Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

SIGNAL CONSTELLATION DESIGN TOOL: A CASE STUDY IN USER INTERFACE SYNTHESIS

Gabriel Robins

July 1988
CSD-880051

Signal Constellation Design Tool: A Case study in User Interface Synthesis

Gabriel Robins

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024, USA
gabriel@vaxb.isi.edu

Abstract

Signal constellation design is a major subtask of constructing an efficient communication system; it essentially entails trading-off error frequency against information throughput, a chief occupation of modem designers. We propose and implement an interactive tool for designing and simulating arbitrary signal constellations. To construct the user interface we have utilized Interface Builder, a new interactive tool that greatly facilitates the synthesis of arbitrary user interfaces through an object-oriented methodology. Using the Interface Builder package and the Signal Constellation Design Tool as the target prototype, we show how an order-of-magnitude improvement can be achieved in the effort required to produce a complex user interface. Our secondary goal is to try to dispel some of the mystique surrounding user interface synthesis on state-of-the-art workstations by describing in detail the construction of an interactive tool for computer-assisted learning.

Keywords: User interfaces, User interface tools, Human computer instruction, Man-machine interaction, Computer-assisted learning, Simulation tools, Object-oriented systems.

1. Introduction

Signal constellation design is a major part of constructing an efficient communication system. This task essentially entails trading off error frequency against information throughput, a chief occupation of modem designers. We propose and implement an interactive tool for designing and simulating arbitrary signal constellations. While the actual code that simulates signal constellations is rather trivial in itself, the user interface to this code is quite complex. To design and construct this user interface we have used Interface Builder, a new interactive tool that greatly facilitates the synthesis of user interfaces through an object-oriented methodology. Using the Interface Builder package and the Signal Constellation Design Tool as the target prototype, we show how an order-of-magnitude improvement can be achieved in the effort required to produce a complex user interface, and then draw some conclusions regarding the synthesis of user interfaces in general.

Our secondary goal is to try to dispel some of the mystique surrounding user interface synthesis on state-of-the-art workstations. Many otherwise informed researchers have very little experience in user-interface design, and consequently view user interface design as some sort of a black art, best left to specialized hackers to dabble in. By user interface design I mean a collection of functionality (running on a bit-mapped display workstation with a mouse) that interacts with the user in a friendly manner via menus, scroll bars, control buttons, icons, mouse clicks, and key strokes.

We intend to show that, quite to the contrary of these myths, given the proper tools and methodology, the synthesis of complex user interfaces could be rather trivial. As a case in point, the user interface described in this document was implemented on a MacIntosh, requiring only several days of coding, including the time to read the manuals and learn how to use the software. As a by-product of our inquiry, we have synthesized an interactive tool for computer-assisted learning.

The first half of this document explains signal constellation design in general and how Interface Builder was used to synthesis the user interface; numerous examples and illustrations are given. The rest of this document describes and illustrates the functionality and usage of the resulting signal constellation design tool. The annotated Common LISP source code is available upon request both in hardcopy and on a MacIntosh diskette.

2. Signal Constellation Design

In designing an efficient communication scheme for band-limited channels, invariably of chief concern are the effects of noise and other kind of interference on the system [Forney, Gallager, Lang, Longstaff, and Qureshi]. To combat such interference, and while still aiming to achieve high throughput, one must carefully design an appropriate signal constellation [Carlyle] [Schwartz] [Sklar].

The task of signal constellation design essentially entails trading off error frequency against information throughput and is a chief occupation of modem designers. We propose and implement an interactive tool to alleviate the task of designing and simulating arbitrary signal constellations. We would like our tool to graphically display the signal constellation in two dimensions, allowing the user to visually observe the progressing simulation under interactive modifications to the interference parameters of the system.

2.1. The Desired Functionality

In this section we describe in more detail the functionality that we would like our Signal Constellation Design Tool to exhibit. Later we explain how this functionality was actually achieved in the implementation.

First, we would like to allow the user to select any of a number of "canned" standard signal constellations. For example, the user may elect to simulate an N-in-a-circle signal constellation and observe its performance under various levels of noise and distortion. Such selections should be done via mouse and menu interaction. Next the user may wish to select a certain probability distribution that would control the generation of random signal points. For example, the user may wish to select a Gaussian distribution with a specified variance.

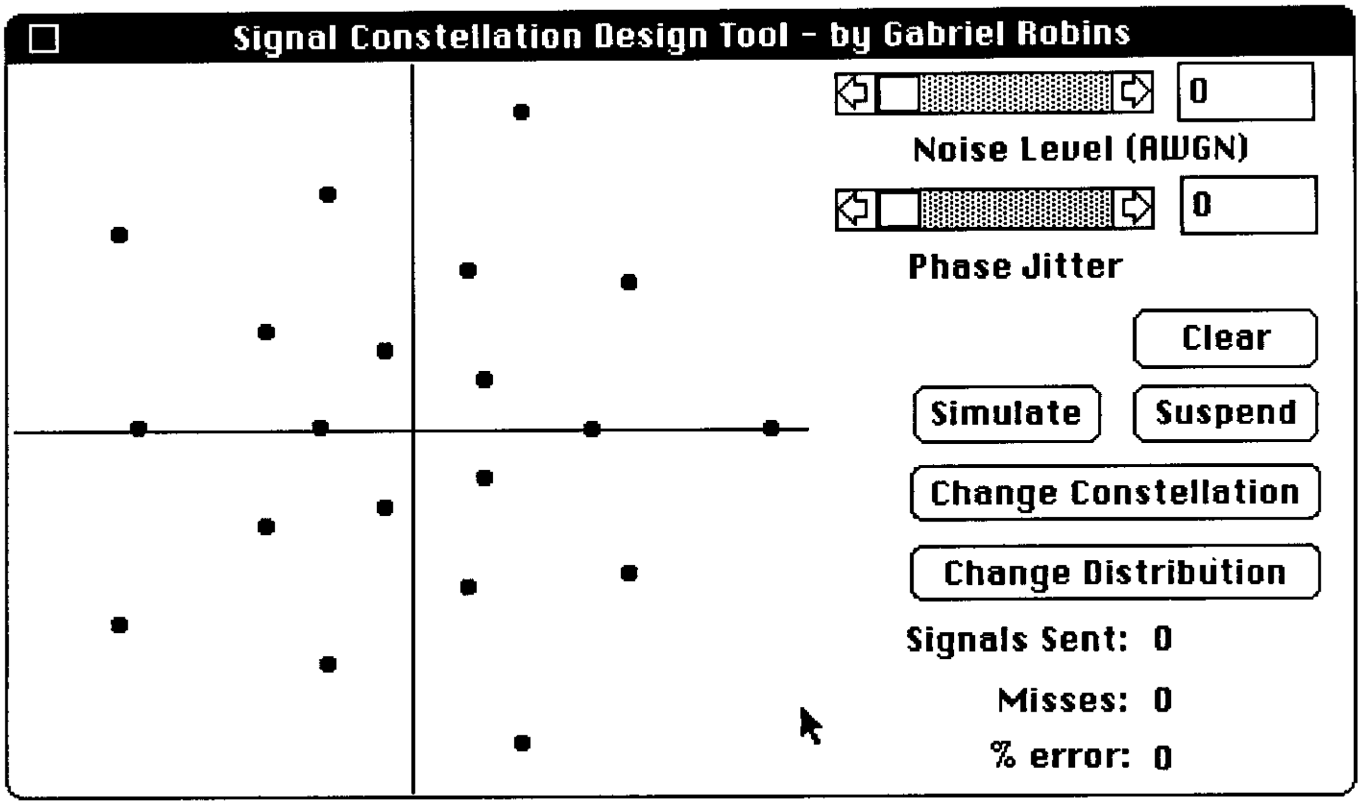
Once the user has selected a particular constellation to simulate/observe, as well as a probability distribution, that constellation should be drawn on the screen and the simulation may proceed. During the simulation, the user may interactively modify a number of system parameters, such as the phase jitter and the additive white Gaussian noise level. This would be accomplished by dragging "scroll-bars" identified with the corresponding parameters, or by directly typing in the desired values.

Using a random number generator, random signals are generated, according to the probability distribution function specified earlier, and are plotted on the signal constellation diagram. After a few minutes, a cumulative scatter-plot of the received signals will become apparent, giving the user an indication of how that signal constellation is performing under the distortion parameter values set previously. A cumulative running total of the number of errors encountered so far should be displayed, as is the empirically derived error-probability (the number of errors divided by the number of signals transmitted.)

The various commands should be also be accessible via clicking appropriate buttons, and alternatively also via menus and keystrokes. In addition, we wish to provide the user with some on-line help and information.

2.2. The Main Panel

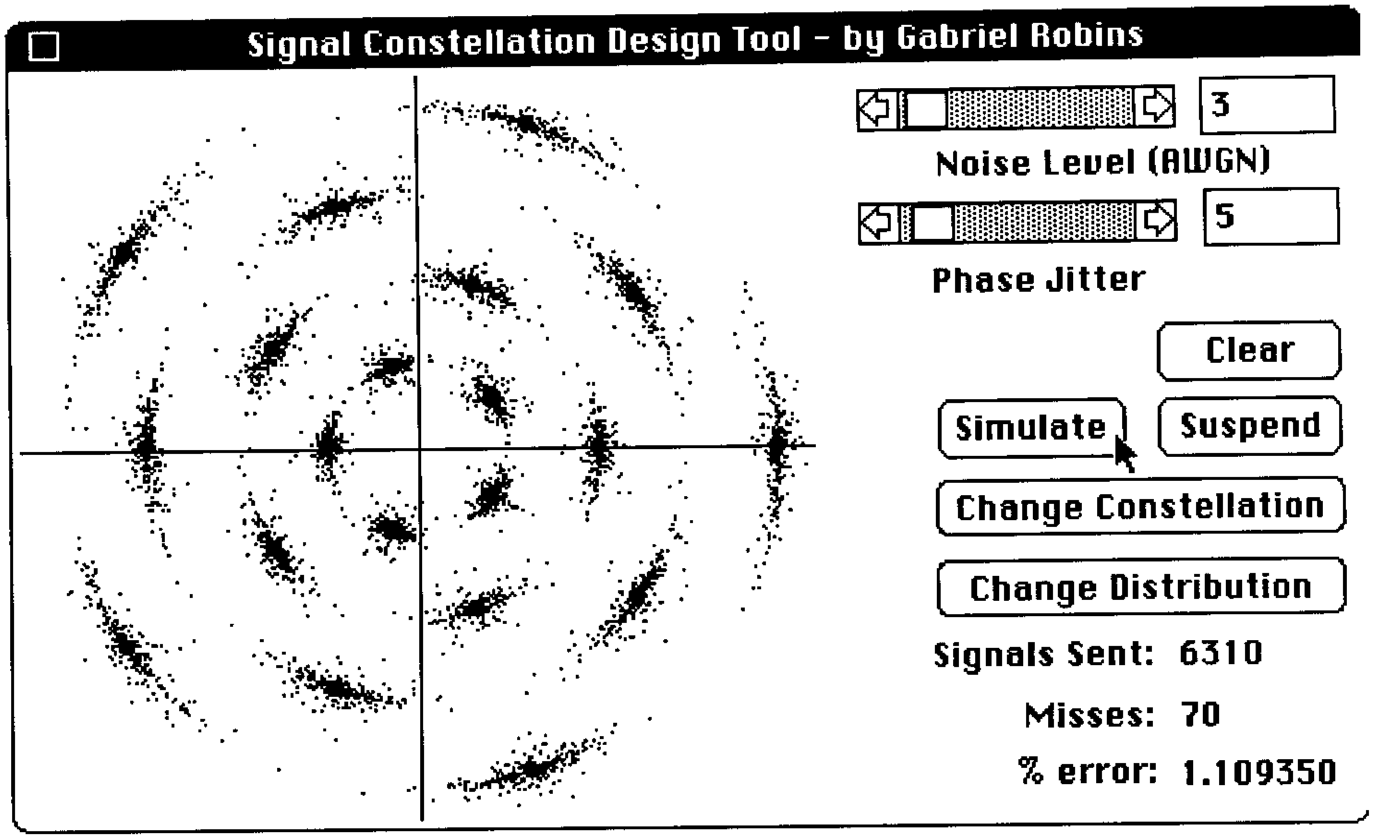
To make the appearance of the user interface more concrete, we give an illustration of how the main panel might appear:



To the left we see the main drawing area where the signal constellation appears; in this case the signal constellation itself consists of 20 points uniformly distributed on 4 concentric circles. At the top right we note the interference parameters, as well as the scroll bars and click boxes used to modify them. Below that we observe several "buttons" each of which will invoke a command if the user clicks it with the mouse. To the lower right we have the running statistics and error-ratio as the simulation progresses.

The user may invoke several operations simply by clicking the corresponding buttons. In addition, all of these commands are also available from the pull-down menus, as well as through keystrokes (i.e. single character keyboard inputs). We may also have at the top a pull-down menu bar, representing the various commands the user may invoke; the menu bar is not visible in this diagram.

