from files, hardware devices, or other programs. A simple example is shown below.




This circuit is comprised of two Array code models, a source (SRC_1) and a sink (SNK_1). The Source Array model reads input from a Microsoft .WAV file. The information from this .WAV file is sent directly to a Sink Array model that plays the .WAV file read by the source.

The Source Array model is described by the following model parameters

```
.MODEL mat_src_001  mat_src( op="source" ; use the  Array_sound server
+      argname="sound_file  wavelet daub20   nplay"
+      dim=[4096] ;  max array size
+      period=.2
+      compress = 0 ; percentage
+      input=hello.wav  )
```

The op parameter determines the OLE Automation Server that will provide the function of the Array model. The `argname` parameter describes the command line that configures the server. In this case the Source server. The `sound_file` entry instructs the Source server that the input will be from a sound file (.WAV) file. The `wavelet` entry instructs the Source server to use wavelet compression on the input using the `daub20` mother function, also on this line, with a compression percentage given by the `compress` parameter. The `nplay` parameter is a placeholder used to disable playing the sound. In order to play the sound file replace `nplay` the `play` key word. The `dim` parameter determines the dimension of the array used to store one period of data from the .WAV file. The `period` parameter determines the time length of the period for one array. The `compress` parameter is used to turn on the compression algorithm chosen by the entries on the `argname` parameter. The number entered is a percentage. The final parameter on this model line is the `input` parameter. This value determines the .WAV file used for input. Depending on the `argname` parameter entries this value could have various meanings. See the Mat_Src section for more details.

The model line for the Sink Array model is quite similar to the Source model.

```
.MODEL mat_snk_001  mat_snk( op="sink"
+      argname="sound_device  wavelet daub20  play"
+      output=hello2.wav  )
```

All of the parameters have the same meaning as the source model discussed previously. Here we have an op parameter that instructs the Array model to use the Sink OLE Automation Server. The Sink Server will use the `argname` parameter entries for configuration. The `sound_device` entry instructs the Sink Server to use the available sound card to play the sound for each period if the `play` keyword is passed on the `argname` parameter. The `output` parameter is used to enter the filename for the output .WAV file if `sound_file` is passed on the `argname` line. Here we pass `sound_device` so no output file will be generated.

The circuit simulation begins by reading one `dim` worth of data from the device described by the Source model, X1 SRC_1. After one complete array has been read the Clk_out signal, node 16, goes high. This places a high on the Clk_in port of the Sink model, X2 Sink_1. This high instructs the Sink to process one array of data. The `play` entry on the Sink's `argname` model parameter causes the Sink model to play the resulting sound file as soon as the entire sound file has been processed. Hence, for a 3.6 sec .WAV file the simulation will have to run for a time greater than 3.6 seconds.

The nodata and reset ports of the Source model are used to signal the end of data and to reset the source respectively. The source can be made to start over by connecting the nodata port to the reset port. This will cause the source to reset as soon as nodata is detected.

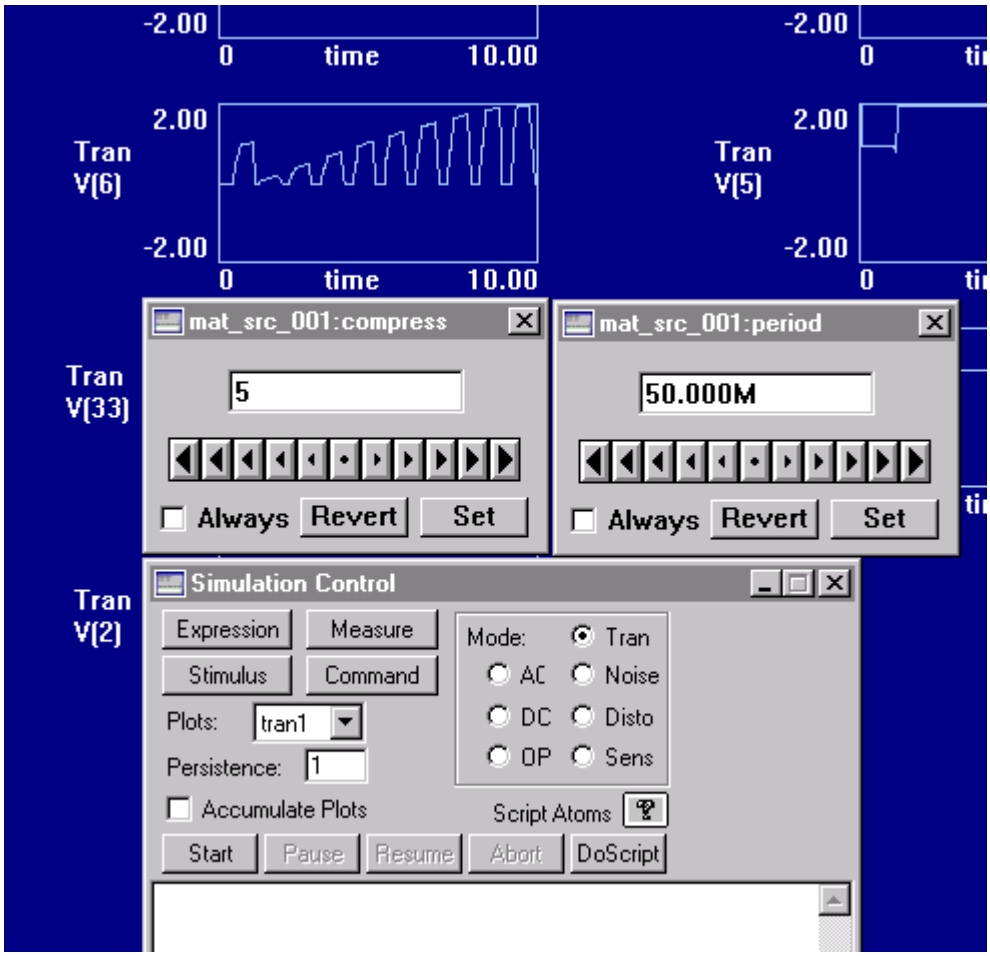
### **HelloC.cir** (Wavelet Compression)

The sound models include a wavelet compression capability. The SRC_1 model syntax,

```
op="source"
argname="sound_file  wavelet daub20   nplay"
dim=[4096]
period=.2
compress = 0 ; percentage
input=hello.wav
```

directs the model to get its data from the file "hello.wav" and apply a wavelet transform using the daub20 mother function. The array contains 4096 points for each .2 second period, and there is no compression. Compression, when selected, will eliminate all but the highest amplitude components. Compression can be changed interactively allowing you to hear the effect as the compression value is changed.

The Microsoft sound applications interface has no provisions for queueing sounds. If the simulation runs fast enough with both input and output sound enabled, the output won't be played. The `nplay` is a convenient place holder for you to remove the "n" to play the input sound. Shown below is the IsSpice4 control panel along with the stimulus dialog setup used to study the compression algorithm. See the IsSpice4 User's Guide for information about interactivly sweeping circuit values.




There are several ways to get a graphical view of an array node. In the examples all of them. In HelloC.cir a wavelet compressed node is to be viewed. This requires special processing. Render is a 32 bit InProcess Server that connects to OLE compliant container application and allows array data to be viewed in real time.

To start Render;

- Double click on the Oclient application located in the Oclient directory of the CMSDK.
- From Oclient's Edit menu, select the Insert renderArray function.
- The renderArray object will be placed and the render dialog will be displayed.
- The insertion pointer will be in the Project edit field. Fill in the name of the circuit. **Note:** The name is case sensitive. Use the File Manager, or Explorer, to determine the proper case for the circuit name.
- Click in the Node edit field. Enter the node number, or name, of the node you wish to view. **Note:** The Node name is case sensitive.

