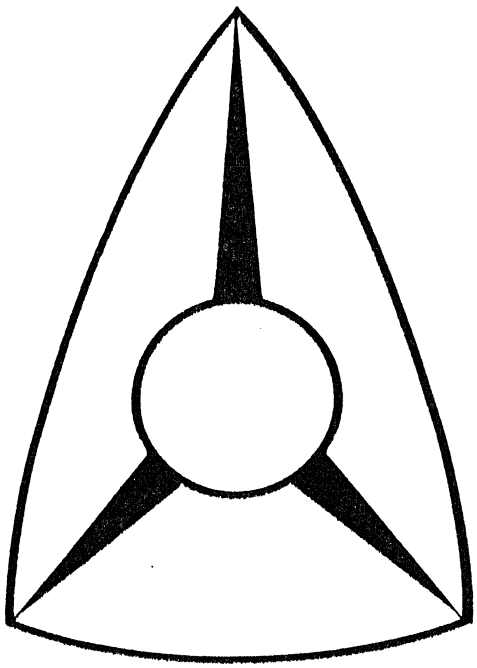
OSA90™

OPTIMIZATION SHELL ASSEMBLY

SOFTWARE SYSTEM

Optimization Systems Associates Inc.
Dundas, Ontario, Canada



Introduction to OSA90

advanced customization-oriented CAE software system

designed to interact with user's in-house programs to control and perform a variety of optimization tasks in a user-friendly environment

enables users to merge OSA's unique, continually evolving technology with their wealth of computer programs

can invigorate user's CAD/CAE capabilities without high in-house development cost or revealing proprietary codes to external organizations

OSA90 can save your company valuable time and money

State-of-the-art Optimizers

three state-of-the-art gradient-based optimizers:

(1) ℓ_1,
(2) ℓ_2 (least squares),
(3) minimax

they have a proven track record in electrical circuit and system optimization

the user supplies or defines individual error functions

the objective function is formulated and its minimization executed entirely by OSA90

sensitivity displays help the user to select the most crucial variables for optimization

OSA90 can take advantage of user-supplied gradients; alternatively, OSA90 will generate and use approximate gradients

Datapipe™ Communication

straightforward optimization problems can be directly defined in the input file using OSA90's Expression block

for complex problems OSA90 facilitates high-speed data connections to and from the user's in-house software

the key to this feature is Datapipe™, a user-programmable system allowing OSA's technology to be merged with non-OSA *executable* programs, running independently in separate processes

easily implementable by the user

maintains complete security of user's software

an in-house program, e.g., a circuit simulator, can be turned into a powerful CAE system, enhanced by an elegant user interface and quality graphics

Datapipe™ Communication

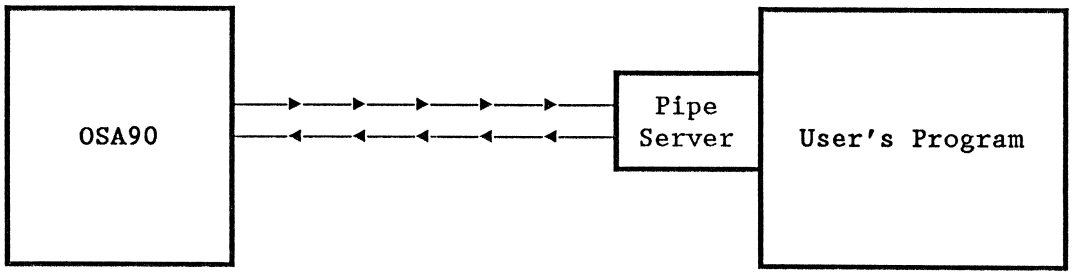
the user can define constants, variables and expressions in the input file

Datapipe handles all the parsing

the inputs received by the user's software can be both pure numerical values and character strings

the outputs from the user's program can be optimized, displayed, printed and/or saved

User Implementation of Datapipe™ Communication



employs inter-process pipes supported by all UNIX operating systems

typical READ and WRITE statements are used to receive and send data

a small pipe server (about 350 lines) establishes the protocols; OSA provides the source code of the pipe server

OSA does not need access to the user's source code

Pre- and Post-Processing

any of the output values returned from the user's program(s) can be postprocessed using standard mathematical functions and operations

the user can define his/her own responses for graphical display, numerical output, optimization or statistical analysis

any of the variables to be passed to the user's program(s) can be preprocessed

preprocessing allows the user to link variables, use output values of one program as input variables to another program, etc.

OSA90 Input File Overview

Expression
...
End

Sweep
...
End

Specification
...
End

MonteCarlo
...
End

Statistics
...
End

OSA90 Input File Expression Block

```
Expression

  x1: ? -0.5 ?;
  x2: ? 0.5 ?;
  z1: x1 * x1 + x2 * x2;
  z2: x1 - x2;

  Datapipe: SIM  FILE = user_program_name
                 N_INPUT = 4       INPUT = (freq, x1, z1, z2)
                 N_OUTPUT = 2      OUTPUT = (gain, sk);

End
```

OSA90 Input File Sweep and Specification Blocks

```
Sweep

    X1: from -1.0 to 1.0 step 0.1  F1 F2;
    X2: from -1.0 to 1.0 step 0.1  F1 F2;

    FREQ:     from 0.01GHZ to 1GHZ step 0.01GHZ
              from 1.1GHZ to 1.4GHZ step=0.05GHZ
              Gain, Insertion_loss;
    C6:       from 0.1NF to 0.5NF step 0.025NF  FREQ: 1ghz
              Gain, Insertion_loss;

End

Specification

    F1 < 1.0,  F1 > 3.0,  F2 < -3.0;

    FREQ:     from 0.02GHZ to 1GHZ step 0.02GHZ
              Insertion_loss < 0.53;
    FREQ:     1.3GHZ
              Insertion_loss > 52;

    x1:       from 0 to 10 step 1
              z1 < 10.3;

End
```