After a simulation has been underway for some time, the main panel might appear as follows:

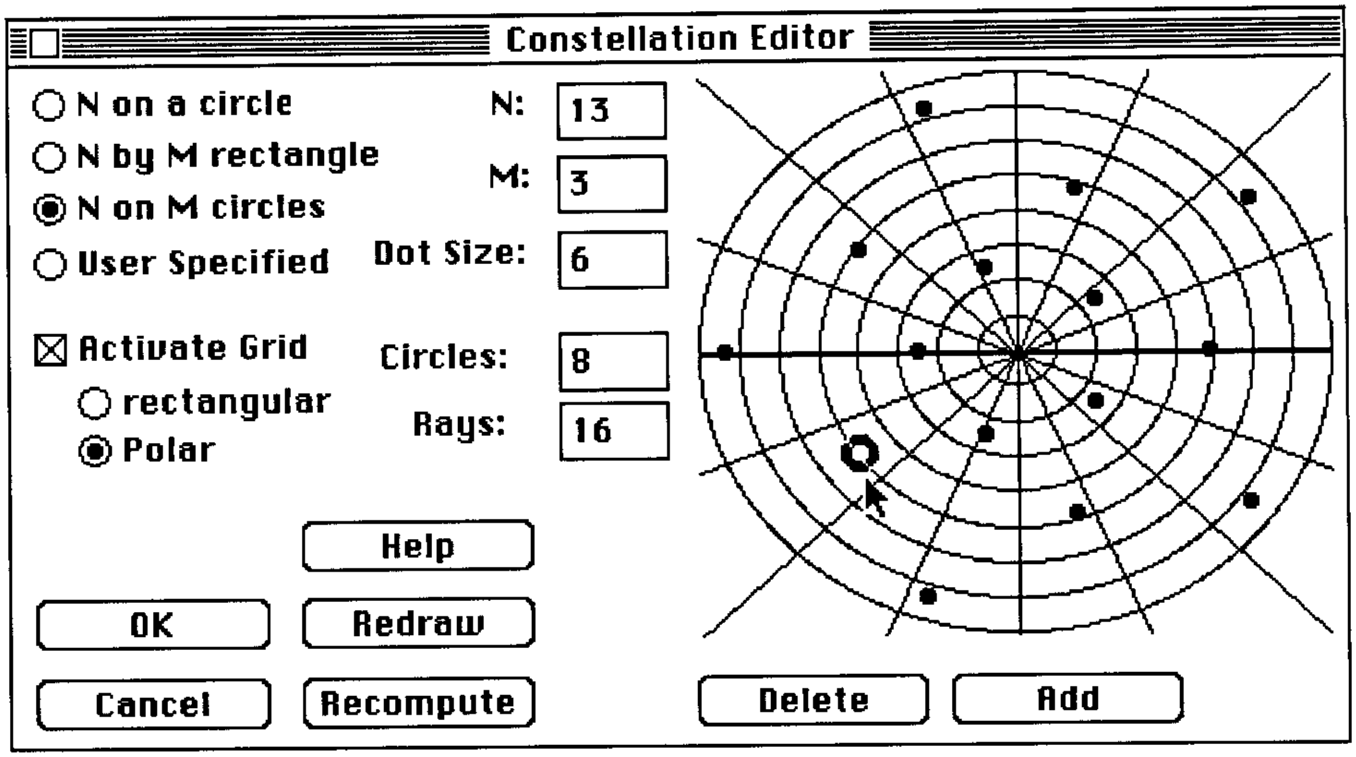


Noise Level (AWGN): 3
Phase Jitter: 5
Signals Sent: 6310
Misses: 70
% error: 1.109350

The clouds around the signal points represent where the randomly generated signals fell around the actual signal constellation points. In this simulation, given the specified noise parameters, we are observing an error rate over over one percent, an undesirable situation.

2.3. The Constellation Editor

The panel that allows the user to select and edit a signal constellation is called the Constellation Editor and may appear as follows:



N on a circle
N by M rectangle
[✓ N on M circles]
User Specified

N: 13
M: 3
Dot Size: 6

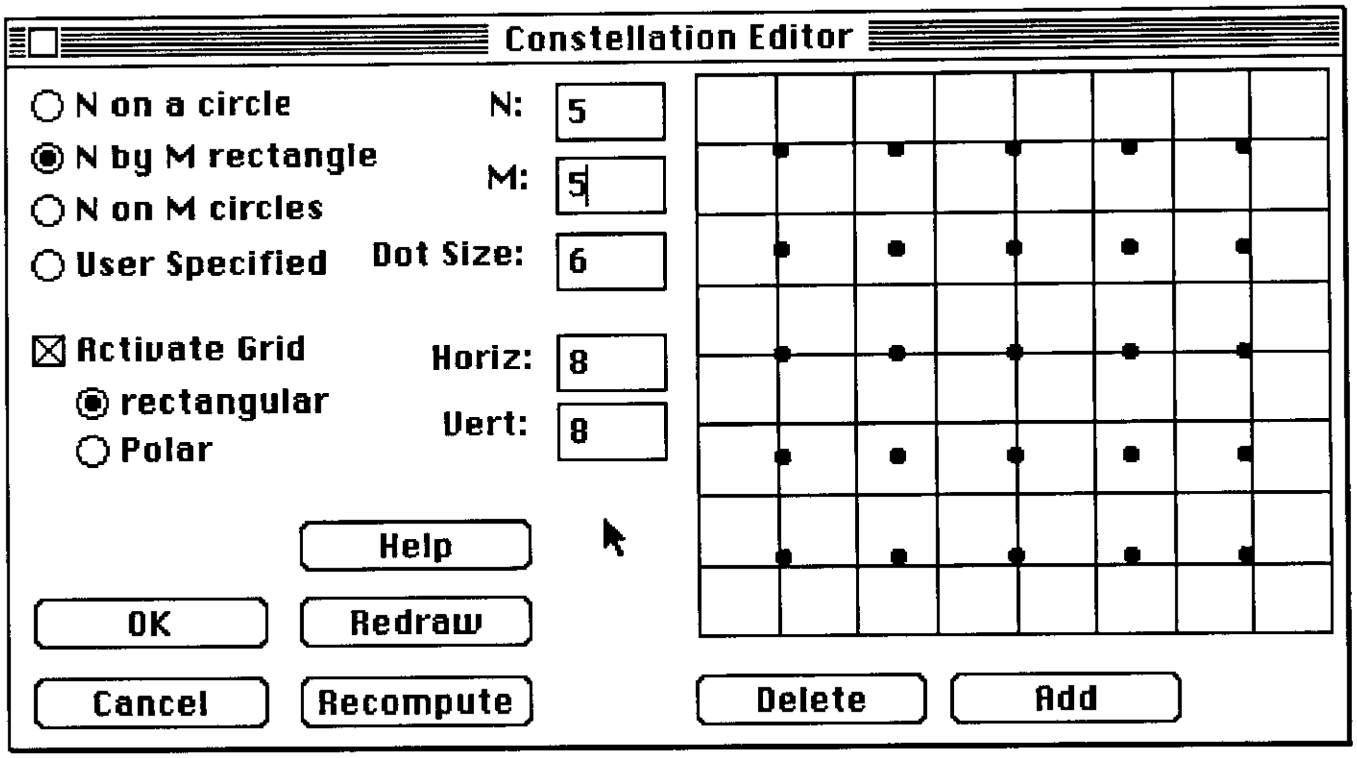
[✓ Activate Grid]
rectangular
[✓ Polar]

Circles: 8
Rays: 16

At the top left the user may select one of several "canned" signal constellations, parametrized by the variables M and N; these parameters are also user-specified: to change them, the user simply clicks in the corresponding box and types in the new value. An optional editing grid is available, and may be either rectangular or polar; the purpose of this grid is to make placement of individual constellation points more precise. The resolution of the grid may be controlled by the user; in the case of the rectangular grid, the number horizontal and vertical lines may be specified, and in the case of the polar grid, the number of circles and rays may be specified.

The user may add or delete constellation points, redraw the display, or obtain on-line help, simply by clicking the corresponding buttons. In addition, all of these commands are also available from the pull-down menus, as well as through keystrokes (i.e. single character keyboard inputs). Note that one of the points of the constellation is highlighted; this is accomplished when the mouse is clicked anywhere in the drawing area, whereupon the closest point to the click becomes highlighted. A "delete" command would subsequently remove the highlighted point, while an "add" command would wait for a new mouse click and a new point would be added to the constellation at the location of that click.

The on-line help consists of several screens of information and will be discussed later. "Ok" saves the current signal constellation and uses it from now on in all future calculations, while "Cancel" reverts back to the signal constellation previously in effect. Had the user selected a rectangular grid instead on a polar grid, the display might have appeared as follows:


Constellation Editor

( ) N on a circle
(●) N by M rectangle
( ) N on M circles
( ) User Specified

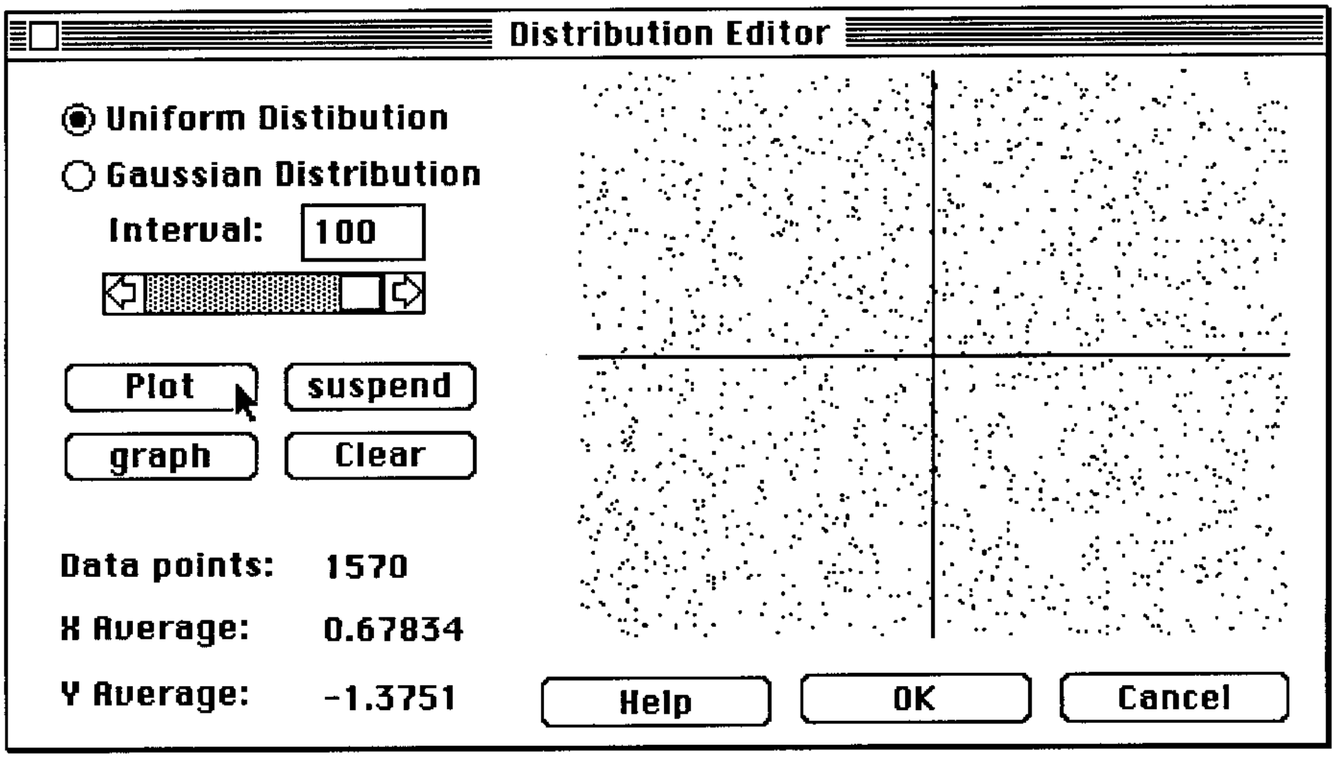
N: 5
M: 5
Dot Size: 6

☒ Activate Grid
(●) rectangular
( ) Polar

Horiz: 8
Vert: 8

2.4. The Distribution Editor

The panel that allows the user to select and edit a signal constellation is called the Distribution Editor and may appear as follows:


Distribution Editor

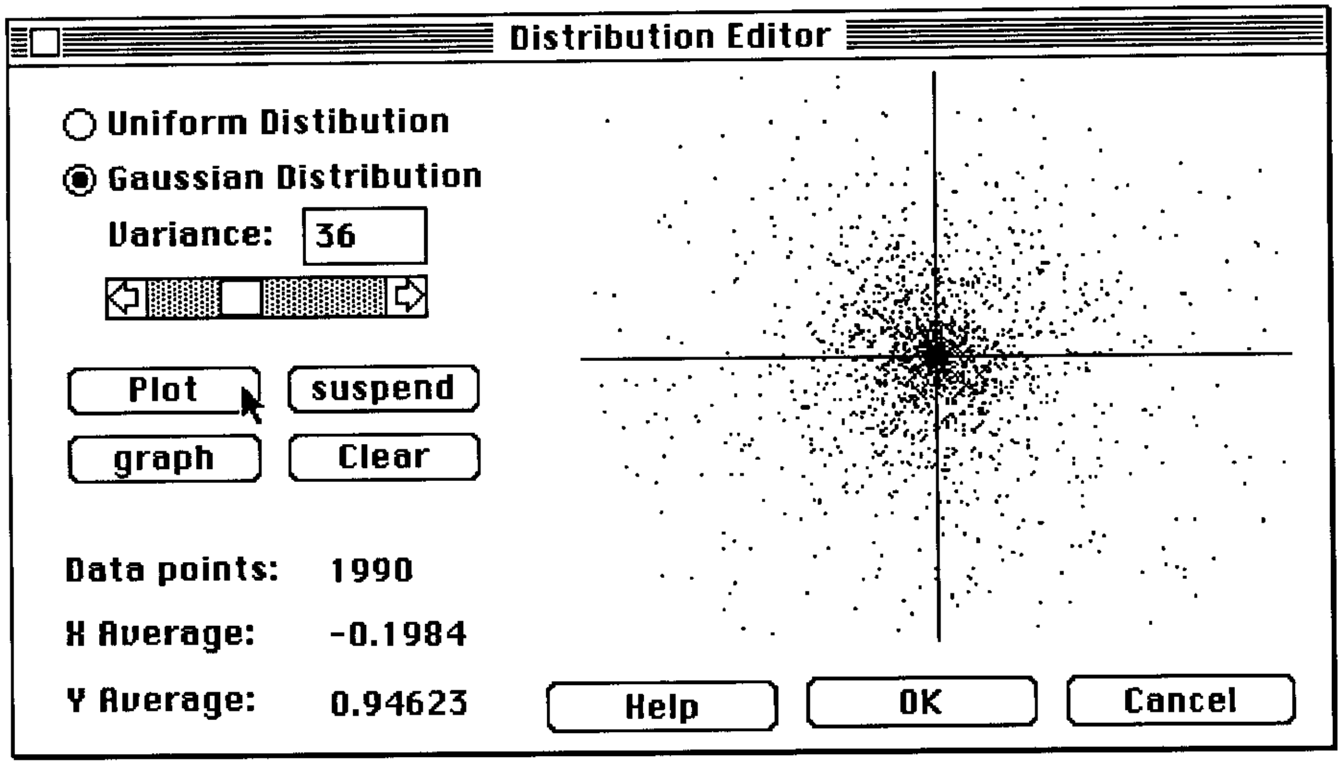
(●) Uniform Distibution
( ) Gaussian Distribution

Interval: 100

Data points: 1570
X Average: 0.67834
Y Average: -1.3751

The user may select from either a uniform distribution on a given interval, or a Gaussian distribution with a given variance. The "Plot" command starts generating and plotting random points according to the distribution specified by the user. The "Graph" command draws a graph of the probability density function in the X/Y plane. The "Help" command provides some on-line help/information, while the "Clear" command clears all the old points from the display. The average X and Y coordinates for the points generated so far are displayed to the lower left. "Ok" saves the current probability distribution and uses it from now on in all future calculations, while "Cancel" reverts back to the probability distribution previously in effect.