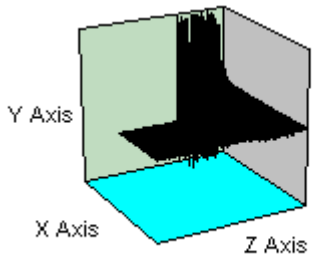
Due to OLE mechanics, the keyboard input belongs to the client application and not the server. The TAB key will not function.

- Click the Apply button.

After clicking the Apply button you can close the render dialog by clicking the Close button. Render and Oclient are now ready to view the data for the desired node. As soon as the simulation is started a link will be made and the data resulting from the simulation will be displayed.

- Start the simulation of Helloc.Cir.

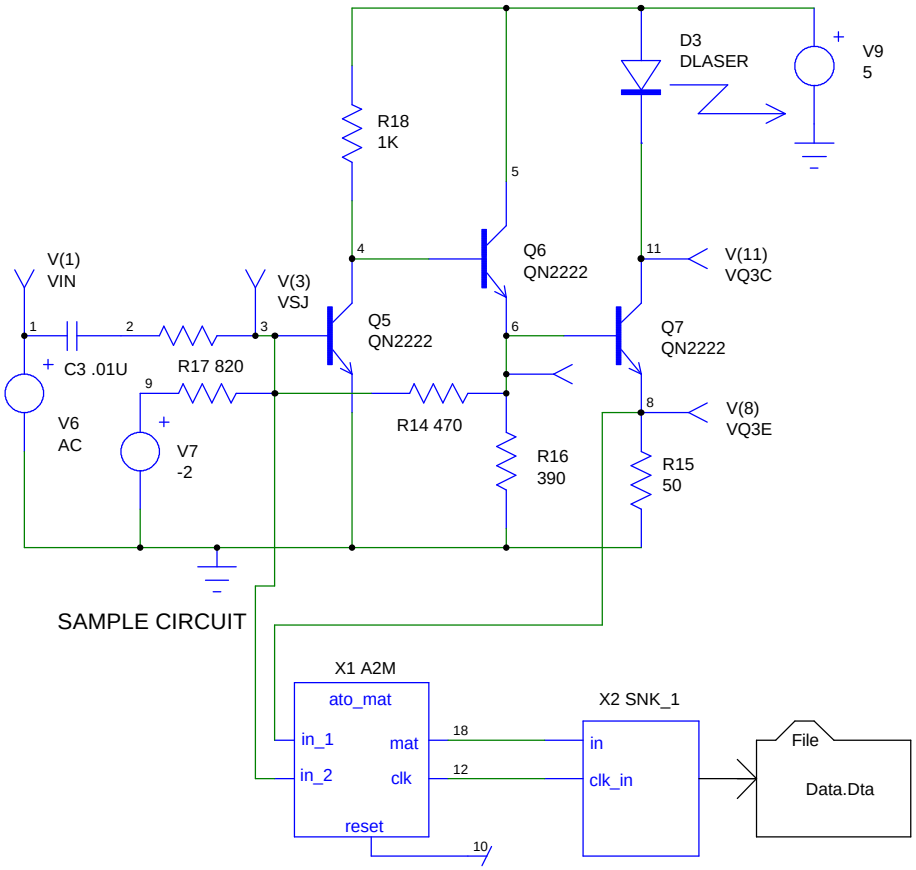
Notice, that data for the compressed waveform is displayed as the simulation progresses. The following figure represents the compressed data for Helloc.cir. If render is installed, and registered, you can play the view by double clicking on the figure.




Each of the traces in the 3D graph shown above keeps the largest values in wavelet space, nulling out the remaining values. This would give a constant bandwidth solution, even when no sound is present; clearly a better solution is to set an absolute threshold in order to eliminate silent periods. You may wish to make your own compression/decompression algorithm and test it using SALT!

## Data.Cir

The Array code models can be used to sample data from an analog simulation and use the sampled data as input to another analog simulation. In this example we will sample two nodes of a sample analog circuit, Sample.cir, using the Array models. The circuit for this example is shown below.




The portion of the schematic that is of interest in this example consists of the A2M, Analog-to-Matrix model and the SNK_1, Sink Array model. These are X1 and X2 respectively.

The Analog-to-Matrix model was discussed in the Bridge.Cir example. This model takes the node voltages for V(3) and V(8) and places them in an Array matrix that is sent to the Sink Array model. The Sink Array model was also discussed in a previous example, Hello.Cir. However, in this example we are not playing a sound, we are recording data. This configuration of the Sink model is established by the Sink's model line shown below.

```
.MODEL  data_snk  mat_snk( op="sink"    ; use the  Array_sink server
+       argname="matrix_file delay=99n tol=10u"
+       output=Data.dta   )
```

The `argname` parameter for this model contains three new entries to configure the output for the sampled data. The `matrix_file` entry tells the Sink server to output a file containing matrix elements. The `delay` entry is used to compensate for the period delay built into the array models. It eliminates the first "dead" period of output.

The `tol` entry assures that the data can be reconstructed using linear interpolation to an absolute tolerance of 10u.

The result of the circuit is a file, Data.dta, that contains the sampled data from the simulation. The sampling rate is determined by the mode's `tol` parameter and the time steps selected by the IsSpice4 simulator. The format of the file is quiet simple. A small header is placed at the top of the file followed by the sampled data. A partial listing is shown below.

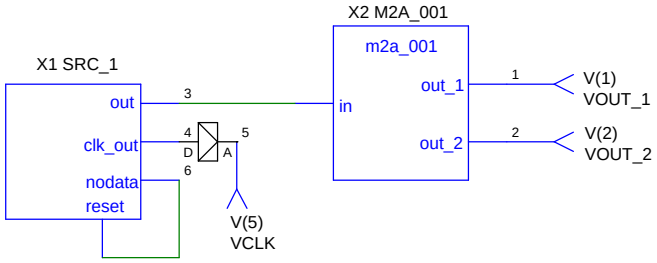
```
Project: DATA
```

```
Instance: a56:x2
SCALE 1
DATA REAL
varstep
0 1.45871  0.659409
7.80315e-010 1.4505  0.667121
1.56063e-009 1.39972  0.67796
2.34094e-009 1.31225  0.685685
3.12126e-009 1.18836  0.68861
```

A description of the contents of the matrix file can be found in the Data File Syntax section.

## Stimulus.Cir

In the previous example we saw how to sample an analog simulation using the array code models. In this example we use the sampled data as stimulus to an analog circuit. The circuit we will use is shown below.