SIM Type of Datapipe™ Connection

outputs from the user's program are individually labelled

syntax:

```
Datapipe:  SIM  FILE = filename
                N_INPUT = n        INPUT = (x1, ..., xn)
                N_OUTPUT = m       OUTPUT = (y1, ..., ym);
```

the user specifies:

filename	-	the name of user's executable program
n	-	the number of numerical values to be passed to the user's program
x1, x2,..	-	individual labels and/or values to be passed to the user's program
m	-	the number of numerical values that will be returned from the user's program
y1, y2,..	-	individual labels for the returned values

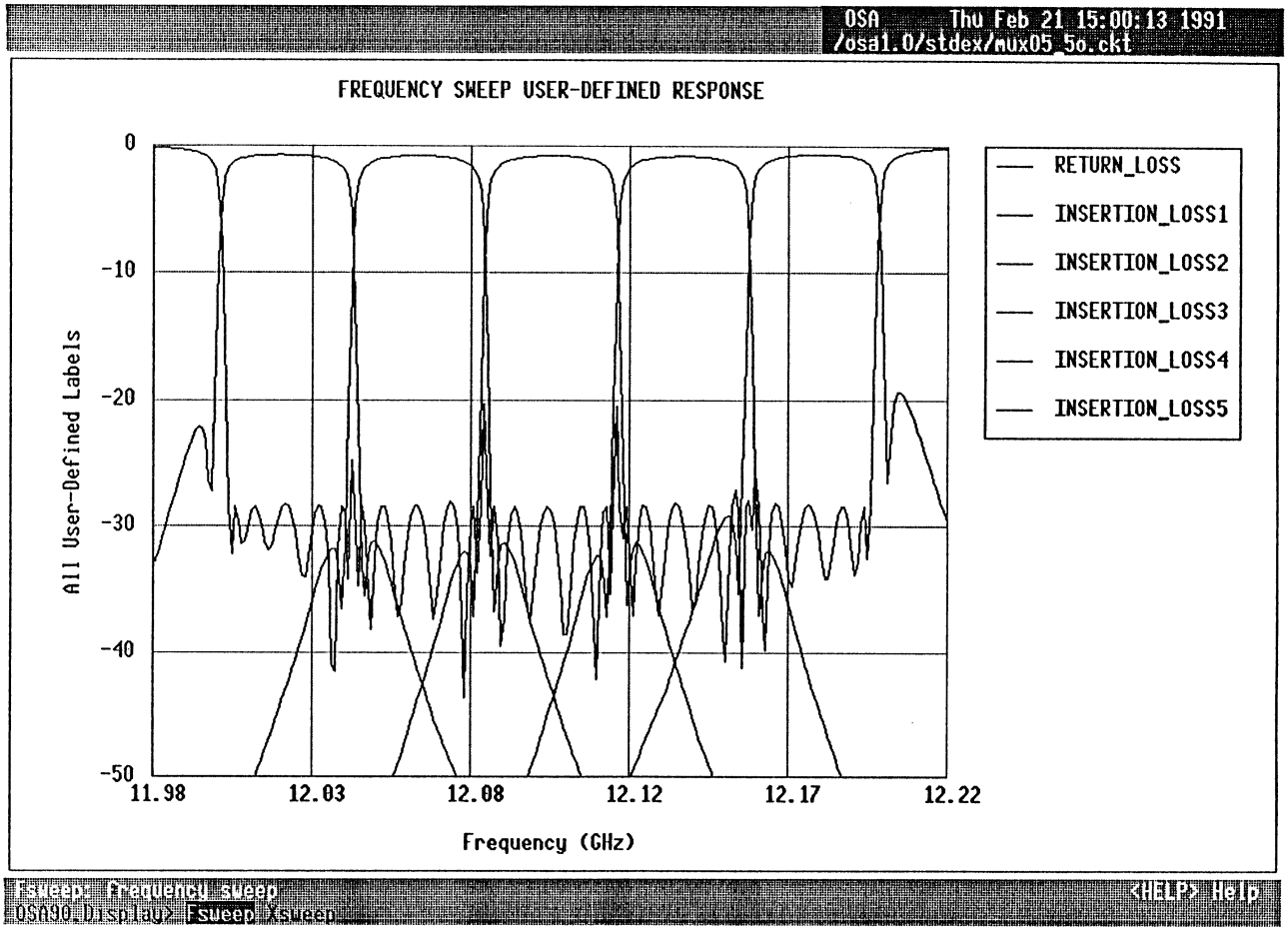
Pipe Server to User's Programs - Simple Interface

the following code may be sufficient in many cases

```
main()
{
  int n, m, error = 0;

  for ( ; ; ) {
    pipe_initialize2();
    pipe_read2(&n, sizeof(int), 1);
    pipe_read2(&m, sizeof(int), 1);
    pipe_read2(x, sizeof(float), n);

    error = simulator(n, m, x, y);

    pipe_write2(&error, sizeof(int), 1);
    if (!error) pipe_write2(y, sizeof(float), m);
  }
}
```