The following diagram illustrates a "Plot" of the Gaussian distribution:



Variance: 36

Data points: 1990

X Average: -0.1984

Y Average: 0.94623

By now the reader would agree that although simulating a given signal constellation may by itself indeed constitute a trivial programming task, the construction of a user interface that would behave as described above is by contrast quite a formidable programming task. In practical terms, the former could be easily accomplished in a couple of hours, while the later may take many weeks to construct. Using Interface Builder and an object-oriented programming methodology, all of these tasks were implemented on a MacIntosh in only several days of coding, including the time to read the manuals and learn how to use the software.

3. Using Interface Builder

The process of constructing the user interface using Interface Builder simply entails specifying inside an interactive environment the various menus, dialogue-boxes, scroll-bars, and menu-buttons, as well as where they should appear on the screen, and what should happened when each is clicked, selected, or dragged. The latter is accomplished by providing the relevant LISP code associated with each object. ExperInterface Builder performs all of the user-specified functions at the right times by usurping the workstation's "main event loop" and substituting the user-specified functionality as the default.

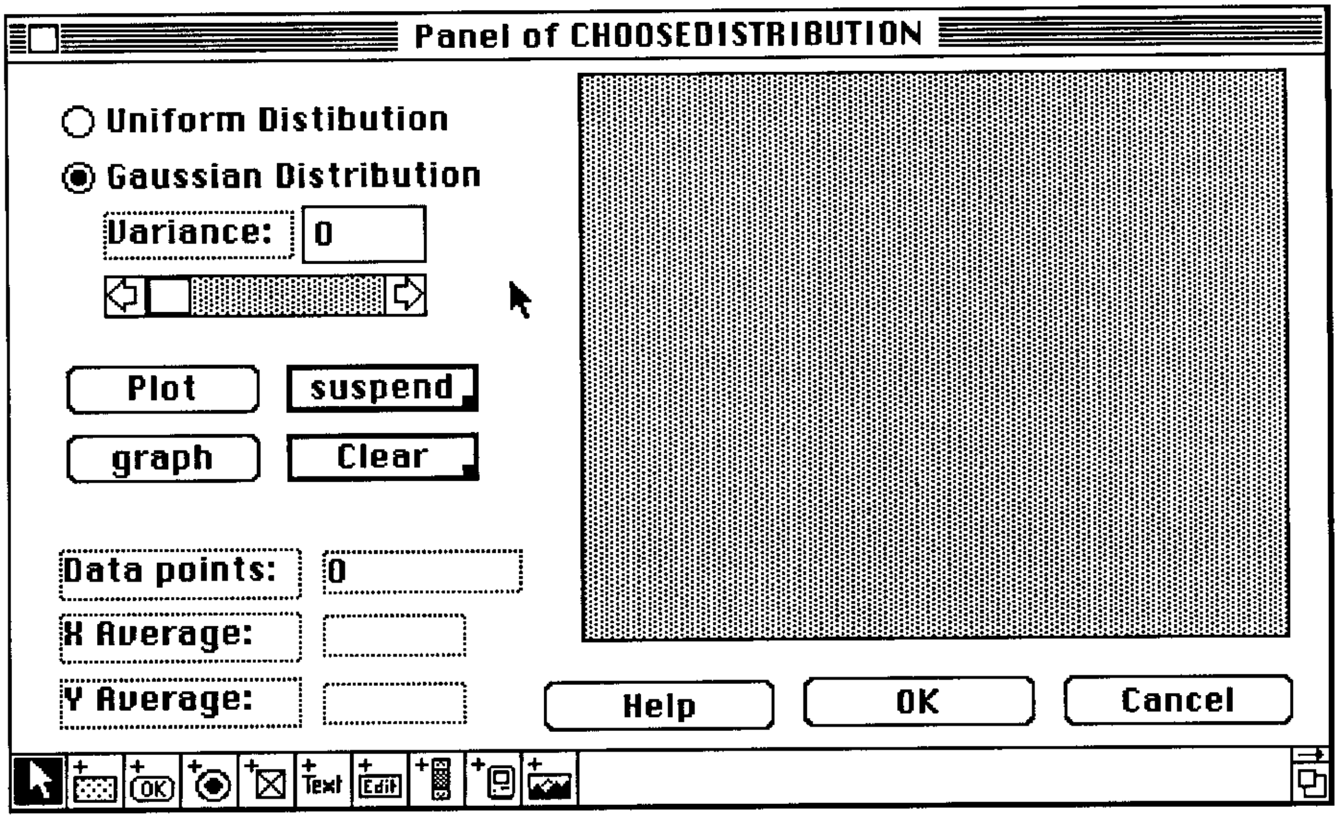
The result is an attractive user interface which is easy to build and modify. This tool was pioneered by ExperTelligence and recently has met with competition from Apple Inc., namely the HyperCard program [Goodman]. The importance of such tools has only recently been fully appreciated, although it has been known for quite some time that most of the effort associated with constructing computer software is invariably spent in programming the user interface; moreover, in many cases the user interface directly determines the utility of a piece of software [Kaczmarek] [Robins].

3.1. The Methodology of Interface Builder

Interface Builder uses an object-oriented paradigm to create a user interface. Objects are rather general entities and may include windows, bitmaps, icons, records, scroll bars, buttons, text strings, regions, points, lines, files, and mouse clicks, among others. Objects communicate by sending *messages* to one another, and each object has a set of messages that it knows how to respond to; for example, a "redraw" message sent to an icon may cause the icon to redraw itself on the display. In addition to various useful default messages (or *methods*), a user may specify additional customized methods to be associated with an object. Messages may contain zero or more arguments and are essentially equivalent to function calls.

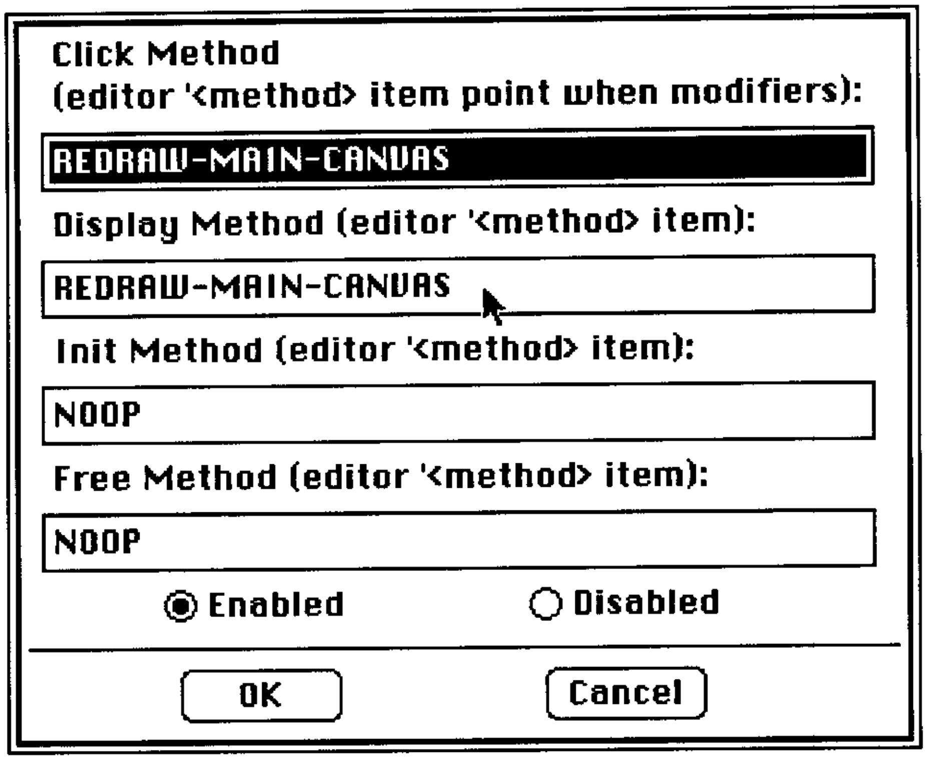
An Interface Builder *editor* is simply a panel consisting of a collection of objects, each with an associated set of methods. In addition to methods, an object may also have some local variables that may store arbitrary values, including other objects. When an object is defined it is specified as a child of some other object, and thus automatically inherits all the methods that apply to its parent; in addition, new methods may be added to the child, specializing it from its parent. An object may have multiple parents, in which case it inherits all of their methods. The astute reader will note that this schema necessitates a conflict-resolution or priority scheme when methods clash through inheritance, but we do not consider these details particularly relevant and therefore do not pursue them any further here.

As a concrete example, let us consider the Distribution Editor described earlier, whose panel inside Interface Builder appears as follows:



Each visible item is an object to which we may send various messages, and with which there is associated functionality that is invoked whenever during execution it is clicked, dragged, resized, etc. The icons at the lower left side are Interface Builder commands and are used to create the various types of objects that they represent pictorially. Once such an object is created, it may be further modified, resized, and redefined.

For example, the "Click" method of the main drawing area to the right may be specified by double-clicking on the main drawing area and filling in the required fields in the resulting dialogue panel as follows:



Click Method
(editor '<method> item point when modifiers):

REDRAW-MAIN-CANVAS

Display Method (editor '<method> item):

REDRAW-MAIN-CANVAS

Init Method (editor '<method> item):

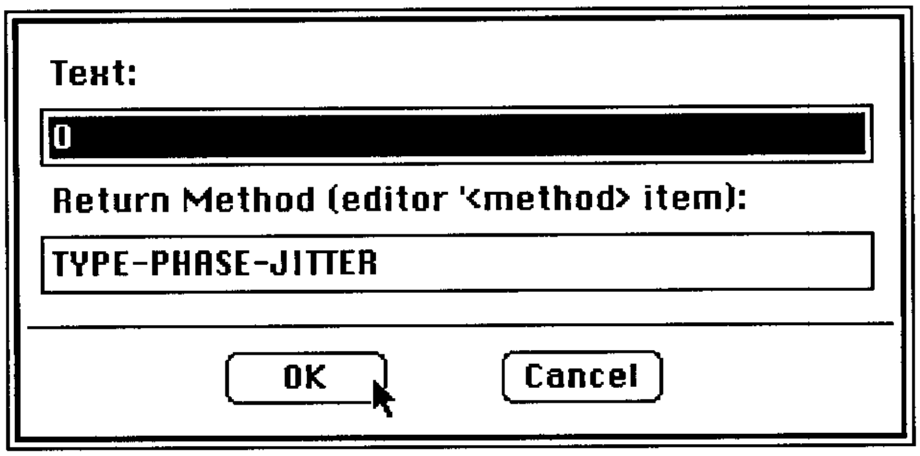
NOOP

Free Method (editor '<method> item):

NOOP

[✓ Enabled] Disabled

The function REDRAW-MAIN-CANVAS is a piece of code that will clear out and redraw that area. Similarly a text item may be specified to have a certain "click" method by creating it, double-clicking it, and filling in the appropriate fields in the resulting dialogue panel, as follows:



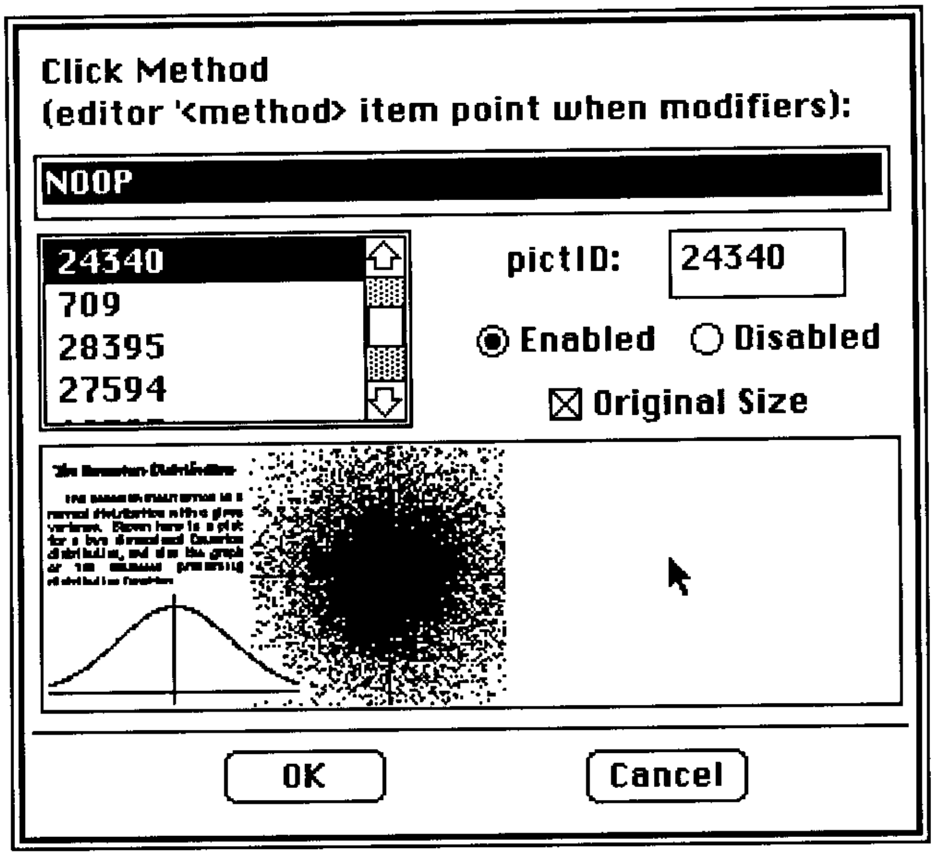
Text:

0

Return Method (editor '<method> item):

TYPE-PHASE-JITTER

Arbitrary icons and bitmaps may also be included as part of the editor panel being constructed. This is done by a dialogue as in the following example, in which a previously created bitmap is designated as part of the display of the forth Distribution Editor help screen:



Click Method
(editor '<method> item point when modifiers):

NOOP

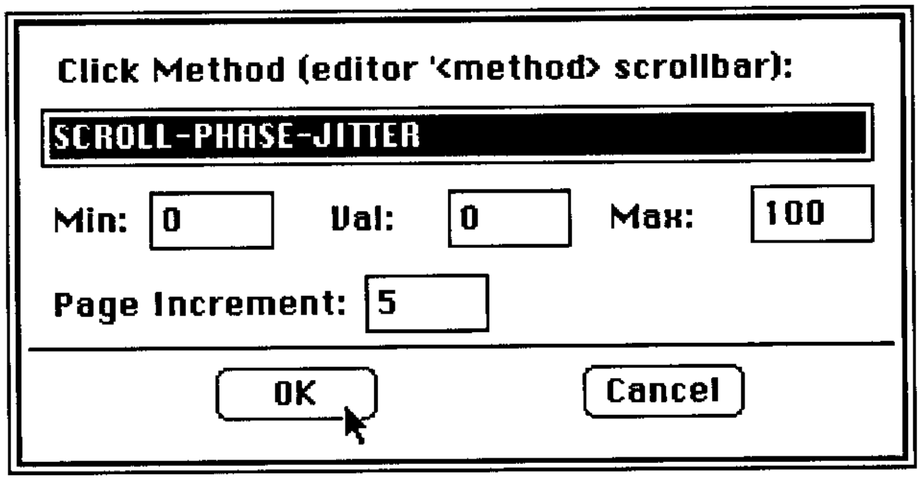
24340
709
28395
27594

pictID: 24340

[✓ Enabled] Disabled

☒ Original Size

Scroll bars of arbitrary sizes may be similarly created and placed in arbitrary locations by specifying the appropriate "click" method, as well as minimum and maximum values for the scroll interval:



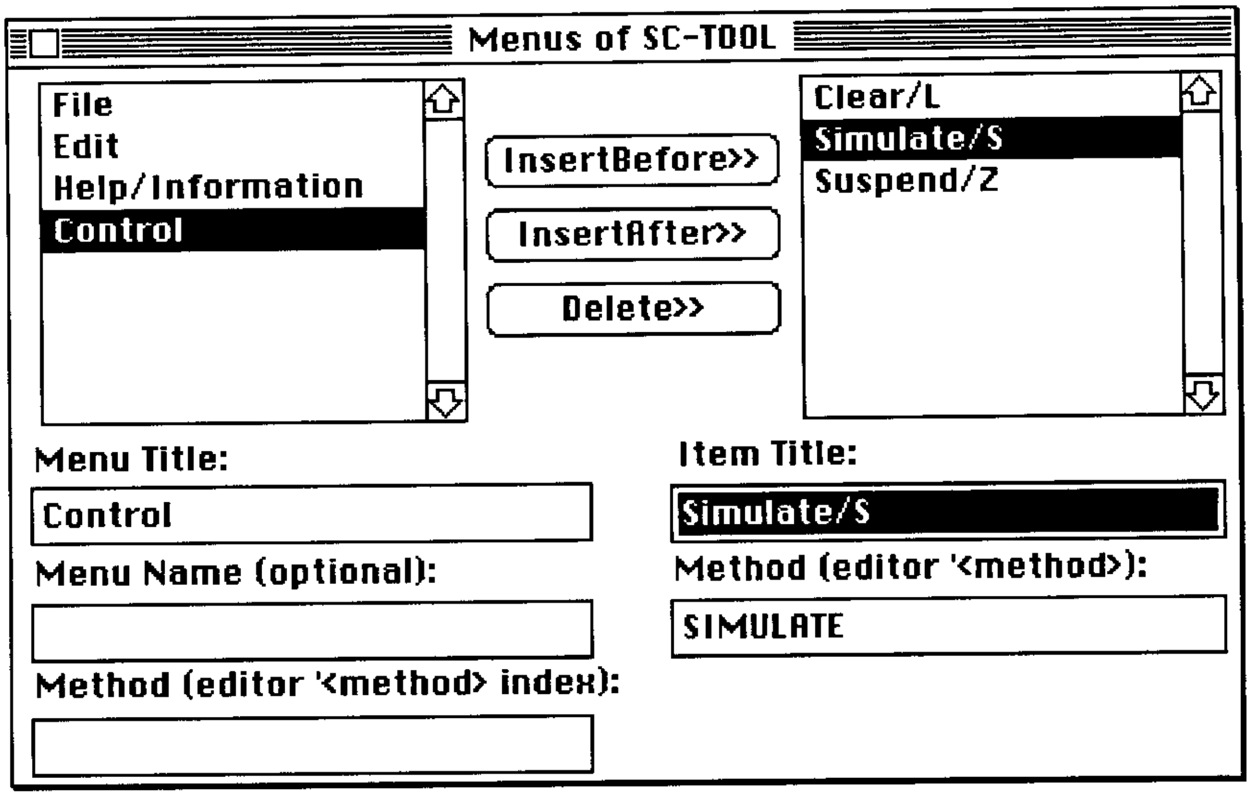
Click Method (editor '<method> scrollbar):

SCROLL-PHASE-JITTER

Min: 0 Val: 0 Max: 100

Page Increment: 5

Each editor panel has associated with it a pull-down menu bar containing several menus, each containing several menu items. A menu item is an entry in a menu that when selected causes some code to be executed. Menus are also constructed interactively in Interface Builder. For each named menu entry the user specifies a function to be called when that entry is selected. In addition the user may optionally specify a keystroke (denoted by a slash and a letter) that will execute the same functionality *without* having to go through the menu system. This is useful to experienced users who would find it easier to memorize a keystroke rather than waste a longer time pulling down and clicking a menu item. The following example illustrates the process of defining a menu, and is part of the main panel of the Signal Constellation Design Tool:



Menu Title: Control

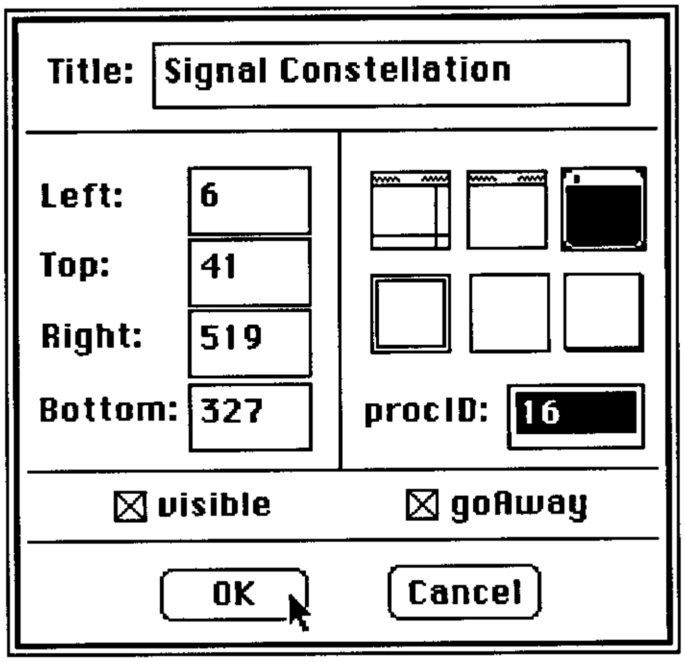
Menu Name (optional):

Method (editor '<method> index):

Item Title: Simulate/S

Method (editor '<method>): SIMULATE

The shape and characteristics of the panel window may itself be modified; a window may be optionally movable, scrollable, resizable, closable, have a title, etc. Here is an example of how these attributes are interactively specified via a dialogue with Interface Builder:



Title: Signal Constellation

Left: 6

Top: 41

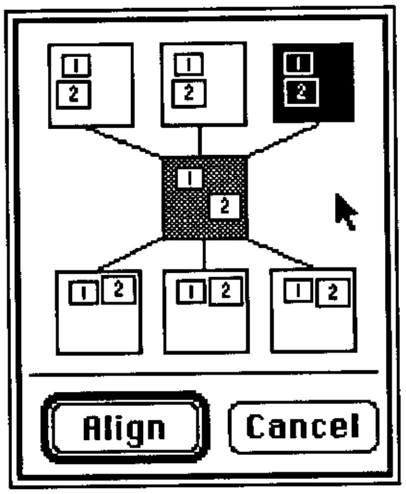
Right: 519

Bottom: 327

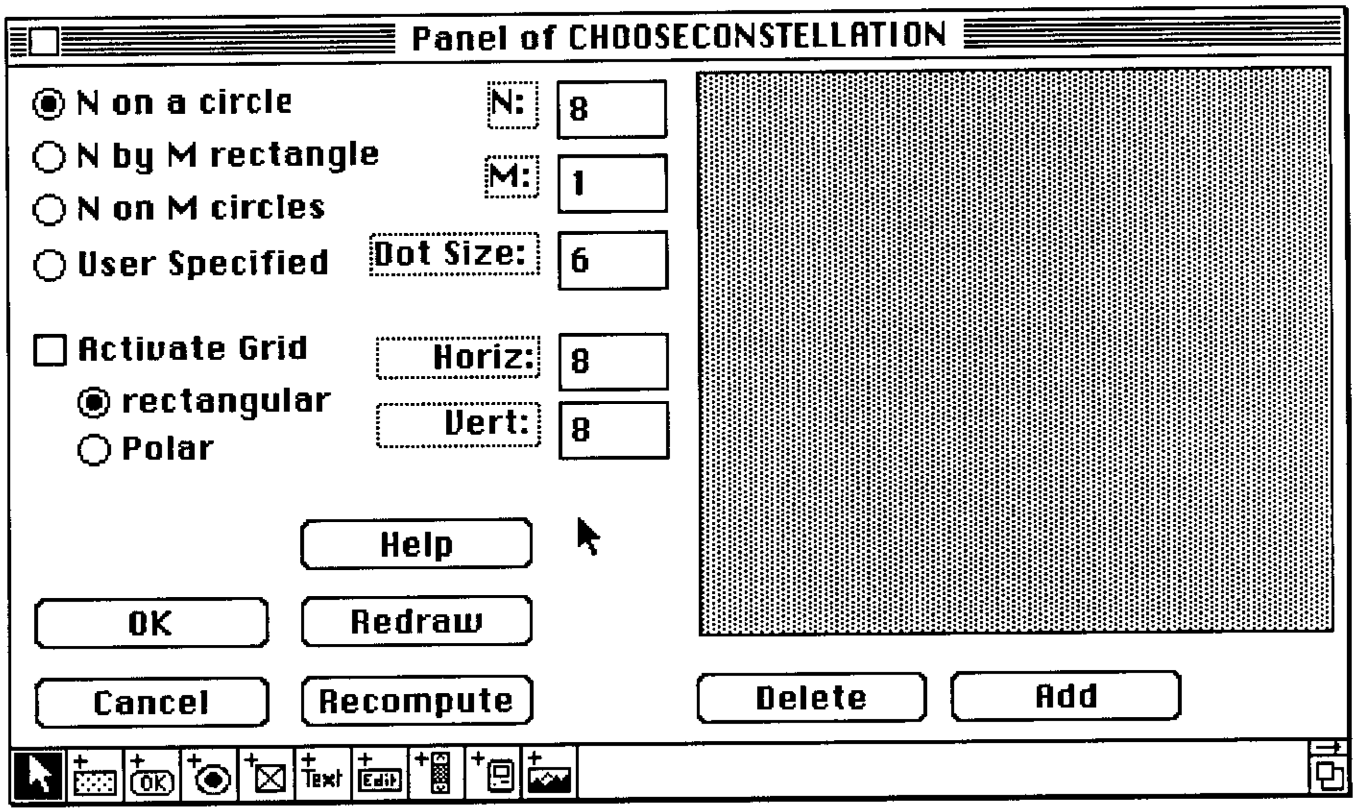
procID: 16

☒ visible ☒ goAway

To make the placement of fields more precise, Interface Builder provides a facility for aligning fields and also making groups of fields the same size. Such alignment/resizing helps to make the resulting panel more uniform in appearance; the alignment command is invoked by clicking on the appropriate icon in the following dialogue panel:



Proceeding in this manner we then construct inside Interface Builder the panel for the Constellation Editor, which appears as follows:


Panel of CHOOSECONSTELLATION

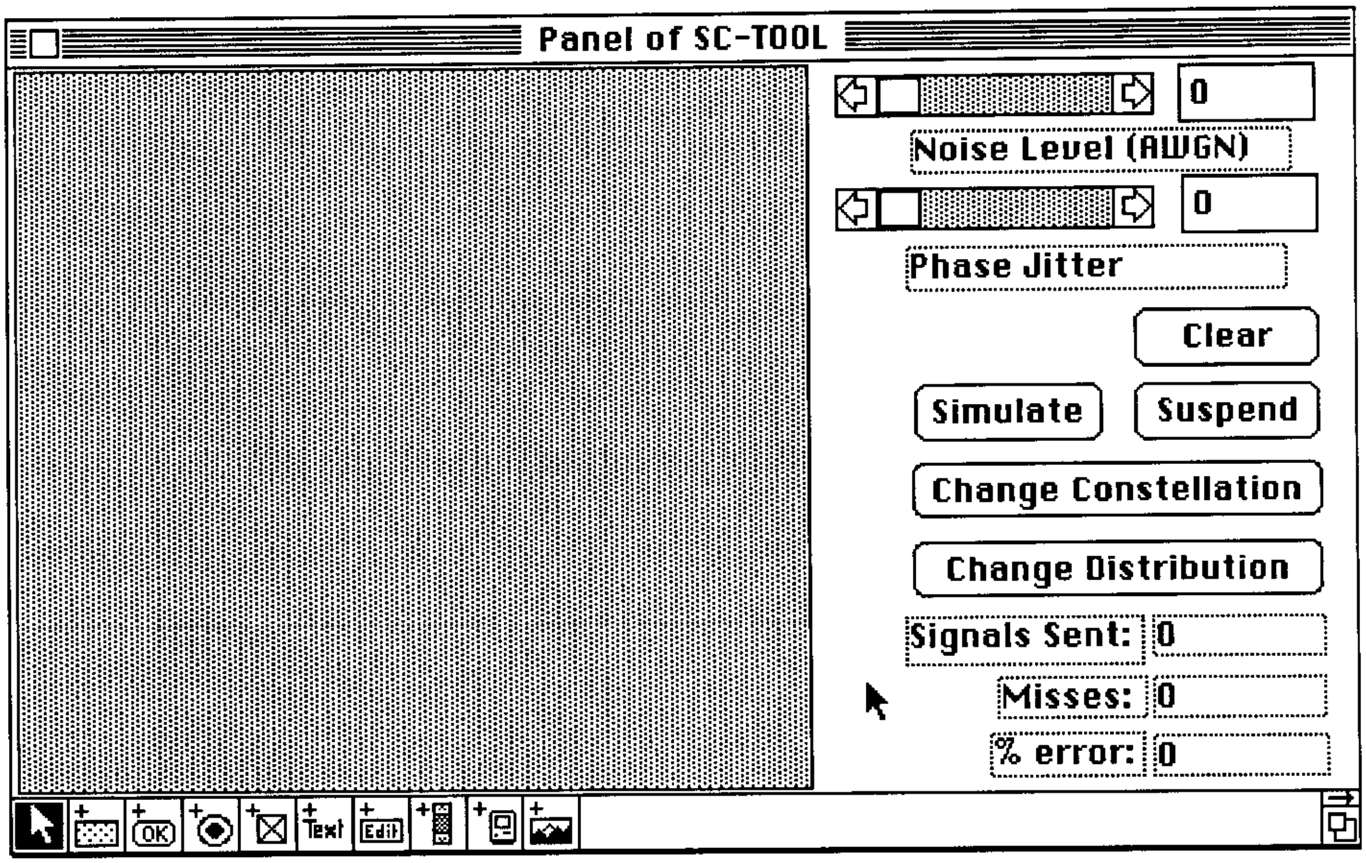
[✓ N on a circle]
N by M rectangle
N on M circles
User Specified

N: 8
M: 1
Dot Size: 6

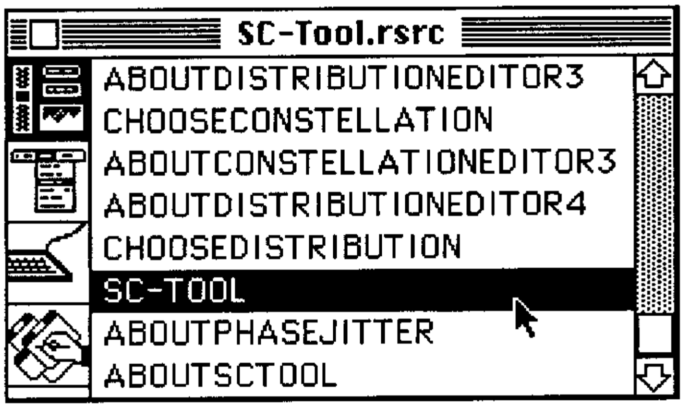
Activate Grid
[✓ rectangular]
Polar

Horiz: 8
Vert: 8

Finally we construct inside Interface Builder the main panel for the Signal Constellation Design Tool, which appears as follows:

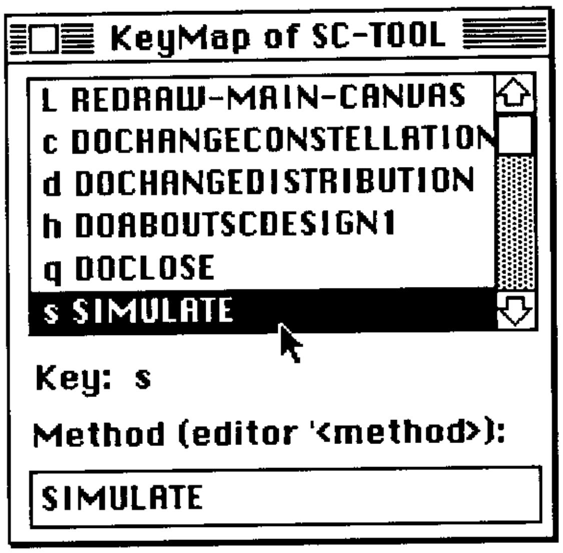


After several editor panels have been constructed, we obtain the following Interface Builder display window, where every line corresponds to an editor panel. The icons on the left (in top to bottom order) correspond to control item editing, menu editing, keymap editing, and subeditor editing, respectively:


SC-Tool.rsrc

ABOUTDISTRIBUTIONEDITOR3
CHOOSECONSTELLATION
ABOUTCONSTELLATIONEDITOR3
ABOUTDISTRIBUTIONEDITOR4
CHOOSEDISTRIBUTION
SC-TOOL
ABOUTPHASEJITTER
ABOUTSCTOOL

Control item and menu editing has been discussed previously. Keymap editing entails binding various functionality to keyboard keys. If a function F is bound to a keyboard key K, then during execution, whenever the key K is pressed, function F gets called. This provides an easy means to quickly invoke certain user-defined commands and functions. A common practice is to bind certain keys to important menu items in order to save experienced users the time to pull down a menu; instead, only a single key needs to be pressed. The dialogue panel in which such key bindings are specified in Interface Builder is given here:



4. Correctness and Functional Orthogonality

Since the underlying paradigm of Interface Builder is object-oriented in nature, a certain functional orthogonality exists in the finished software in the following sense. Messages sent to an object do not directly affect any other object, and moreover objects can only communicate by passing "messages" to one another (actually there is another way for objects to communicate, namely by assignment/reading of global variables, but this practice is not encouraged). This implies that flow of control is highly constrained and therefore the formation of side-effects, although possible, is nevertheless tightly controlled.

If a set of objects has been created and debugged and is found to operate correctly, adding new objects is not likely to affect any of the old objects or the correctness of their behavior. Moreover, the functionality of any of the objects may be invoked at any point in time via an appropriate message from any other object. Although at first glance this would seem to give rise to a certain "non-determinism" in execution, in practice, the programmer will be very informed about what code should/would execute under various circumstances, and my experience has shown that if the programmer has adhered to the standard object-oriented programming conventions, the "right thing" usually happens under even the most pathological circumstances.

The programmer's code does not have to worry about a "main-event-loop" and about dispatching certain pieces of code depending upon what event has transpired, because Interface Builder usurps the system's "main-event-loop" already and does all the necessary dispatching based on the programmer's specifications. This takes much of the complexity out of the application code, complexity that would otherwise have had to be duplicated from scratch in each application. Thus considerable programmer effort is saved by this scheme.

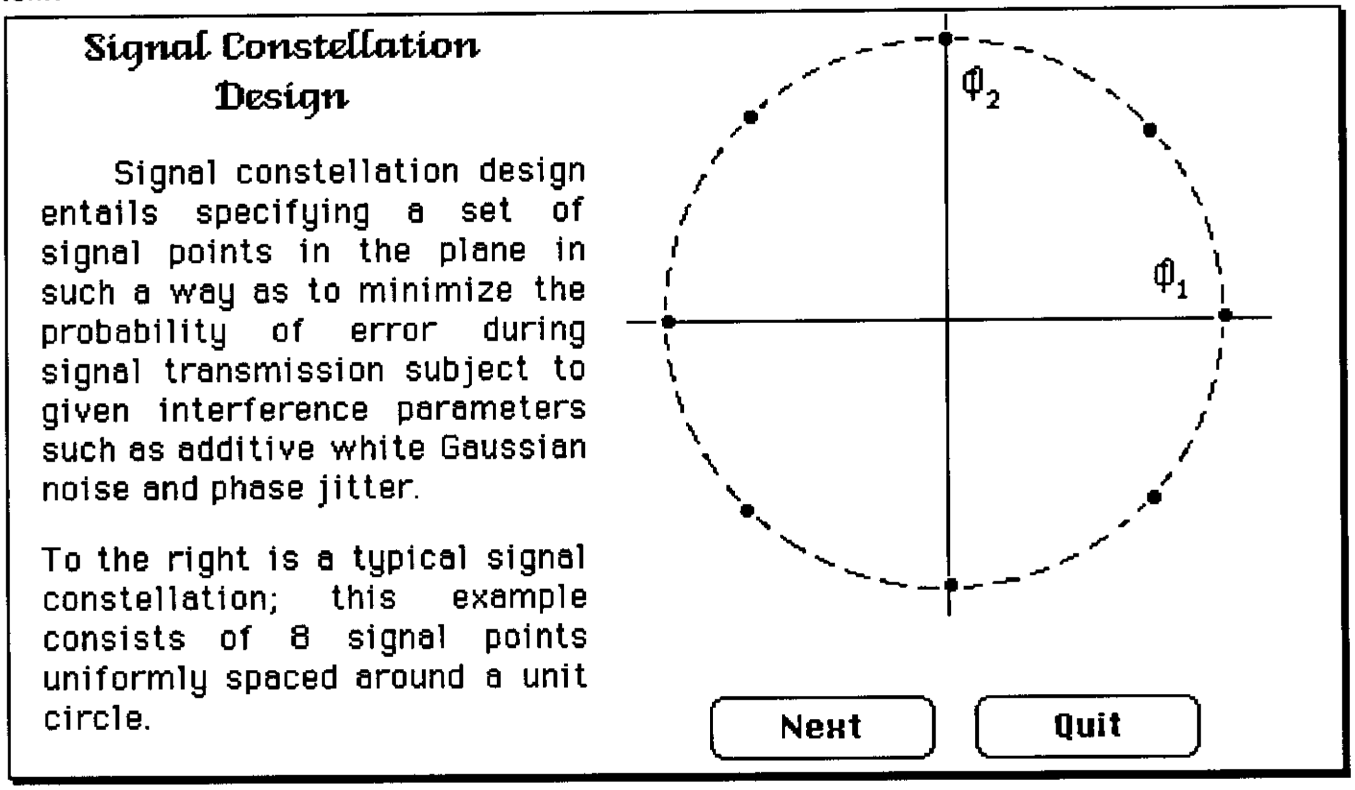
5. The On-Line Help Screens

As part of our user interface design, we provide a mechanism for presenting some interactive on-line help to the user. This help may be invoked via clicking a button, pulling a menu, or pressing a key. The help itself consists of one of more screens full of information, directions, and diagrams. The user may jump between these screens, or quit and return to the original mode before calling the help. Each screen (except the first) contains a "Previous" command button, that will expose and activate the screen that immediately precedes the current one in the logical continuation of the help. In addition, each screen (except the last) contains a "Next" command button, that will expose and activate the screen that immediately succeeds the current one in the logical continuation of the help sequence.

Having some on-line help is essential in many applications and often saves considerable (manual look-up) time for the user, especially if the help is also crossed referenced or indexed in some manner. A good example of useful on-line help facilities is contained in the MicroSoft Word 3.01 text processing program.