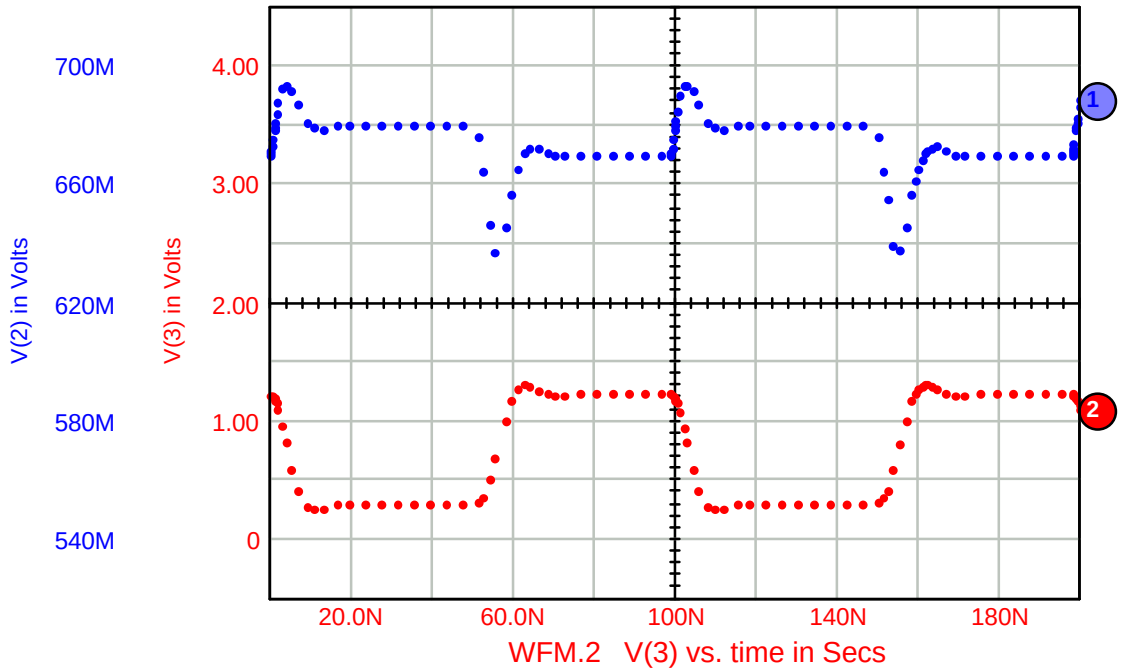



The two main components of this circuit, X1 SRC_1 and X2 M2A_001, have been discussed in previous examples. The new feature we are demonstrating is controlled by the Source Array model found in SRC_1. The model statement is shown below.

```
.MODEL data_src MAT_SRC(OP="SOURCE"
+         ARGNAME="MATRIX_FILE"
+         DIM=[128 3]
+         PERIOD=99N
+         COMPRESS = 0
+         INPUT=Data.DTA  )
```

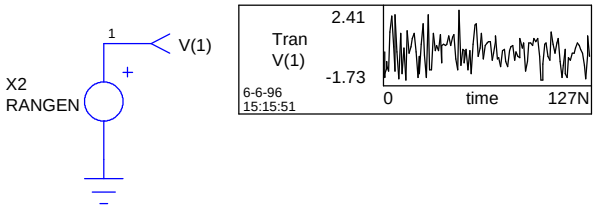
All of the parameters are similar to previous examples. The only new item in this model is the parameter assigned to `argname` . The `matrix_file` entry instructs the Source server to read its input from a file name by the `input` parameter. The source reads a periods worth of data into the array matrix established by the `dim` model parameter. After every complete period the Clk_out signal goes high. At the end of the data file the nodata signal goes high. This is fed into the reset signal which instructs the source to repeat the contents of the file. Hence we have a repetitive input signal from a finite set of data.

When playing back, the last array read in is played over and over if the simulation is run for a longer time than specified in the array. If you want the signal to stay at the last value, copy the last value in the file and change the time to something greater than the maximum simulation time. If the nodal output is connected back to the reset input, the data input will be recycled at the first period after the last input time. For periodic signals it is best to specify exactly one period of data so it won't matter if the nodata output is connected to the reset input. The following figure shows the result for several cycles, illustrating how the time step is reduced for rapidly changing data without the usual IsSpice4 breakpoints being introduced.




## Random.cir

This example illustrates how a random noise generator can be introduced into a simulation. The schematic is shown below.



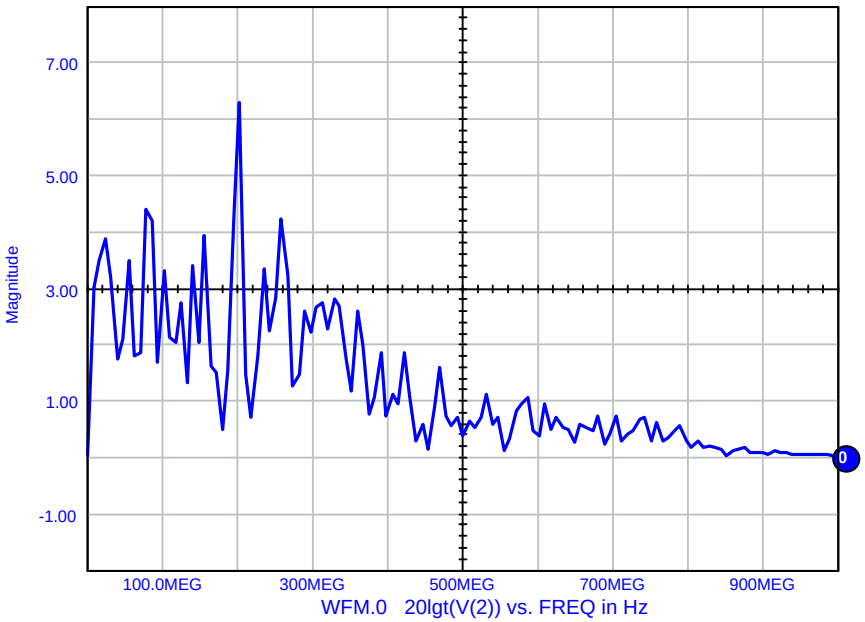

The circuit consists of one subcircuit that contains an Array Source model and an Matrix_to_analog bridge.

```
.SUBCKT RANGEN hi  lo
.MODEL MAT_TOA_003 MAT_TOA(DIM=[128] LOC=[1 1 ] )
.MODEL MAT_SRC_001 MAT_SRC(OP="SOURCE"
+       ARGNAME="RANDOM SEED={ SEED }"
+         DIM=[128]
+       PERIOD=99N
+       COMPRESS = 0
+       )
A1 99 CLOCKOUT2  RESET  RESET MAT_SRC_001
A2 99 [2] MAT_TOA_003
E1 hi  lo 2 0 {  Mag }
.ENDS
```

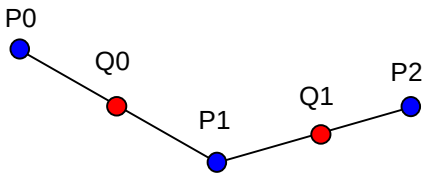
The subcircuit uses Intusoft's parameter passing feature to facilitate experimentation with the random number seed. (See the IsSpice4 User's Guide for details about Parameter Passing.) The array model of interest for this example is MAT_SRC_001. The parameters for this model have been discussed previously. The only new items are the entries on the `argname` line. The `RANDOM` entry instructs the Source server to generate random values to be used as input to the simulation. The random numbers that are generated are controlled by the `SEED` value specified.

The SEED value is used to initialize the random pattern for each simulation. Using different SEED values for different random sources will insure that the sources are uncorrelated, but give repeatable data for each simulation.

The resulting IntuScope FFT show the effects of time step and accuracy on the frequency spectrum.




The time step is automatically constrained by the Matrix-to-Analog bridge to give the default accuracy to the analog node ( steptol=.005). However, since the internal time step of the simulation varies with circuit activity the value used for a specific analog time may be an interpolation of adjacent values as shown below.




In the above figure points P0, P1, and P2 are the data generated by the random noise source. Points Q0 and Q1 are interpolated points. This value to value correlation could, in one extreme, produce completely independent points (no Q points). In the other extreme the interpolated points become the average of the points produced. This causes the signal to be the average of successive random numbers. This phenomena will cause the high frequency terms to be attenuated and is apparent from the FFT of the noise signal. (Note, the FIR filter response of the average of 2 samples is a cosine shape, going to zero at ½ the sampling frequency).