Optimization of 5-Channel Multiplexer

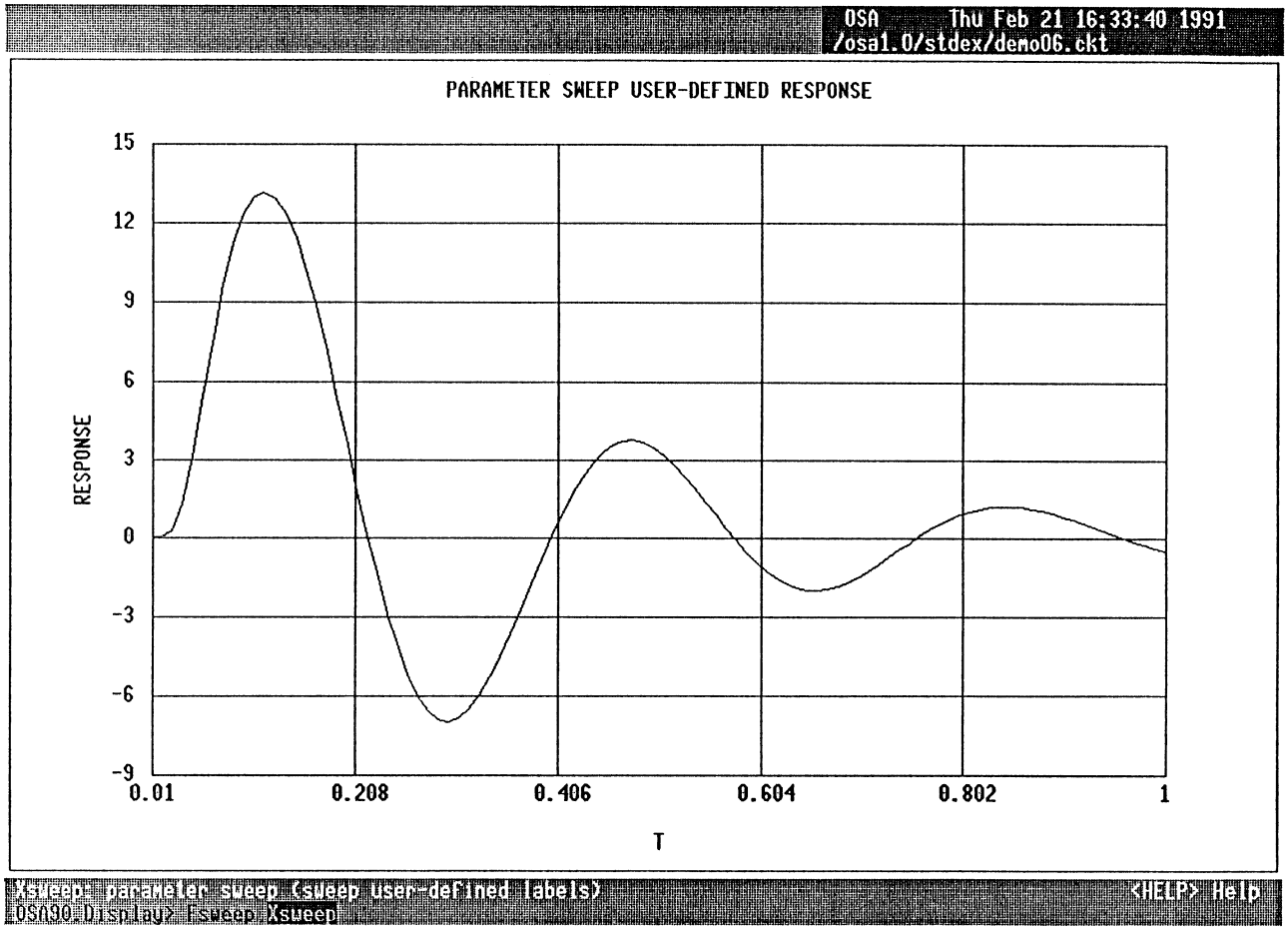
common port return loss and individual channel insertion loss responses after optimization:



OSA90 and a multiplexer simulator interacted for 3724 iterations

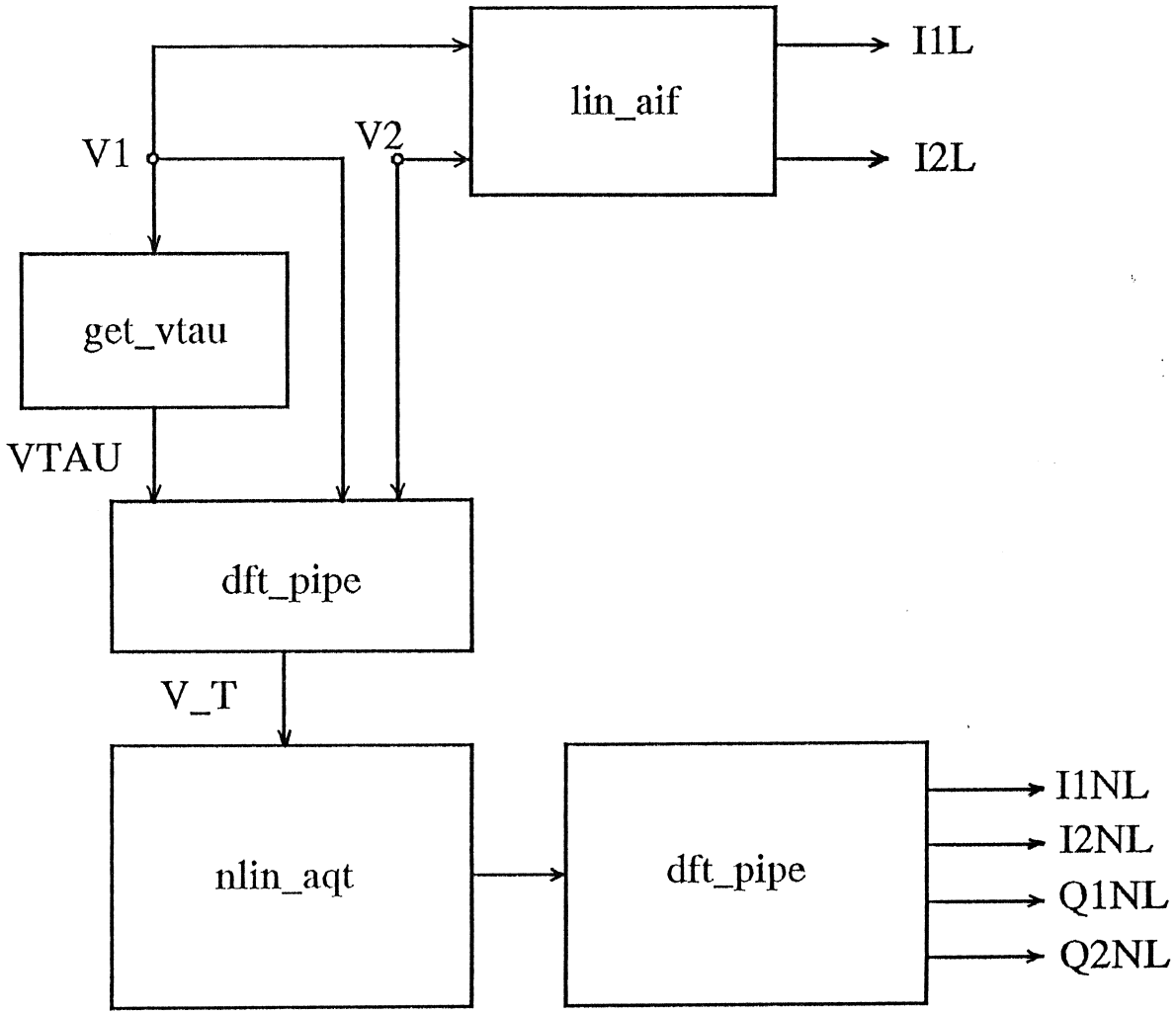
Time-Domain Simulation of a Feedback Network

transient response to a pulse excitation:



OSA90 and a time-domain simulator interacted 100 times

Solving Harmonic Balance with Datapipe™



I1L + I1NL + j * K * OMEGA * Q1NL = 0

I2L + I2NL + j * K * OMEGA * Q2NL = 0

Solving Harmonic Balance with Datapipe™

frequency-domain state variables: V1 V2

user program *lin_aif* for linear subcircuit simulation

inputs: V1 V2
outputs: I1L I2L

user program *get_vtau* computes delayed voltage

inputs: TAU V1
outputs: VTAU

user program *dft_pipe* for DFT (frequency to time)

inputs: V1 V2 VTAU
outputs: V_T

Solving Harmonic Balance with Datapipe™

user program *nlin_aqt* for nonlinear device simulation

inputs: V_T PAR
outputs: I1NL_T I2NL_T Q1NL_T Q2NL_T

call *dft_pipe* again for DFT (time to frequency)

inputs: I1NL_T I2NL_T Q1NL_T Q2NL_T
outputs: I1NL I2NL Q1NL Q2NL

formulate the harmonic balance equation

I1L + I1NL + j * K * OMEGA * Q1NL = 0

I2L + I2NL + j * K * OMEGA * Q2NL = 0

complex vectors are split into real and imaginary parts, e.g., V1 is represented by V1_R and V1_I

Solving Harmonic Balance with Datapipe™

macro definitions (text substitution)

```
#define  FREQ_TO_TIME      1
#define  TIME_TO_FREQ      2
#define  N_HARM            4
#define  N_T               9
#define  N_PORTS           2
#define  N_PARS           17
#define  N_T_WAVEFORM    128
```

macro definitions reusing macros already defined

```
#define  N_VT              (N_PORTS + 1)
#define  N_SPECTRA         (2 * (N_HARM + 1))
#define  N_STATE_FREQ      (N_PORTS * N_SPECTRA)
#define  N_STATE_TIME      (N_VT * N_T)
```

Solving Harmonic Balance with Datapipe™

define variables in Expression block

```
Expression

  OMEGA: 2.0 * PI * FREQ;
```

use arrays for frequency-domain state variables

```
V1_R[0:N_HARM]   = [?-0.673?  ?0.5?  ?-1 0 1?  ?-1 0 1?
                    ?-1 0 1?];
V1_I[0:N_HARM]   = [0  ?-5 0 5?  ?-1 0 1?  ?-1 0 1?  ?-1 0 1?];
V2_R[0:N_HARM]   = [?4?  ?-5 0 5?  ?-1 0 1?  ?-1 0 1?
                    ?-1 0 1?];
V2_I[0:N_HARM]   = [0  ?-5 0 5?  ?-1 0 1?  ?-1 0 1?  ?-1 0 1?];
```

call user program *lin_aif* for linear subcircuit simulation

```
Datapipe: SIM  FILE = "lin_aif"
    N_INPUT         = (N_STATE_FREQ + 3)
    INPUT           = (N_PORTS, N_HARM, FREQ,
                       V1_R, V1_I, V2_R, V2_I)
    N_OUTPUT        = N_STATE_FREQ
    OUTPUT          = (I1L_R[0:N_HARM], I1L_I[0:N_HARM],
                       I2L_R[0:N_HARM], I2L_I[0:N_HARM]);
```

Solving Harmonic Balance with Datapipe™

call user program *get_vtau* to compute delayed voltage

```
TAU: 3PS;
    Datapipe: SIM  FILE = "get_vtau"
    N_INPUT          = (N_SPECTRA + 3)
    INPUT            = (N_HARM, TAU, OMEGA, V1_R, V1_I)
    N_OUTPUT         = N_SPECTRA
    OUTPUT           = (VTAU_R[0:N_HARM],
                        VTAU_I[0:N_HARM]);
```

call user program *dft_pipe* for DFT (frequency to time): returned time samples will be used in the time domain simulation of the nonlinear subcircuit

```
Datapipe: SIM  FILE = "dft_pipe"
    N_INPUT   = (N_STATE_FREQ + N_SPECTRA + 4)
    INPUT     = (FREQ_TO_TIME, N_VT, N_HARM, N_T, V1_R,
                 V1_I, V2_R, V2_I, VTAU_R, VTAU_I)
    N_OUTPUT      = N_STATE_TIME
    OUTPUT    = (V_T[1:N_STATE_TIME]);
```

Solving Harmonic Balance with Datapipe™

call user program *nlin_aqt* for nonlinear device simulation

```
PAR[1:N_PARS] =  [TAU  0.15  0.15  -0.02  -0.02  1  0.03  2
                  1E-14  1  0.545PF  0.092PF  0.77  1.0E-07
                  0.8  30.0  298];

Datapipe: SIM  FILE = "nlin_aqt"
    N_INPUT         = (2 + N_PARS + N_STATE_TIME)
    INPUT           = (N_PARS, N_T, PAR, V_T)
    N_OUTPUT        = (2 * N_PORTS * N_T)
    OUTPUT          = (I1NL_T[1:N_T], I2NL_T[1:N_T],
                       Q1_T[1:N_T], Q2_T[1:N_T]);
```

call *dft_pipe* again for DFT (time to frequency): returned spectra will be used in harmonic balance equations

```
Datapipe: SIM  FILE = "dft_pipe"
    N_INPUT         = (2 * N_PORTS * N_T + 4)
    INPUT           = (TIME_TO_FREQ, (2 * N_PORTS),
                       N_HARM, N_T, I1NL_T, I2NL_T,
                       Q1_T, Q2_T)
    N_OUTPUT        = (2 * N_PORTS * N_SPECTRA)
    OUTPUT    = (I1NL_R[0:N_HARM], I1NL_I[0:N_HARM],
                 I2NL_R[0:N_HARM], I2NL_I[0:N_HARM],
                 Q1NL_R[0:N_HARM], Q1NL_I[0:N_HARM],
                 Q2NL_R[0:N_HARM], Q2NL_I[0:N_HARM]);
```