5.1. On-Line Help Screens for Signal Constellation Design

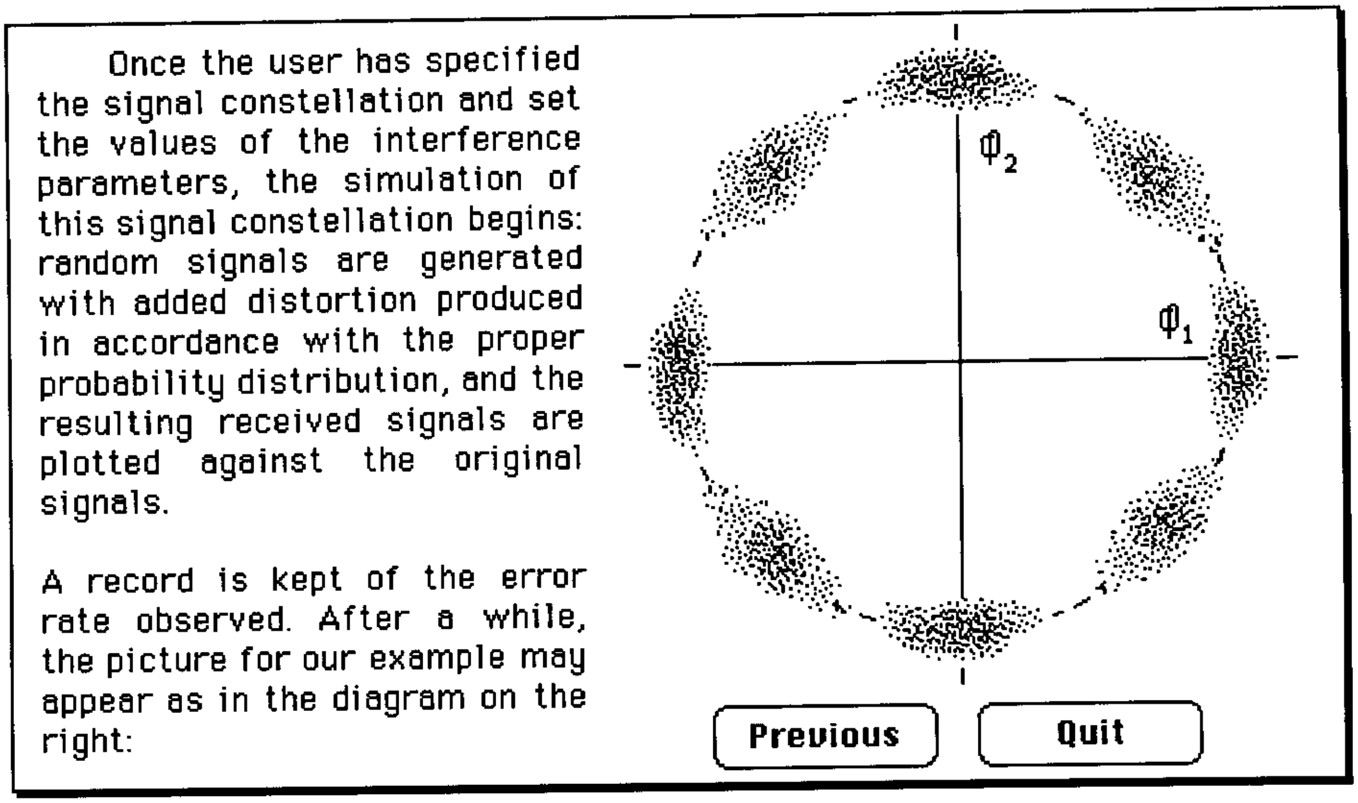
As an example of this discussion we depict here some of the help screens included in the user interface, beginning with the two on-line help screens for the "About signal constellation design" item:



Signal Constellation Design

Signal constellation design entails specifying a set of signal points in the plane in such a way as to minimize the probability of error during signal transmission subject to given interference parameters such as additive white Gaussian noise and phase jitter.

To the right is a typical signal constellation; this example consists of 8 signal points uniformly spaced around a unit circle.

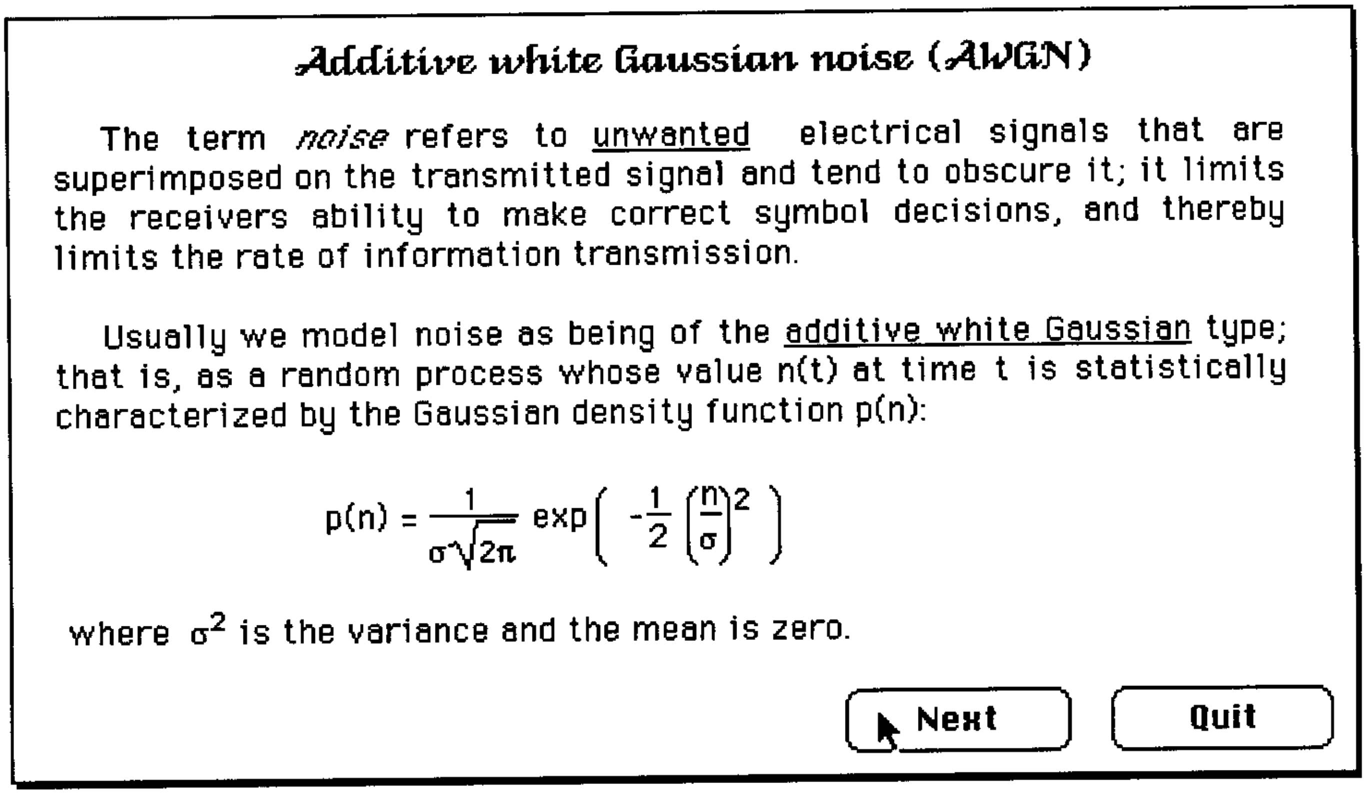


Once the user has specified the signal constellation and set the values of the interference parameters, the simulation of this signal constellation begins: random signals are generated with added distortion produced in accordance with the proper probability distribution, and the resulting received signals are plotted against the original signals.

A record is kept of the error rate observed. After a while, the picture for our example may appear as in the diagram on the right:

5.2. On-Line Help Screens for Additive White Gaussian Noise

Here are the three on-line help screens for the "additive white Gaussian noise" item:



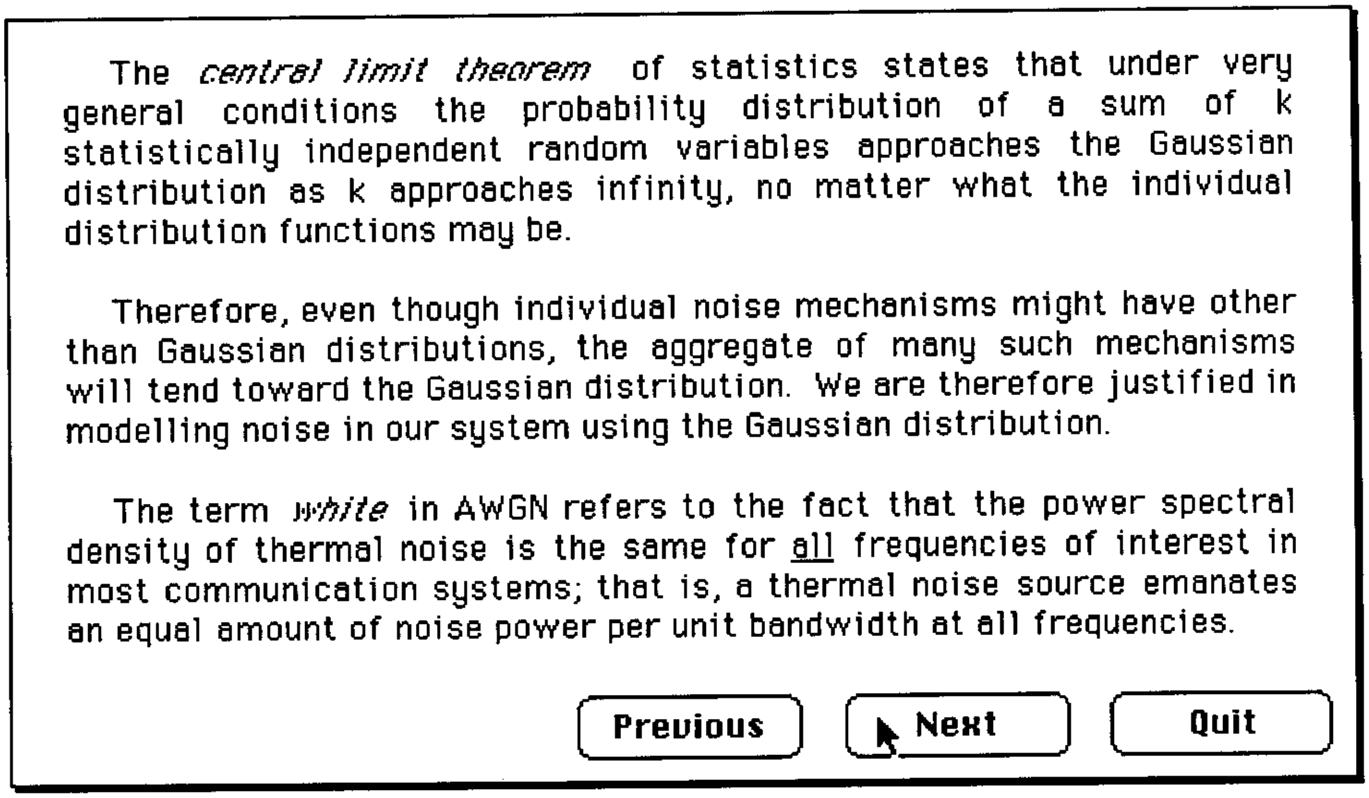
Additive white Gaussian noise (AWGN)

The term *noise* refers to unwanted electrical signals that are superimposed on the transmitted signal and tend to obscure it; it limits the receivers ability to make correct symbol decisions, and thereby limits the rate of information transmission.

Usually we model noise as being of the additive white Gaussian type; that is, as a random process whose value n(t) at time t is statistically characterized by the Gaussian density function p(n):

$$p(n) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2} \left(\frac{n}{\sigma}\right)^2 \right)$$

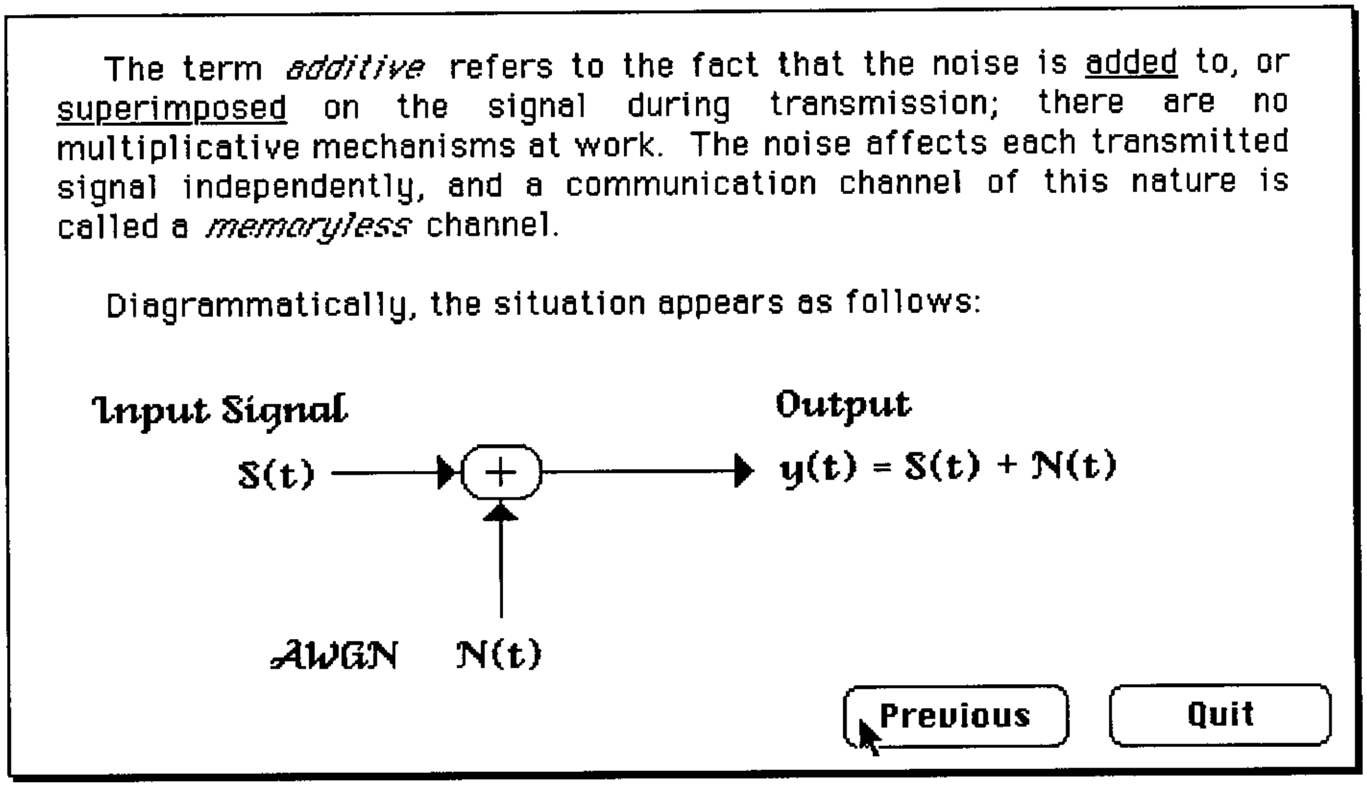
where σ^2 is the variance and the mean is zero.