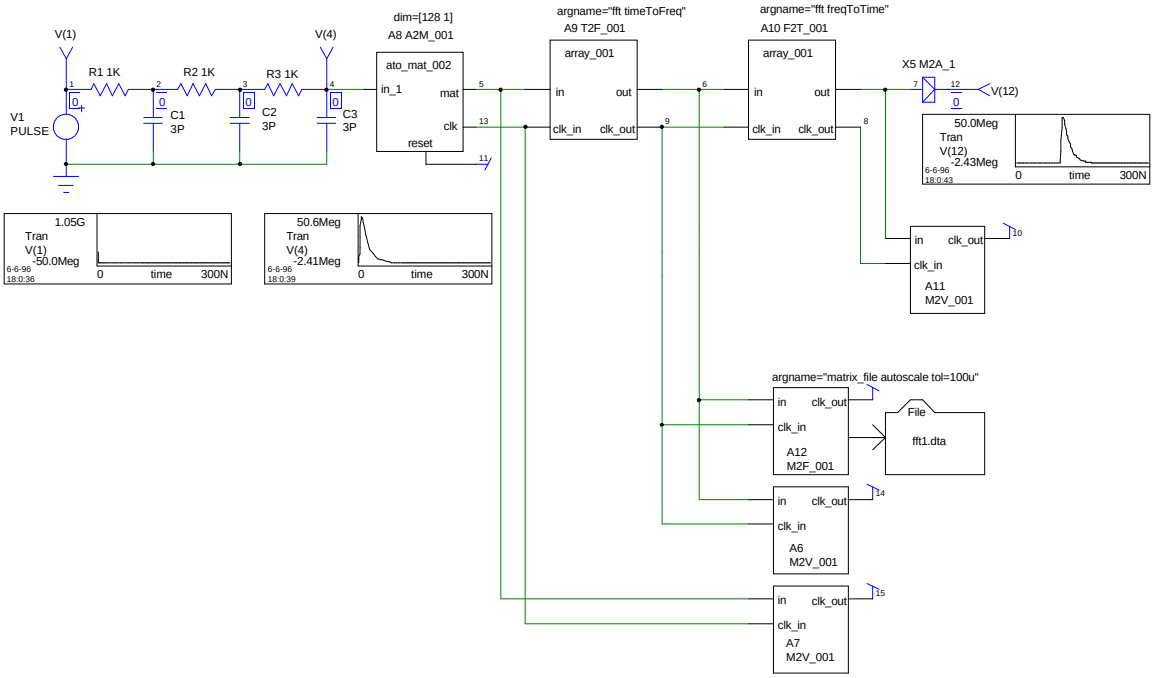
## **Vector.cir** (Creating IsSpice4 vectors to view array data)

Viewing array contents is accomplished using 3 methods:

1. Convert array variables into analog nodes using node bridges.
2. Convert array variables into analog IsSpice4 vectors using a sink server.
3. Use Render.dll to view data formats for which plot drivers have been created.

This example focuses on the second method to illustrate viewing array data. The second method uses less computational overhead than the first method because the analog data does not have

to be processed for each iteration. It also allows output to be viewed for cases not supported by the third method, such as complex data. The schematic for this example is shown below.




From the schematic we can see that an analog signal is converted to an array matrix, A8 A2M_001, that is then sent to a Sink mode, A7 M2V_001 (Bottom), controlled by the following model statement.

```
.MODEL M2V_001 MAT_SNK( op=sink argname=vector)
```

Here we see the familiar op parameter signifying the use of the Sink OLE Automation server. We also see a new entry for the `argname` parameter. The `vector` entry on the `argname` parameter instructs the Sink server to produce an internal IsSpice4 vector for the output matrix node it is connected to. For this case, we see the Sink model A7 connected to node 5 of the Analog-to-Matrix model A8. This produces an analog vector for the output of A8. The resulting vectors are enumerated for each element of the matrix array. Here we have a [128 1] array matrix. Therefore, we get an IsSpice4 vector of A8_1. If this were a [128 2] we would get two vectors, A8_1 and A8_2. The enumeration is done from the first memory location to the last. If you recall the figure shown in the Array section we see that an [8 3 4] array matrix would produce A8_1 to A8_12. One IsSpice4 vector for each of the vectors in the array matrix. For each set of IsSpice4 vectors created for an array matrix node an independent variable is also created. This independent variable will be named relative to the array model such as A8_time or A8_freq. When displaying the IsSpice4 vector data it is important to use the correct independent axis data.

The output from A8 is also sent to an array model that performs an FFT. The result of this FFT is sent to two Sink models; A12 M2F_001 and A6 M2V_001. The Sink model A6 is identical to the Sink model previously discussed. The Sink model A12 is used to produce the array matrix output similar to that covered in the Data.Cir example.

The vectors created by this Sink function are available during a simulation and while the IsSpice4 simulator is running. A list of available vectors can be obtained by issuing the IsSpice4 ICL command display from IsSpice4's Simulation Control dialog. Please see the IsSpice4 User's Guide for more information.

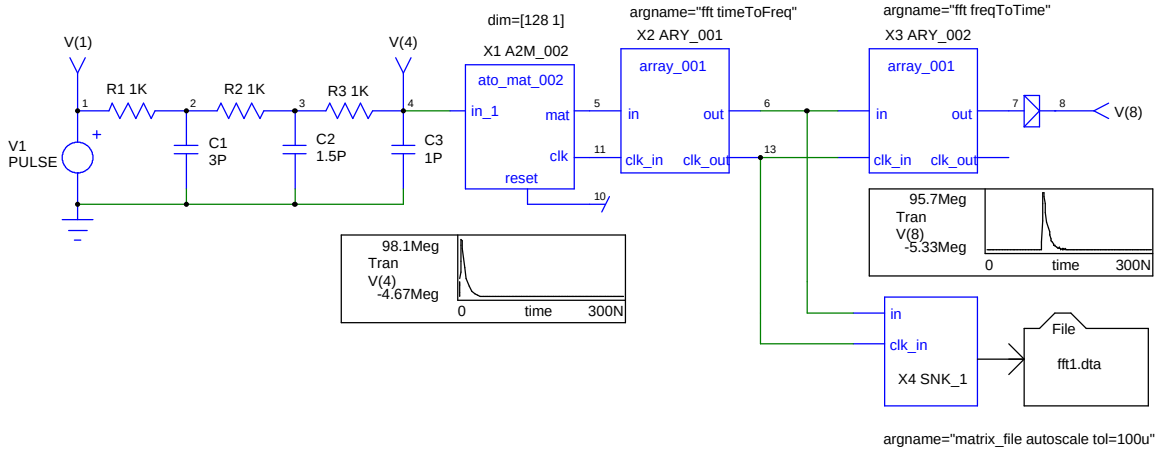
The result of a Display command for this example is shown below.

```
V(1)          : voltage, real, 102 long
V(12)          : voltage, real, 102 long
V(4)          : voltage, real, 102 long
a10_1          : notype, real, 128 long
a10_time        : notype, real, 128 long
a8_1          : notype, real, 128 long
a8_time         : notype, real, 128 long
a9_1_im         : notype, real, 128 long
a9_1_re         : notype, real, 128 long
a9_freq         : notype, real, 128 long
time          : time, real, 102 long [default scale]
```

These vectors can be accessed using IntuScope. Notice that each sink instance has its own copy of an independent variable. You need to choose the independent variable the belongs to the correct array in the IntuScope dialog, effectively making this an x-y plot. (See the Design Entry And Data Analysis Manual for details on using IntuScope.)

## FFT.Cir

This example illustrates the use of the Fast Fourier Transform, FFT, to produce the frequency domain response that is the impulse response of a test network. In a later example we will use this response to build a convolution filter. Several unique features of the matrix file I/O are illustrated in these examples.




In this example we find two new Array elements, X2 and X3, performing an FFT and an IFFT respectively. The remaining elements have been discussed in other examples.

This circuit takes the analog output of a simple RC ladder and feeds it to an Array-to-Matrix bridge. This bridge reads one period of data, as defined by the A2M_002 subcircuit, and feeds it to the ARRAY_001 subcircuit. This subcircuit consists of an Array model with the following model statement.

```
.MODEL array_001 array( dim=[128 1]
+      o="fourier"
+      agname="fft  timeToFreq")
```

This model describes an Array model that uses the Fourier OLE Automation server, given in the op parameter, to perform the function specified in `argname`. The output of this Array model flows to another Array model described by the following model statement.

```
.MODEL array_002 array( dim=[128 1]
+      op="fourier"
+      argname="fft  freqToTime")
```

This model, similar to the previous Array model, uses the Fourier OLE Automation server. This device performs an inverse fft on the input data as specified by the `argname` parameter.