Solving Harmonic Balance with Datapipe™

formulate harmonic balance equations (residual errors)

```
KS[0:N_HARM]              = [0 1 2 3 4];
K_OMEGA[0:N_HARM]         = KS * OMEGA;

Error1_R[0:N_HARM]  = I1L_R + I1NL_R - K_OMEGA * Q1NL_I;
Error1_I[0:N_HARM]  = I1L_I + I1NL_I + K_OMEGA * Q1NL_R;
Error2_R[0:N_HARM]  = I2L_R + I2NL_R - K_OMEGA * Q2NL_I;
Error2_I[0:N_HARM]  = I2L_I + I2NL_I + K_OMEGA * Q2NL_R;
```

define current magnitude spectrum for output

```
I1_SPECTRUM[0:N_HARM] = sqrt(I1L_R * I1L_R + I1L_I * I1L_I);
I2_SPECTRUM[0:N_HARM] = sqrt(I2L_R * I2L_R + I2L_I * I2L_I);
```

Solving Harmonic Balance with Datapipe™

postprocessing: call *dft_pipe* again (DFT frequency to time) to calculate output waveforms

```
Datapipe: SIM  FILE="dft_pipe"   !
    N_INPUT          = (N_SPECTRA + 4)
    INPUT            = (FREQ_TO_TIME, 1, N_HARM,
                        N_T_WAVEFORM, V1_R, V1_I)
    N_OUTPUT         = N_T_WAVEFORM
    OUTPUT           = (V1_T[1:N_T_WAVEFORM]);

Datapipe: SIM  FILE="dft_pipe"
    N_INPUT          = (N_SPECTRA + 4)
    INPUT            = (FREQ_TO_TIME, 1, N_HARM,
                        N_T_WAVEFORM, V2_R, V2_I)
    N_OUTPUT         = N_T_WAVEFORM
    OUTPUT           = (V2_T[1:N_T_WAVEFORM]);
```

define a convenient sweep variable

```
    K: 1;
end
```

Solving Harmonic Balance with Datapipe™

define simulation ranges and specify responses for both numerical and graphical outputs

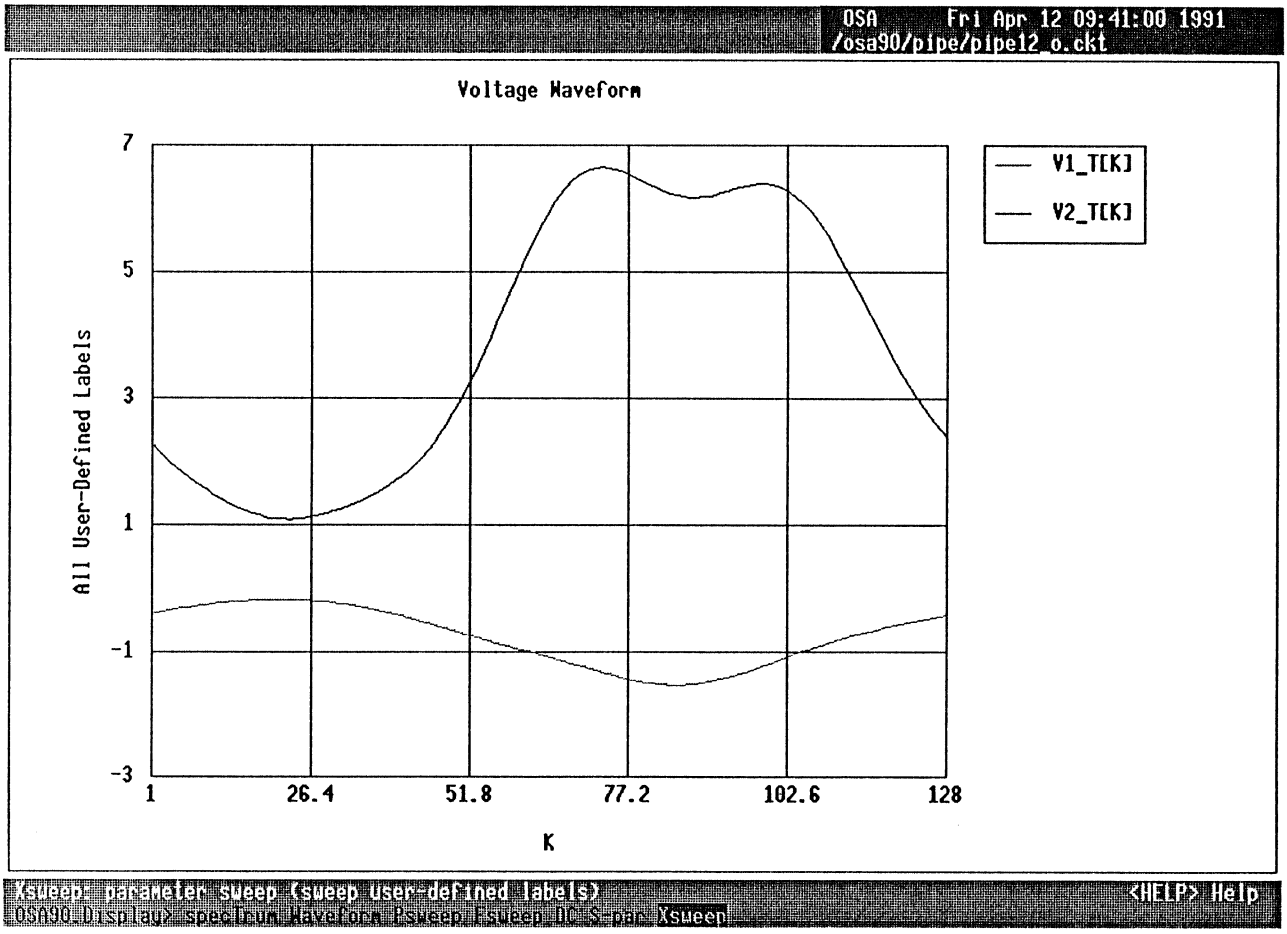
```
Sweep
  FREQ: 6GHZ  K: from 0 to 4 step=1
  Error1_R[K], Error1_I[K], Error2_R[K], Error2_I[K]
  Title = "Harmonic Balance Equation Residual Errors";

  FREQ: 6GHZ  K: from 0 to 4 step=1
  I1_SPECTRUM[K], I2_SPECTRUM[K]
  Title = "Current Spectrum"  Draw=BAR;

  FREQ: 6GHZ  K: from 1 to N_T_WAVEFORM step=1
  V1_T[K]  V2_T[K]  Title = "Voltage Waveform";
end
```