The *central limit theorem* of statistics states that under very general conditions the probability distribution of a sum of k statistically independent random variables approaches the Gaussian distribution as k approaches infinity, no matter what the individual distribution functions may be.

Therefore, even though individual noise mechanisms might have other than Gaussian distributions, the aggregate of many such mechanisms will tend toward the Gaussian distribution. We are therefore justified in modelling noise in our system using the Gaussian distribution.

The term *white* in AWGN refers to the fact that the power spectral density of thermal noise is the same for <u>all</u> frequencies of interest in most communication systems; that is, a thermal noise source emanates an equal amount of noise power per unit bandwidth at all frequencies.



The term *additive* refers to the fact that the noise is <u>added</u> to, or <u>superimposed</u> on the signal during transmission; there are no multiplicative mechanisms at work. The noise affects each transmitted signal independently, and a communication channel of this nature is called a *memoryless* channel.

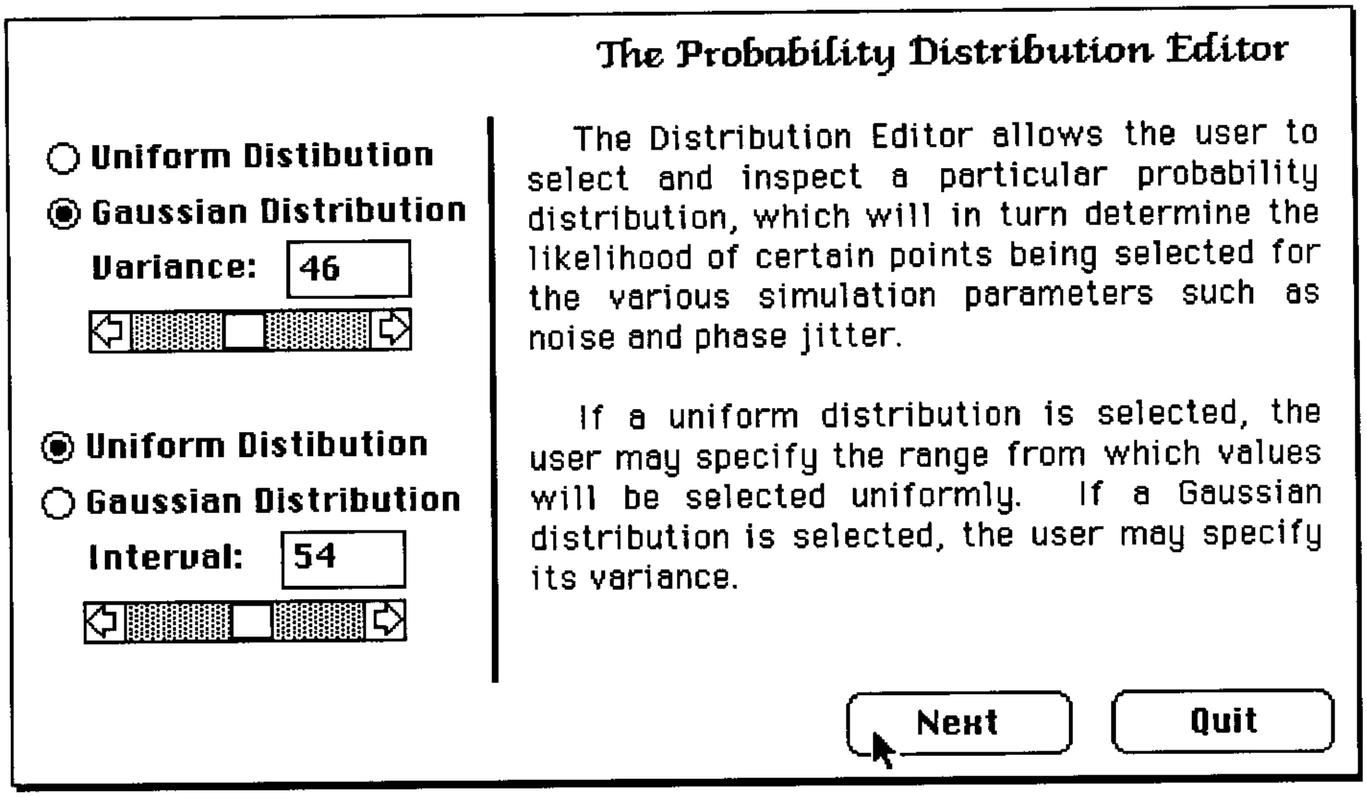
Diagrammatically, the situation appears as follows:

$$\begin{tikzcd} \text{Input Signal} & & \text{Output} \\ S(t) \arrow[r] & \boxed{+} \arrow[r] & y(t) = S(t) + N(t) \\ \text{AWGN} & N(t) \arrow[u] & \end{tikzcd}$$

Note that such help screens may contain both text, equations, graphics, and active control objects such as buttons and scroll bars; the functionality of the latter is completely user-controlled, as is the general layout, placement, and appearance of these items.

5.3. On-Line Help Screens for Distribution Editor

Here are the four on-line help screens for the Distribution Editor:



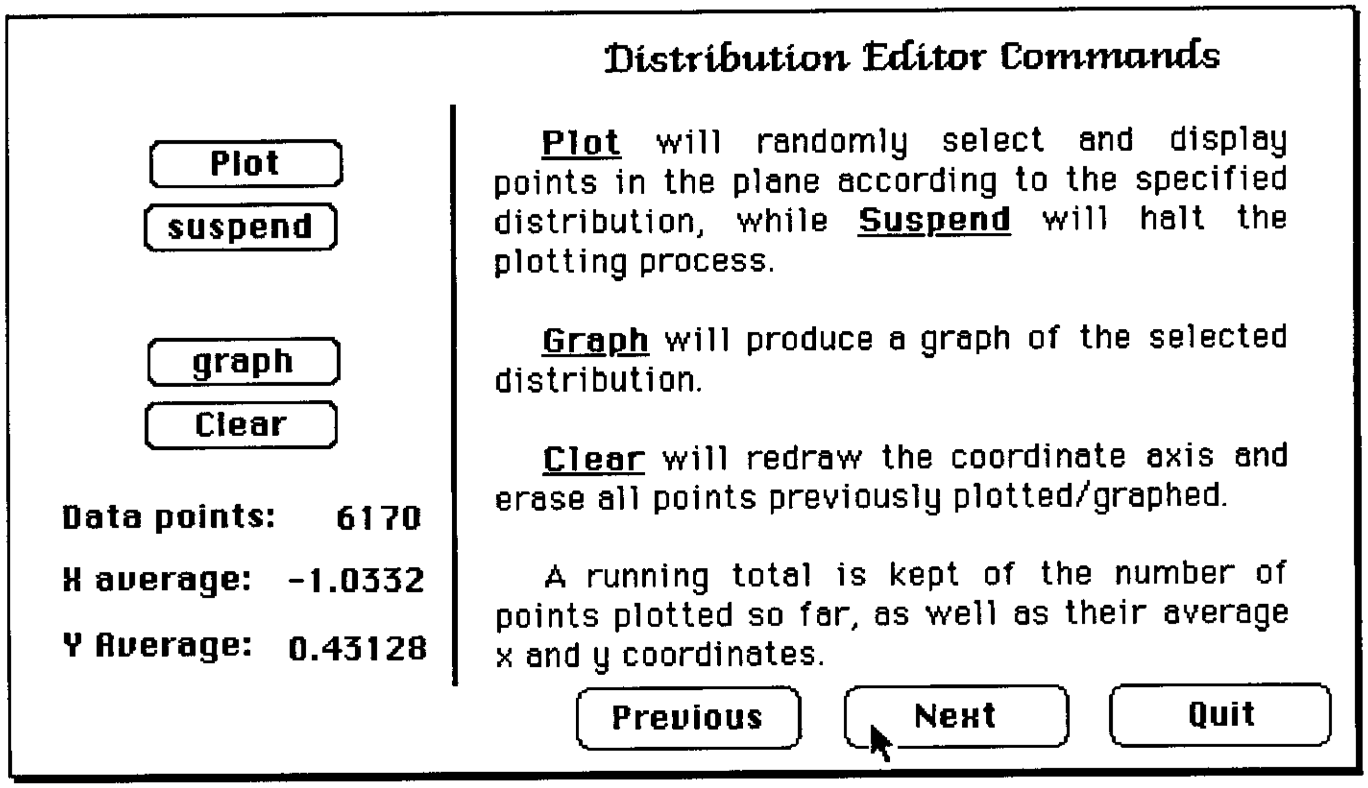
The Probability Distribution Editor

○ Uniform Distibution
[✓ Gaussian Distribution]
Variance: 46

[✓ Uniform Distibution]
○ Gaussian Distribution
Interval: 54

The Distribution Editor allows the user to select and inspect a particular probability distribution, which will in turn determine the likelihood of certain points being selected for the various simulation parameters such as noise and phase jitter.

If a uniform distribution is selected, the user may specify the range from which values will be selected uniformly. If a Gaussian distribution is selected, the user may specify its variance.



Distribution Editor Commands

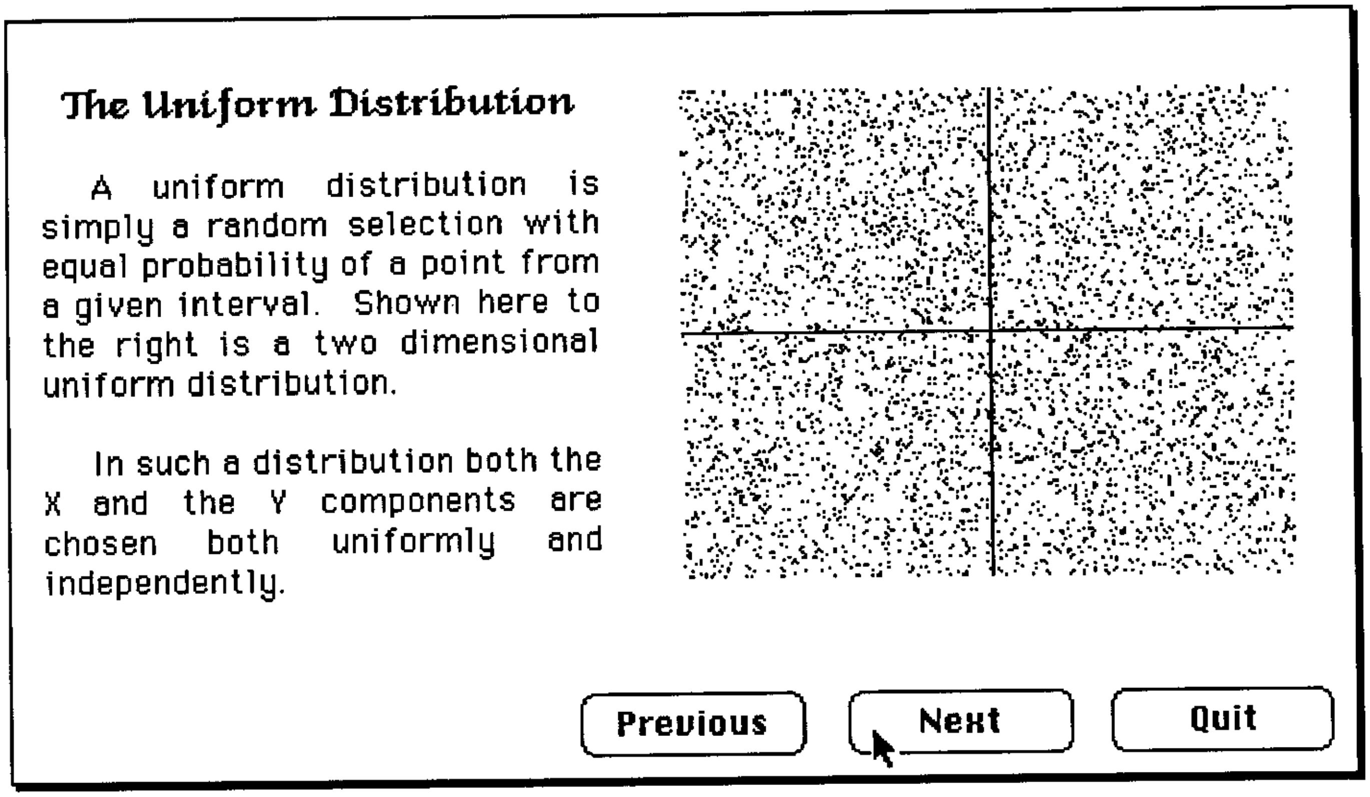
Data points: 6170
X average: -1.0332
Y Average: 0.43128

Plot will randomly select and display points in the plane according to the specified distribution, while **Suspend** will halt the plotting process.

Graph will produce a graph of the selected distribution.

Clear will redraw the coordinate axis and erase all points previously plotted/graphed.