The output of subcircuit X2 also flows to a Sink Array model that is controlled by the following model statement.

```
.MODEL mat_snk_001  mat_snk(op="sink"   ; use the Array_sink server
+       argname="matrix_file  autoscale tol=100u"
+       output=fft1.dta  )
```

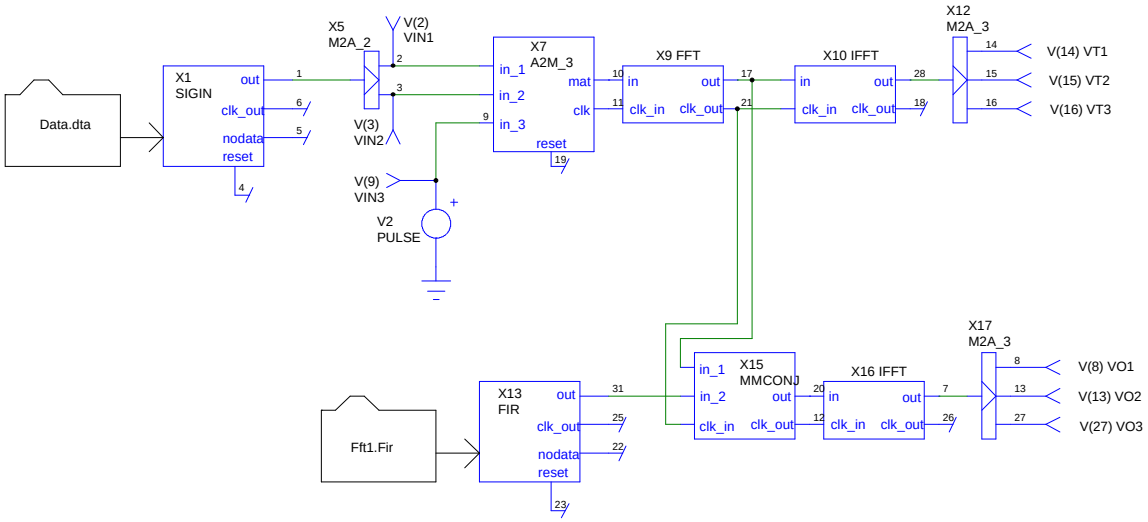
Here we see that the sink model is used to construct a `matrix_file` named fft1.dta. This file is the desired output of this example. A portion of the contents of this file are shown below.

```
Project: FFT
Instance: a3
SCALE AUTO
DATA COMPLEX
varstep
0 0 0
0.984375 0 0
0 0.98685 0.00338636
0.015625 0.759111 0.481766
0.03125 0.378214 0.619598
...
0.984375 5.92339e-014 1.03307e-015
```

Notice that the output is complex. The complex data type was automatically determined by the fft model. Autoscaling is set using the autoscale keyword. This means that the independent variable is normalized, running from 0 to 1. When the data is read back, for filtering, it will be adjusted to the period of the array. In this manner, a filter can be applied to a wave shape, independent of its frequency. In order to compress the stored data, the `tol=100u` parameter causes the output of data for an absolute accuracy of 100u. The error limit assumes data will be uncompressed using linear interpolation; therefor, a change in slope is necessary to cause extra data to be saved.

## FFT_Conv.Cir

In this example we use the data collected by the previous example, fft1.dta, and the data collected from the Data.Cir example, Data.Dta, to construct a convolution filter.




Recall that the output of the previous example was the impulse response of the RC circuit. The data collected represent the impulse coefficients for the filter. These coefficients are used in a frequency domain multiplication to filter the signals collected by the Data.Cir example.

In order to use FFT1.Dta as input to the filter we first have to edit the file. First enter the following line before the DATA COMPLEX line

MATRIX_TYPE = STANDARD

This is required because the data is used as input to the circuit. Recall from the **XXXX** section that matrix types are inherited. Since there is Array model to inherit from at the input we must set the matrix type. This keyword is not required for the Filter.Cir example since the file is used as input to a model that inherits a matrix type.

After entering the matrix type remove the second and third data lines;

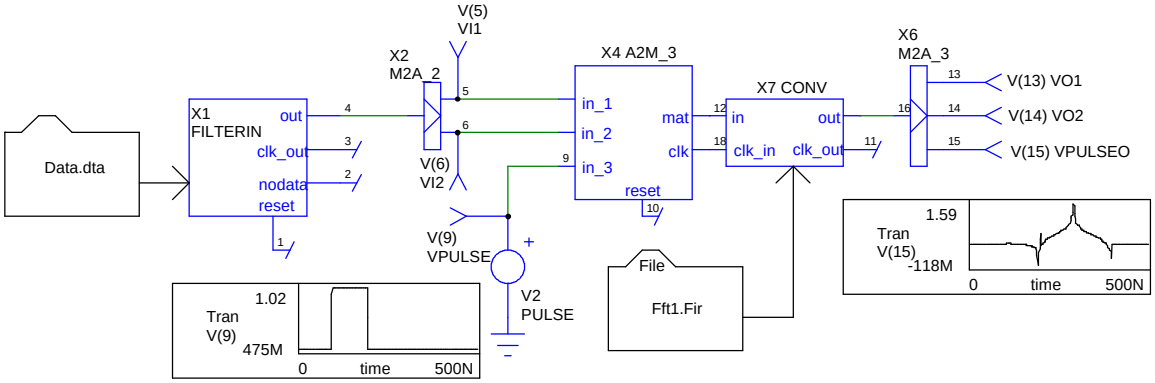
```
0.984375 0  0
0 0.990555 0.00313012
```

from the file. This data represents the erroneous points for the first “dead” period. After doing this move to the end of the file and place the END keyword after the last line. See the FFT1.FIR file for more a complete listing of the input file.

The inputs VIN1, VIN2, and VIN3 are transformed to the frequency domain by means of the FFT performed by X9. The output of this FFT is sent directly to an IFFT block where the original signal is recovered. (Delayed by the periods specified.) The output of the FFT block is also sent to X15 MMCONJ where the frequency domain input signals and the impulse coefficients from fft1.dta are multiplied. This output is transformed back to the time domain by the IFFT block and the output is displayed as VO1, VO2, and VO3.

## Filter.Cir

This example combines all of the aspects of the previous example to demonstrate the convolution filter built into the Array model.




The element of interest here is the convolution element, X7. This model eliminates the need for the FFT, IFFT and MMCONJ blocks of the previous example. The model statement for the array model is shown below.

```
.MODEL CONV_001 array(    dim=[128 3]
+                         op="fourier"
+                         argname="convolution"
+                         paramfile="TEST1.FIR")
```

As with the FFT and IFFT models discussed in the previous examples this array model uses the Fourier OLE Automation server to perform the convolution function described by the `argname` parameter. The additional parameter, `paramfile` , contains a text description of the coefficients for the convolution filter. This file is shown below.

```
SCALE AUTO
DATA COMPLEX CIRCULAR
varstep
0 1 0
.00001 0 1
1 0 1
END
```