define optimization goals: minimize harmonic balance residual errors

```
Spec
  FREQ: 6GHZ   Error1_R = 0   Error1_I = 0;
  FREQ: 6GHZ   Error2_R = 0   Error2_I = 0;
end
```

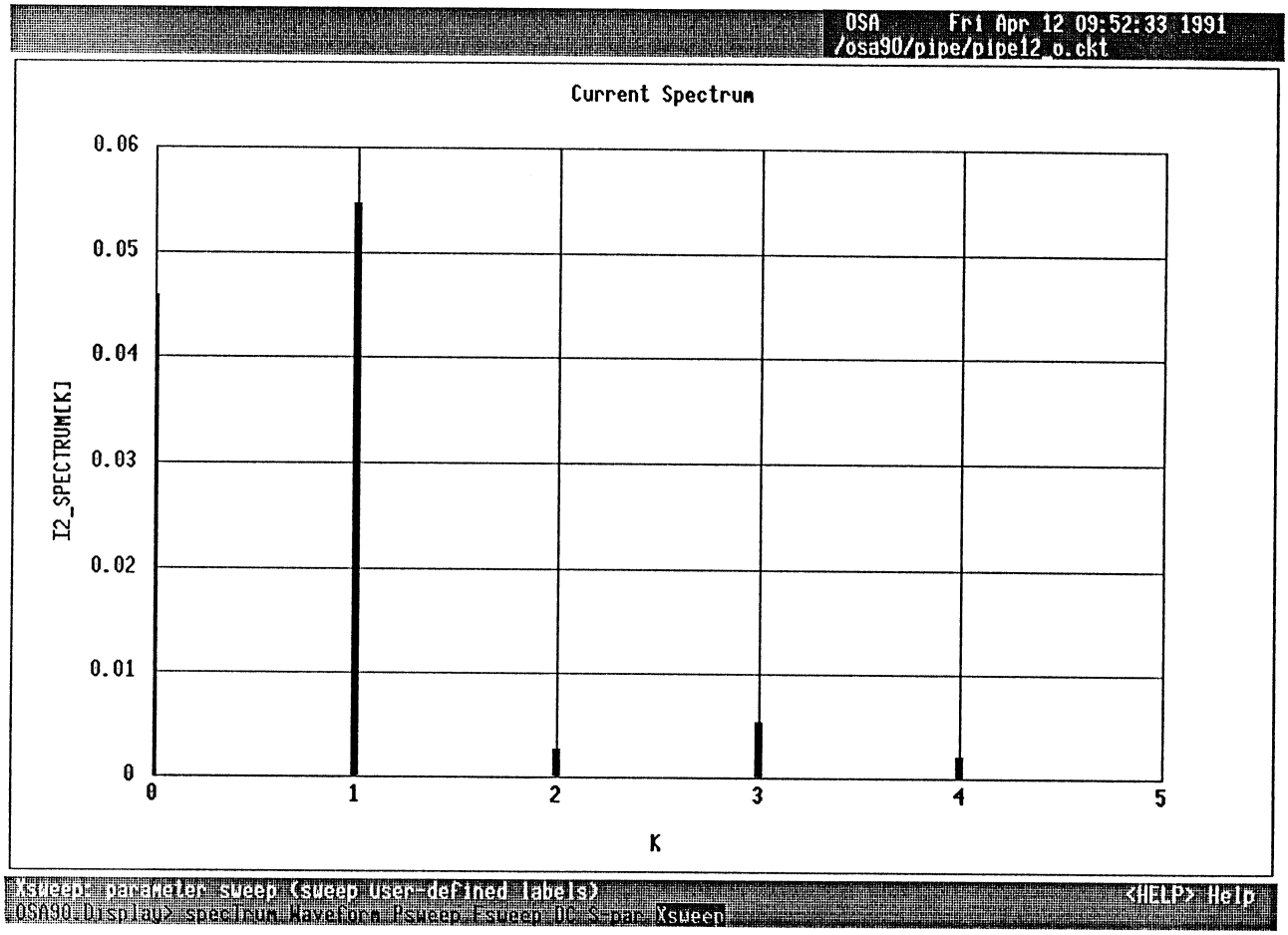
Solving Harmonic Balance with Datapipe™

voltages at port 1 and port 2: waveform responses



Solving Harmonic Balance with Datapipe™

current at port 2: magnitude spectrum



Solving Waveform Balance with Datapipe™

Expression Block modifications: after frequency-domain simulation of linear subcircuit transform linear currents from the frequency domain to the time domain

```
Datapipe: SIM  FILE = "dft_pipe"
    N_INPUT         = (N_STATE_FREQ + 4)
    INPUT           = (FREQ_TO_TIME, N_PORTS, N_HARM, N_T,
                       I1L_R, I1L_I, I2L_R, I2L_I)
    N_OUTPUT        = (N_PORTS * N_T)  OUTPUT=(I1L_T[1:N_T],
                       I2L_T[1:N_T]);
```

after time-domain simulation of the nonlinear device transform nonlinear charge from the time domain to the frequency domain

```
Datapipe: SIM  FILE = "dft_pipe"
    N_INPUT         = (N_PORTS * N_T + 4)
    INPUT           = (TIME_TO_FREQ, N_PORTS, N_HARM, N_T,
                       Q1_T, Q2_T)
    N_OUTPUT        = (N_PORTS * N_SPECTRA)
    OUTPUT          = (Q1_R[0:N_HARM], Q1_I[0:N_HARM],
                       Q2_R[0:N_HARM], Q2_I[0:N_HARM]);
```