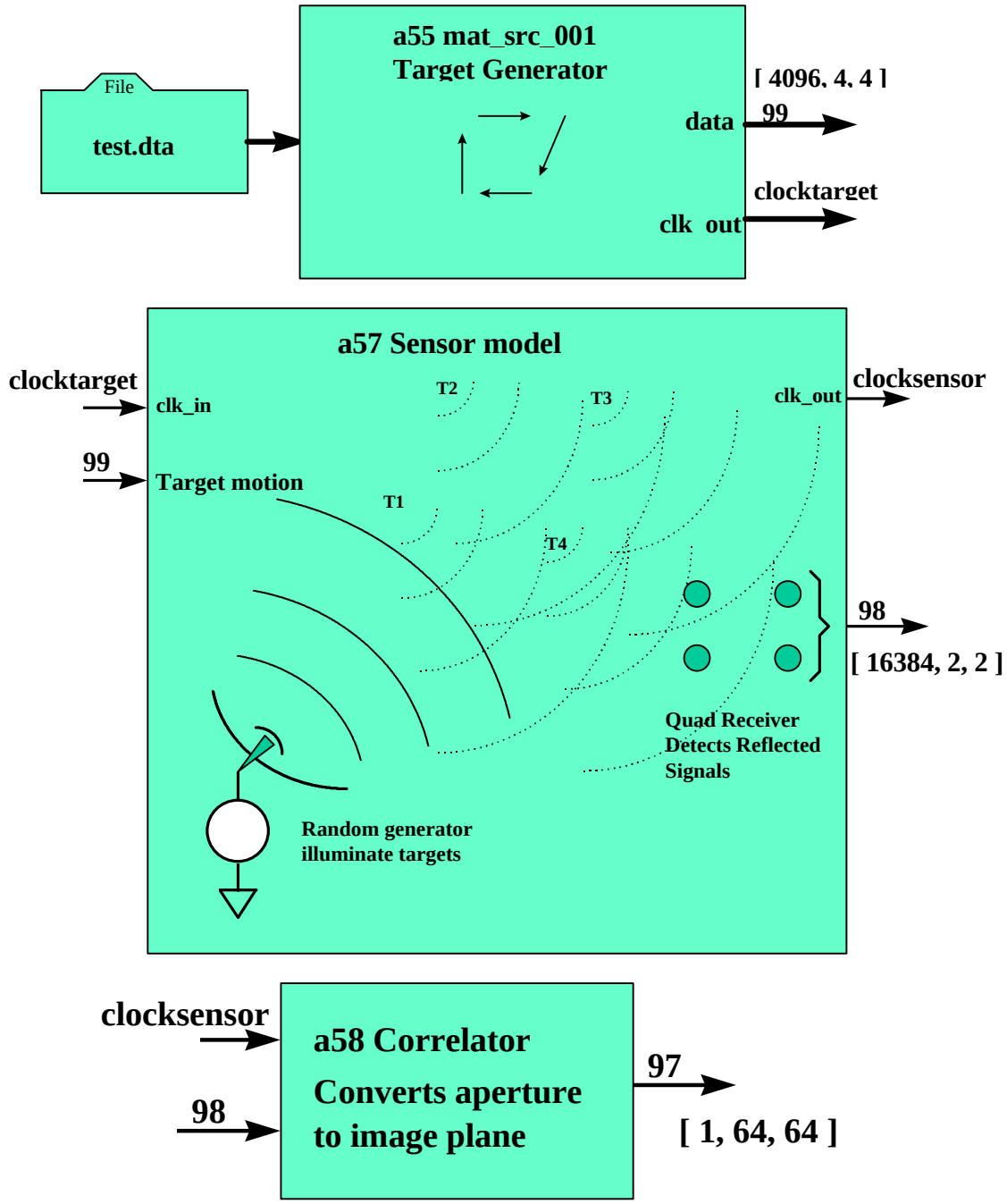
The improvements made in this model over the pervious implementation of a convolution filter, FFT_Conv.Cir example, is that circular convolution is handled without extensive work. That is, by entering the CIRCULAR keyword on the DATA line of the input filter file we instruct the convolution algorithm to perform circular convolution. Time domain data, when transformed to

frequency, has unique properties at frequencies greater than ½ the sampling frequency. The resulting frequency spectrum, above this Nyquist frequency is the complex conjugate of the lower frequency spectrum, due to the time series having no imaginary terms. When using this spectrum in a filter, it produces what is called “circular” convolution which causes the filter to appear periodic. For aperiodic filtering, it is necessary to make the second half of the frequency spectrum zero. This effectively doubles the length of the filter. If you are applying a filter to a periodic wave form use the CIRCULAR keyword. Note that these convolution filters are only approximations of continuous filters. The impulse response must approach zero at the end of the filter period. Filters that don’t meet this criteria will tend to become circular.

Since the FFT_Conv.Cir example did not account for the circular nature of the data the answers will appear different. This is most evident if you examine the pulse input, V2, as it propagates through both circuits. It is not as evident in the periodic waveforms such as the first output node VO1 in both circuits.

## Sensor.Cir

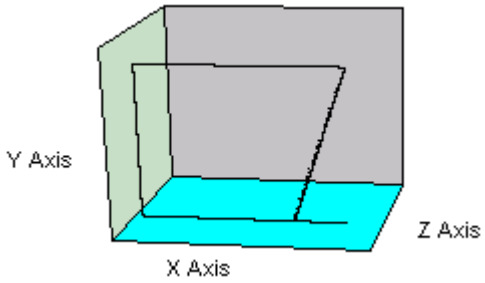
As shown in the Data File Syntax section, a SALT source can represent a family of points flying through a 3 dimensional space. The following Figure puts this together with a sensor model that illuminates the objects with a continuous pseudo random signal.




The return signals are detected using a quad sensor. The four time varying target signals, T1, T2, T3, and T4, can be thought of as samples at the aperture of a large lens. It may then be possible to map these signals to the focal plane and image the objects. To do this the way a camera works, one would perform a 2 dimensional Fourier transform, described in literature as a Fresnel equation.

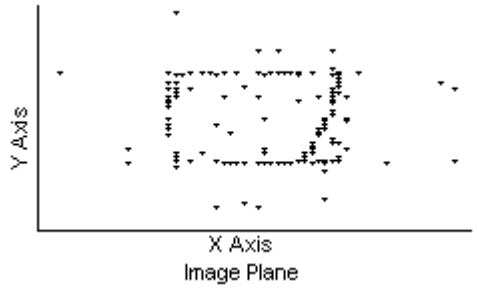
There are 2 major differences for the problem as presented. First, the illumination of each object (T1, T2, T3, and T4) comes from the same source of radiation so that there is a cross correlation between signal received at each element. Secondly, the aperture sample uses only 4 points. Using the Fresnel equation, there are only as many image plane pixels as there are aperture sensors. Imaging many pixels from a few sensors is very similar to the imaging problems faced by Radio Telescopes. The main problem is that radiation from several objects can combine to

produce phantom or ghost images. Using a coherent illumination source lets us perform multiple cross correlation's; not only along the x and y axes but also along the diagonals. Unfortunately, the diagonal beams are narrower than those formed in the x-y direction so that it is possible to miss a target. If the beams are widened, then more ghost images are formed. In the following figure, (origin =-50,-50,0  scale =100,100,100 )



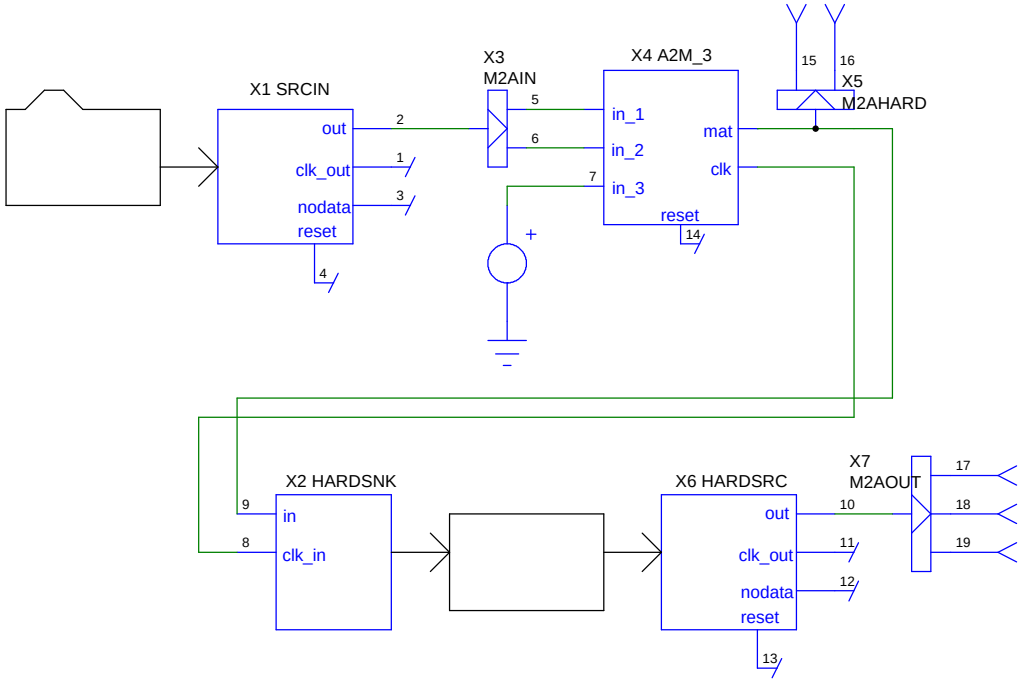

the target tracks are shown over the 10 seconds of simulation.

The following figure (origin = 0,0  scale =64,64 ) is the processed image plane data seen on the 64 x 64 image plane, recovered from a quad aperture sensor, illustrating the ghost image and showing the actual image. The source code for the detector has several optional switches that can be invoked from the debugger at run time to evaluate tradeoffs between detecting false images and missing real images.




## Hardware.Cir

In this example we will use two new functions in the Source and Sink OLE Automation Servers. These functions provide access to external hardware during an IsSpice4 simulation. The schematic representing this simulation is shown below.




From the figure we can see that the input to the simulation is read from a file of sampled data. The input data can be taken from any source. A file was used here for the sake of simplicity. This sampled data represents the output from the Data.Cir example. This data is converted to analog signals that are combined with a standard IsSpice4 pulse source and transformed back to an Array matrix. These three signals are then sent to the Sink Array server that calls the dac function of the Sink server. This is shown in the following model statement.

```
.MODEL MAT_SNK_001 MAT_SNK(OP="SINK"
+       ARGNAME="DAC NATIONAL INITFILE=AFILE.DTA"
+       OUTPUT=NIDAC2.CFG   )
```

The `argname` value specifies the functions type, `DAC`, card vendor `NATIONAL`, and the initialization file, `INITFILE`. The initialization file is used to centralize the configuration information for the specific digital to analog converter (DAC) card being used. The format is shown below.

```
BOARD=AT-AO-6/10
GROUP=1
DEVICE=2
RATE=58k
CHANNEL=0,1,2
ITERATIONS=0
GAIN=10k ,.6k , 500
OFFSET=-.66 , -.7 , -.5
END
```

Once the signal leaves the HARDSNK subcircuit it is a live signal ready to use at a test bench.

When returning from the test bench the signals enter the HARDSRC, Source Array server. The controlling model statement is shown below.

```
.MODEL MAT_SRC_002 MAT_SRC(OP="SOURCE"
+       ARGNAME="ATOD NATIONAL "
+         DIM=[128 3]
```

```
+       PERIOD=99N
+       COMPRESS = 0
+       INPUT=NIATOD.CFG )
```

The INPUT parameter is used to specify the configuration file. The format for the configuration file is shown below.

```
BOARD=AT-2150
GROUP=1
DEVICE=1
RATE=10k
CHANNEL=0,1,2,3
ITERATIONS=0
GAIN=.9e-4
OFFSET=0
COUPLING=DC
TRIG_CHANNEL=2
TRIG_LEVEL=0
TRIG_TYPE=ANALOG
TRIG_SLOPE=1
END
```

The signals are then sent to a Matrix-to-Analog bridge to be used in an analog simulation.

# Appendix A: IsSpice4 Vectors

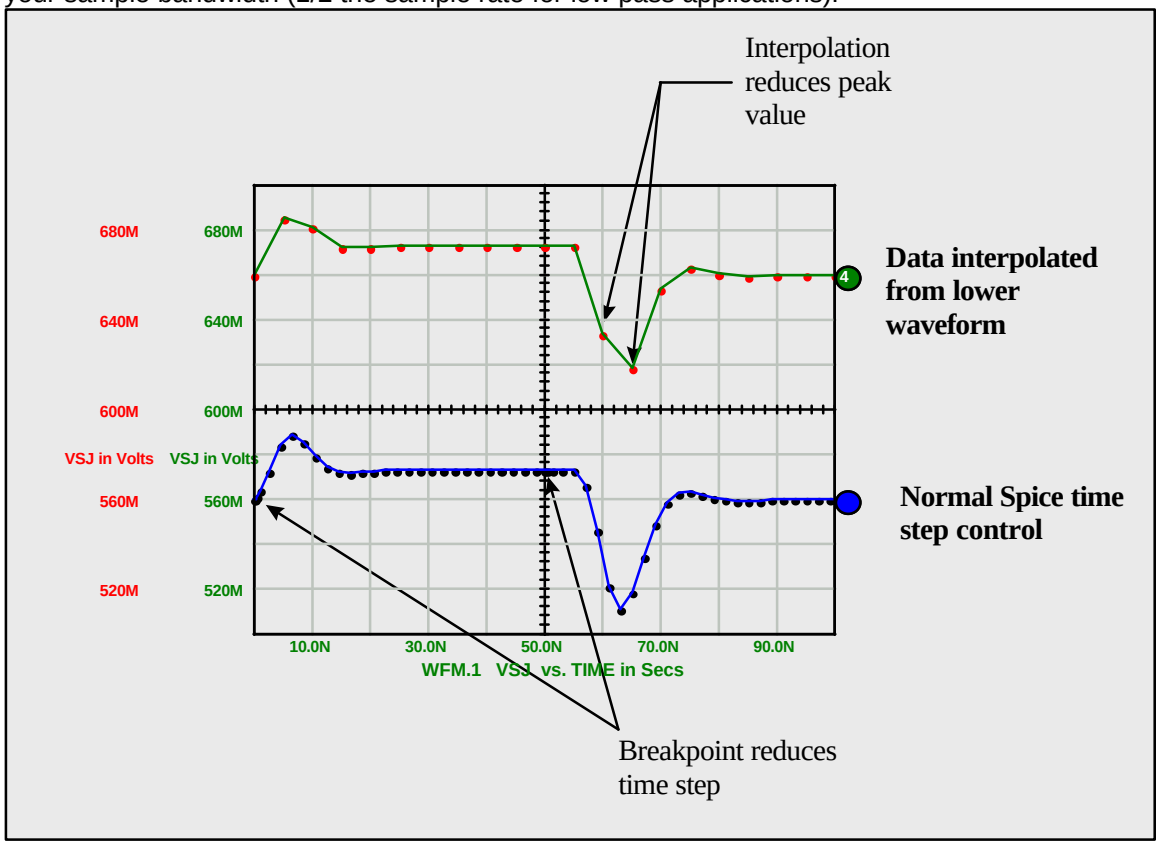
Spice accumulates output data for each analysis as a set of vectors. Data is output to each vector as the simulation proceeds; therefore, the nth element of each vector corresponds to the same analysis point. In the AC analysis, the data points correspond to frequency. In the transient analysis, the points correspond to time. Spice stores a default vector, time or frequency so you can find the independent variable for each plot. You can get a summary of information about the current plot vectors using the “display” command in the IsSpice4 Simulation Control Panel Script field as shown below for the sample.cir project.