Solving Waveform Balance with Datapipe™

perform charge differentiation in the frequency domain to get nonlinear capacitor currents

```
KS[0:N_HARM]              =  [0 1 2 3 4];
K_OMEGA[0:N_HARM]         =  KS * OMEGA;

I1C_R[0:N_HARM]           = -K_OMEGA * Q1_I;
I1C_I[0:N_HARM]           =  K_OMEGA * Q1_R;
I2C_R[0:N_HARM]           = -K_OMEGA * Q2_I;
I2C_I[0:N_HARM]           =  K_OMEGA * Q2_R;
```

transform capacitor currents from the frequency domain to the time domain

```
Datapipe: SIM  FILE = "dft_pipe"
    N_INPUT         = (N_PORTS * N_SPECTRA + 4)
    INPUT           = (FREQ_TO_TIME, N_PORTS, N_HARM, N_T,
                       I1C_R, I1C_I, I2C_R, I2C_I)
    N_OUTPUT        = (N_PORTS * N_T)  OUTPUT=(I1C_T[1:N_T],
                       I2C_T[1:N_T]);
```

Solving Waveform Balance with Datapipe™

formulate errors in the time domain

```
Error1_T[1:N_T] = I1L_T + I1NL_T + I1C_T;
Error2_T[1:N_T] = I2L_T + I2NL_T + I2C_T;
```

Sweep Block: specify residual errors for output in time domain

```
FREQ: 6GHZ  K: from 1 to N_T step=1
Error1_T[K], Error2_T[K]
Title = "Waveform Balance Equation Residual Errors";
```

Specification Block: specify optimization goals in the time domain

```
FREQ: 6GHZ  Error1_T=0;
FREQ: 6GHZ  Error2_T=0;
```

Existing User's Routines

Discrete Fourier Transform (DFT) - frequency to time

```
int dft_ft ( nh, nt, xf_r, xf_i, xt )
int nh, nt;
float *xf_r, *xf_i, *xt;
{
  float tmp1, tmp2;
  int j, k;

  tmp1 = 8.0 * atan(1.0) / nt;

  for (j = 1; j <= nt; j++, xt++) {
    *xt = *xf_r;  /* DC */

    for (k = 1; k <= nh; k++) {
      tmp2 = tmp1 * k * j;
      *xt += (cos(tmp2) * xf_r[k] - sin(tmp2) * xf_i[k]);
    }
  }

  return(0);
}
```

Existing User's Routines

Discrete Fourier Transform (DFT) - time to frequency

```
int dft_tf ( nh, nt, xf_r, xf_i, xt )
int nh, nt;
float *xf_r, *xf_i, *xt;
{
  float rnt, tmp1, tmp2;
  int j, k;

  rnt = (float) nt;
  tmp1 = 8.0 * atan(1.0) / rnt;

  for (j = 0; j < nt; j++) xt[j] /= rnt;
  for (k = 0; k <= nh; k++, xf_r++, xf_i++) {
    for (*xf_r = *xf_i = 0.0, j = 0; j < nt; j++) {
      tmp2 = tmp1 * k * (j + 1);
      *xf_r += cos(tmp2) * xt[j];
      *xf_i -= sin(tmp2) * xt[j];
    }
    if (k) {
      *xf_r *= 2.0;
      *xf_i *= 2.0;
    }
  }

  return(0);
}
```

Pipe Server to User's Programs

define:

```
#include <stdio.h>
#include <math.h>
#include "ippcv2.h"

#define  FREQ_TO_TIME  1
#define  TIME_TO_FREQ  2

#define  NXMAX          8
#define  NHMAX          8
#define  NHMAX1         9
#define  NTMAX        512

#define  NMAX        4096   /* NXMAX * NTMAX */

float x[NMAX], y[NMAX];
```

Pipe Server to User's Programs

write *main* program to communicate through pipes

```
main ()
{
  float *xt, *xf_r, *xf_i;
  int action, i, n, m, nn, nh, nh1, nt, error = 0;
```

infinite loop for pipe read/write

```
  for (; ;) {
    pipe_initialize2();
    pipe_read2(&n, sizeof(int), 1);
    pipe_read2(&m, sizeof(int), 1);
    pipe_read2(x, sizeof(float), n);

    action = x[0];
    nn = x[1];
    nh = x[2];
    nh1 = nh + 1;
    nt = x[3];
```

Pipe Server to User's Programs

call user's routines as requested by OSA90

```
if (nn > NXMAX || nh > NHMAX || nt > NTMAX) error = 1;
else if (action == FREQ_TO_TIME) {
  if (n != (2 * nn * nh1 + 4) || m != (nn * nt)) error = 1;
  else for (xt = y, xf_r = x + 4, i = 0; i < nn; i++) {
    xf_i = xf_r + nh1;
    dft_ft(nh, nt, xf_r, xf_i, xt);

    xf_r = xf_i + nh1;
    xt += nt;
  }
}
else if (action == TIME_TO_FREQ) {
  if (n != (nn * nt + 4) || m != (2 * nn * nh1) || nt != (2 * nh + 1))
    error = 1;
  else for (xt = x + 4, xf_r = y, i = 0; i < nn; i++) {
    xf_i = xf_r + nh1;
    dft_tf(nh, nt, xf_r, xf_i, xt);

    xf_r = xf_i + nh1;
    xt += nt;
  }
}
else error = 2;
```

Pipe Server to User's Programs

return data to OSA90

```
    pipe_write2(&error, sizeof(int), 1);
    if (!error) pipe_write2(y, sizeof(float), m);
  }
}
```