Title: SAMPLE
Name: tran1 (transient)
Date: Tue Dec 05 13:02:45  1995

| | |
|---|---|
| V(1) | : voltage, real, 61 long |
| V(11) | : voltage, real, 61 long |
| V(11) | : voltage, real, 61 long |
| V(3) | : voltage, real, 61 long |
| V(3) | : voltage, real, 61 long |
| V(8) | : voltage, real, 61 long |
| V(8) | : voltage, real, 61 long |
| time | : time, real, 61 long [default scale] |

Spice vectors can be converted from variable spacing to uniform spacing using an interpolate function. Interpolation is available using the linearize command in the Simulation Control Panel Script field, as a check box in IntuScope or as an API call in the SALT SDK. When the independent vector is uniformly spaced, then it is possible to get meaningful information from mathematical operations such as mean, rms or standard deviation. The uniformly stepped storage is what we generally use for Array Processing. The graph shown below illustrates the data point spacing for the original Spice vector and the interpolated vector from the output of the sample project. You can see that there is a loss in accuracy where Spice collected lots of data and the interpolator output fewer data points.

When using Arrays, you will have to set the vector size sufficiently large in order to accommodate the highest data rate of significance and minimize this type of data loss. From a practical point of view, there cannot be significant data content at frequencies that are outside of your sample bandwidth (1/2 the sample rate for low pass applications).




If you are using the SALT SDK, then you have access to create and view Spice Vectors. The code in toarray.cpp illustrates how to read vector data, and the code in sink.cpp shows how to create vectors. Moreover, the entire Interactive Command Language, ICL, is at your disposal through the command data structure, accessed using spc_coms_ptr illustrated in errors.c and sinkxtra.c. ICL includes a math parser so that you can perform a wide range of operations. Unlike Spice, you can perform these operations while the simulation is running.

## Appendix B: Names Used To Identify Servers And Files

The following names are used in various combinations to access data files and DLL's:

1. Project name
2. Instance name
3. Node name
4. The op model parameter name

Any process that wishes to examine SALT data must know the first three names. Table SAT.1 shows how these names are combined to give access to various to the data.

| Name | Function | Example |
|---|---|---|
| Model parameter op=<server name> | Used to derive the registry program ID name ProgID = Array_<server name> ProgID = Array_fourier stored in <server name>.dll | fourier.dll |
| Model's memory mapped file accessed using privatefileptr | Data storage and communication flags used by array, source and sink servers projectname_instancename_nodename | sample_a2_99 |
| Node's memory mapped file | used by array, array_to_analog, accessed using publicfileptr sink and rendering servers projname_ nodename_public | sample_4_public |
| Running object moniker | used to identify running objects, that is, nodes being rendered by the render server projname_ nodename_public_user_<view number> | sample_4_public_user.2 |
| Running object file | a memory mapped file for each node that keeps track of the view numbers associated with active render objects projname_ nodename_public_view | sample_4_public_view |
| Real vectors | default, usually time<br>instance a2, vector 3 | a2_time<br>a2_3 |
| Complex vectors | default<br>instance a3, vector4 | a3_freq<br>a3_re_4<br>a3_im_4 |

# Appendix C: Memory mapped array data

Items marked by → are offsets to data stored in the memory mapped file after this data structure. They can be accessed using the following macro:

```
#define arrayPtr(array,offset,type)(type*)((char*)(array)+array->offset)
```

For example;

```
double * data = arrayPty((arrayFile *)aptr, arrayOffset,double)
```

The following typedef is reproduced from the common.h header file.

```
typedef  struct {
        char    modelData[256]; // a place for the server to sick instance data and status,
                                // including pointers
        char    errorMessage[128];      // a C string, if  errorMessage[0], output the
                                        // message and stop the simulator
        char    instancename[32];
        char    projectname[32];
        long    numOutConnections;
        long    wordsize;       // number of bytes in a word
        long    blocksize;      // size of block to read in doubles, see    sourc.cpp

        MODELS  model;          // tell the server what model is invoked
        void    * instance;     // a copy of the c++ instance
        long    done;           // we have finished using private data, set to zero on reset
```

```
        double compression;     // the requested percentage used by server to compute the
                                // compressed  param
        long    compressed;     // if not zero, it tells us that we are using compressed
                                // data and gives us the length of that data
                                // the compressed vector; time for example, is stored at the
                                // end of the array
        double  end;    // the final value of the independent variable, usually time
        double  step;   // the step for the independent variable, usually time which is the
                        // period parameter

        double  period; // step times number of samples
                        // this will be returned to the model on initialization
        double cktstep;         // tstep from spice
        double cktstop ;        // tstop from spice
        double cktinit;         // initTime from spice
        double  ymax,ymin;      // scaling range from server
        long    render;         // a flag to tell the server the data has changed
                                // and needs to be rendered
               // if the server is connected to a client that draws the rendered data,
               // the server must issue an event to the client, the server clears
               // the flag when done
               // this requires the server to have a draw method...
        long     nodata; // a flag the server uses to tell the client its out of data,

        RESOLVE  resolve; // tells us what to do with this array when its wired
                                // to another array, SUM, REPLACE, etc.
        long    numDim;         // number or dimensions
→       long    sizeOffset;     // the size of each dimension stored at
                           // (long*)(start +  sizeOffset), start = (char*)&  arrayfile
        char    argname[ARGSTRLEN]; // the function arguments in    ascii
        long     numInFiles; // number in the list of input node names
→       long    inputFileNameOffset;  // name to access the input, each filename is
                                // FILENAMESIZE bytes long
                                // stored at (char*)(start +   inputFileNameOffset)
→       long    paramFileOffset; // name including path of a file
                                // that stores coefficients needed by the operation
                                // (char *)(start +  paramFileOffset);
        double state;           // generally the time stamp from the simulator,
                                    // which is the start time
        long    istate; // an alternate to state, incremented for each operation
→       long  arrayOffset; // array of doubles stored at
                        // (double *)(& arrayOffset +  arrayOffset)
        long arraySize; // number of doubles in the array
        } arrayFile;
```

# Appendix D: Error handling:

Error handling in the servers is accomplished in two ways. If the error needs to abort the simulation, then a text message is placed in the errorMessage member of the private memory mapped file. The ABORT… macros in array1.h takes care of loading the appropriate memory mapped file. You may need to set a flag in your server object when this is done so that no further action will be performed until initialization or re-initialization.

The presence of an error message in the private file will cause the array code model to abort Spice. If the simulator was running from the original netlist, the simulation will quit and the error will be reported in the ".err" file. If the simulation was running from the script window, then the error will be reported in the error/status window. You may opt to also display a dialog to help direct the user. If you do not want the simulator to abort; then you can either do nothing or put up a modal MessageBox. Errors can cause memory leaks, that is, memory can be allocated and not freed if the Spice simulator continues to run. This won't cause data errors, but could restrict performance.

If you are making your own array code model, then you can use IsSpice4 services directly. These include automatic out of memory messages and aborts if you use the MALLOC, CALLOC, REALLOC or FREE macros in codedefs.h. You can also use or modify the error_abort () method in errors.c.

Operating system errors are trapped using __try(), __except() around the entire Spice program. This allows us to do a somewhat orderly shutdown; however, neither Windows95 nor Win32s are completely protected and neither does a complete cleanup when a program terminates. This method only works when the error occurs in the IsSpice4 thread. Servers running in different threads should have there own error recovery methods.

## Arrays and Delays:

Arrays delay data. This allows easy insertion into IsSpice4 since the simulator knows ahead of time where to make break points. Instead of scaling back the time step, it is only necessary to adjust the time step to suit the analog input data accuracy. The down side is that iterations are not possible except with an arrays worth of delay. Most control system problems work this way; however, many models require iterations which can't be accommodated using this technique. When applicable this is a very fast way to couple external signal processing in and out of the analog simulator. All synchronous logic works this way, as well as array based signal processing.