FUN Type of Datapipe™ Connection

the user's program provides directly the error functions for the optimizer; OSA90 generates gradients

syntax:

```
Datapipe:  FUN  FILE = filename     NAME = id_name
                N_INPUT = n         INPUT = (x1, ..., xn)
                N_OUTPUT = m;
```

the user specifies:

filename	-	the name of executable user's program
n	-	the number of numerical values to be passed to the user's program
x1, x2,..	-	individual labels and/or values to be passed to the user's program
m	-	the number of error values that will be returned from the user's program
id_name	-	name for reference elsewhere in the input file

FDF Type of Datapipe™ Connection

the user's program provides directly for the optimizer both the error functions and gradients

the only difference w.r.t. the FUN type is that $m(n + 1)$ values will be returned from the user's program, i.e., for each of m error functions the error value and n partial derivatives will be returned

syntax:

```
Datapipe:  FDF  FILE = filename     NAME = id_name
                N_INPUT = n         INPUT = (x1, ..., xn)
                N_OUTPUT = m;
```

Predefined Labels in OSA90

PI	constant 3.141592654
FREQ	frequency, time or any other variable

Mathematical Functions Supported by OSA90

EXP,	LOG,	LOG10,	SQRT,	ABS,
SIN,	COS,	TAN,	ASIN,	ACOS,
ATAN,	SINH,	COSH,	TANH	

Conditional Expression

if (*condition*) (*expression1*)
[else (*expression2*)];

where *condition* can take the forms

expression3 > *expression4*
expression3 = *expression4*
expression3 < *expression4*

Monte Carlo Analysis

variables which are subject to statistical variations are identified in the input file

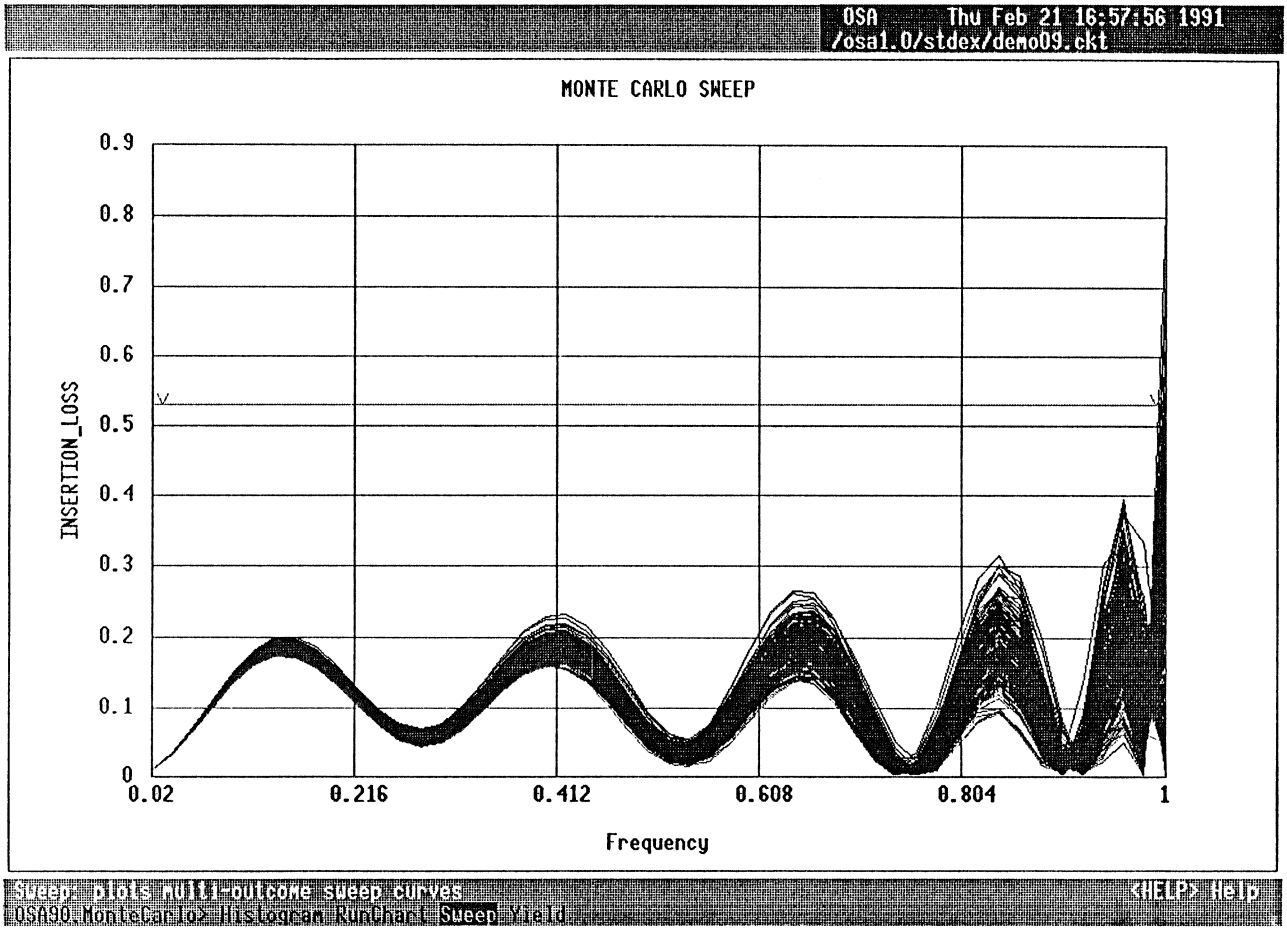
uniform, normal, exponential and lognormal distributions are available; variables can be correlated

OSA90 generates the specified number of statistical outcomes, calls the user's functions, checks design specifications, and evaluates yield

after Monte Carlo analysis is completed the user can examine histograms, run charts and sweep responses

Monte Carlo Analysis of 11-Element LC Filter

statistical insertion loss response:



200 statistical circuit outcomes

OSA90 and a filter simulator interacted 10,000 times

OSA90 Based Consulting Services

OSA is available to create specific features you would like to have in your copy of OSA90

this can include existing OSA technology modules or some other software solutions to be developed to your specifications

Platforms and Availability

OSA90 is available on Apollo, Hewlett-Packard and Sun workstations