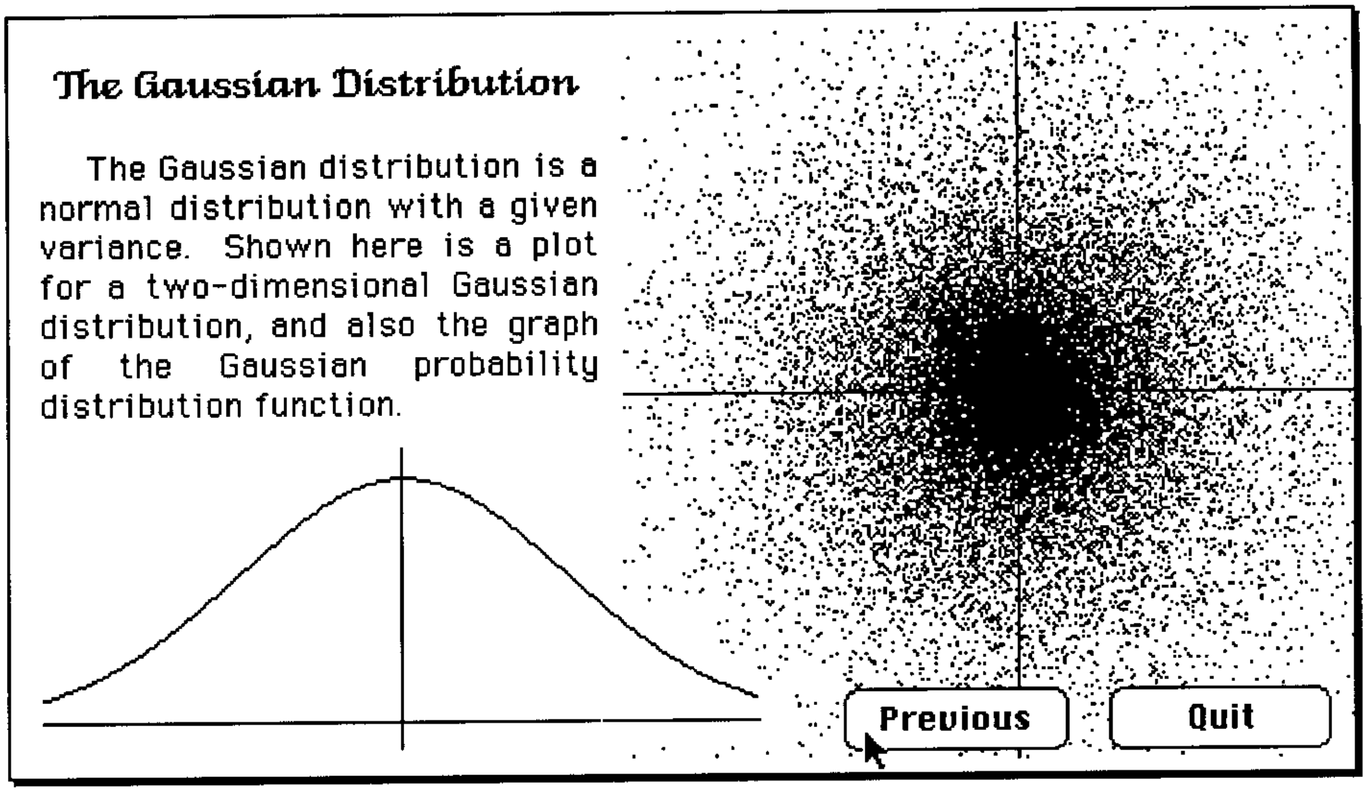
A running total is kept of the number of points plotted so far, as well as their average x and y coordinates.



The Uniform Distribution

A uniform distribution is simply a random selection with equal probability of a point from a given interval. Shown here to the right is a two dimensional uniform distribution.

In such a distribution both the X and the Y components are chosen both uniformly and independently.

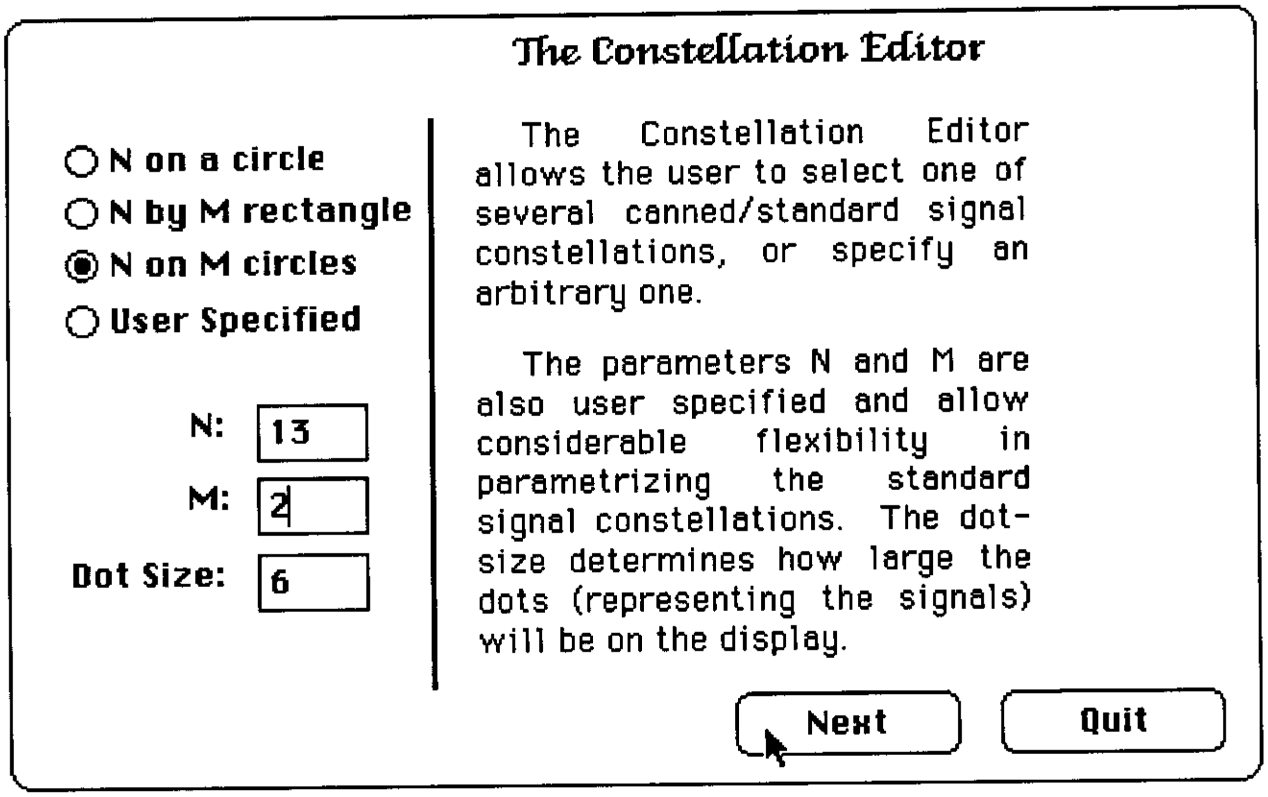


The Gaussian Distribution

The Gaussian distribution is a normal distribution with a given variance. Shown here is a plot for a two-dimensional Gaussian distribution, and also the graph of the Gaussian probability distribution function.

5.4. On-Line Help Screens for Constellation Editor

Here are the three on-line help screens for the Constellation Editor:



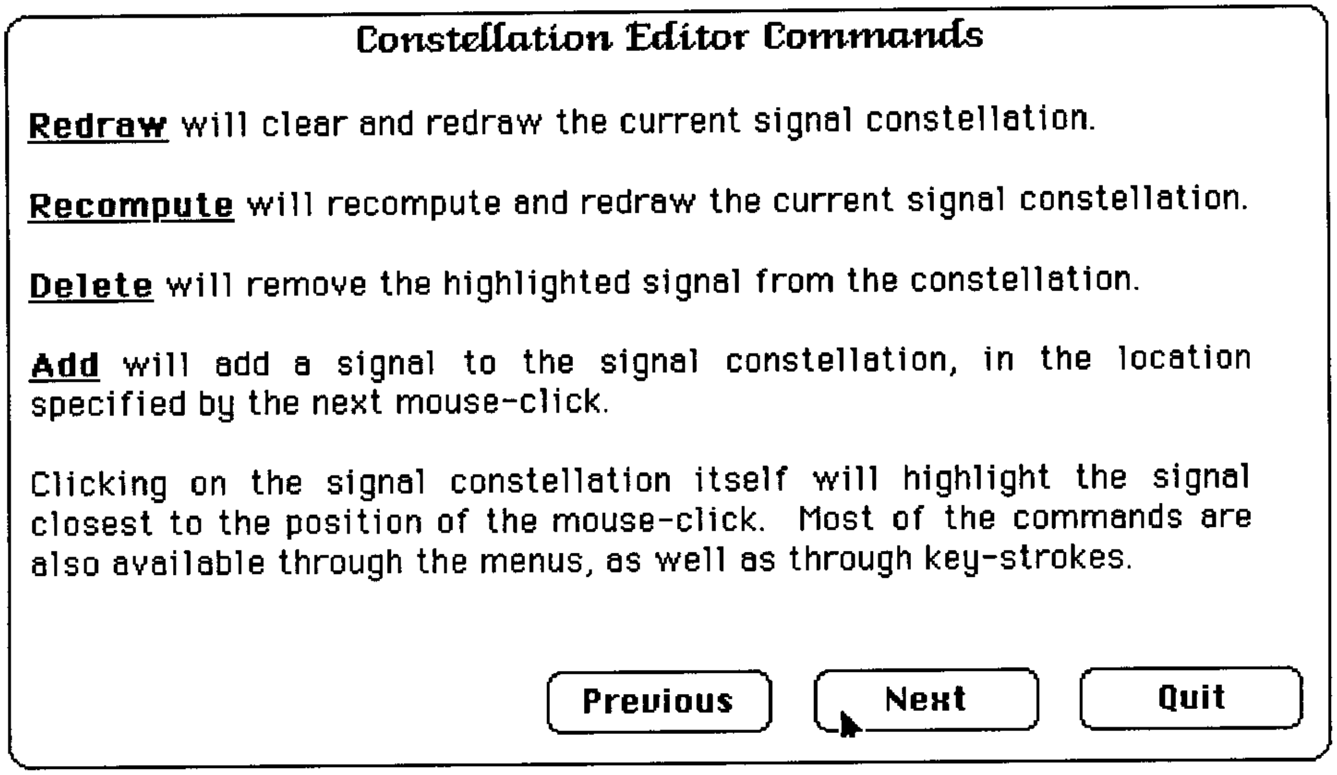
The Constellation Editor

○ N on a circle
○ N by M rectangle
[✓ N on M circles]
○ User Specified

N: 13
M: 2
Dot Size: 6

The Constellation Editor allows the user to select one of several canned/standard signal constellations, or specify an arbitrary one.

The parameters N and M are also user specified and allow considerable flexibility in parametrizing the standard signal constellations. The dot-size determines how large the dots (representing the signals) will be on the display.



Constellation Editor Commands

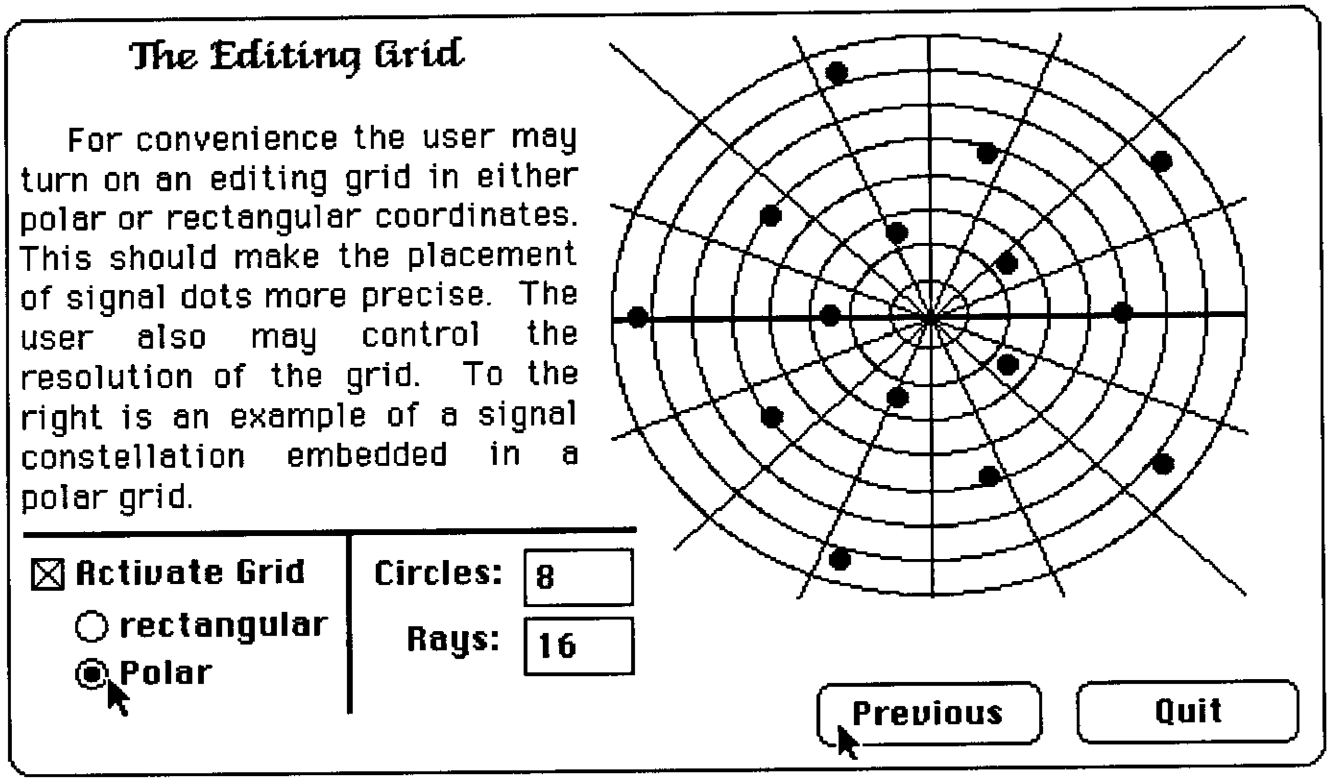
Redraw will clear and redraw the current signal constellation.

Recompute will recompute and redraw the current signal constellation.

Delete will remove the highlighted signal from the constellation.

Add will add a signal to the signal constellation, in the location specified by the next mouse-click.

Clicking on the signal constellation itself will highlight the signal closest to the position of the mouse-click. Most of the commands are also available through the menus, as well as through key-strokes.



The Editing Grid

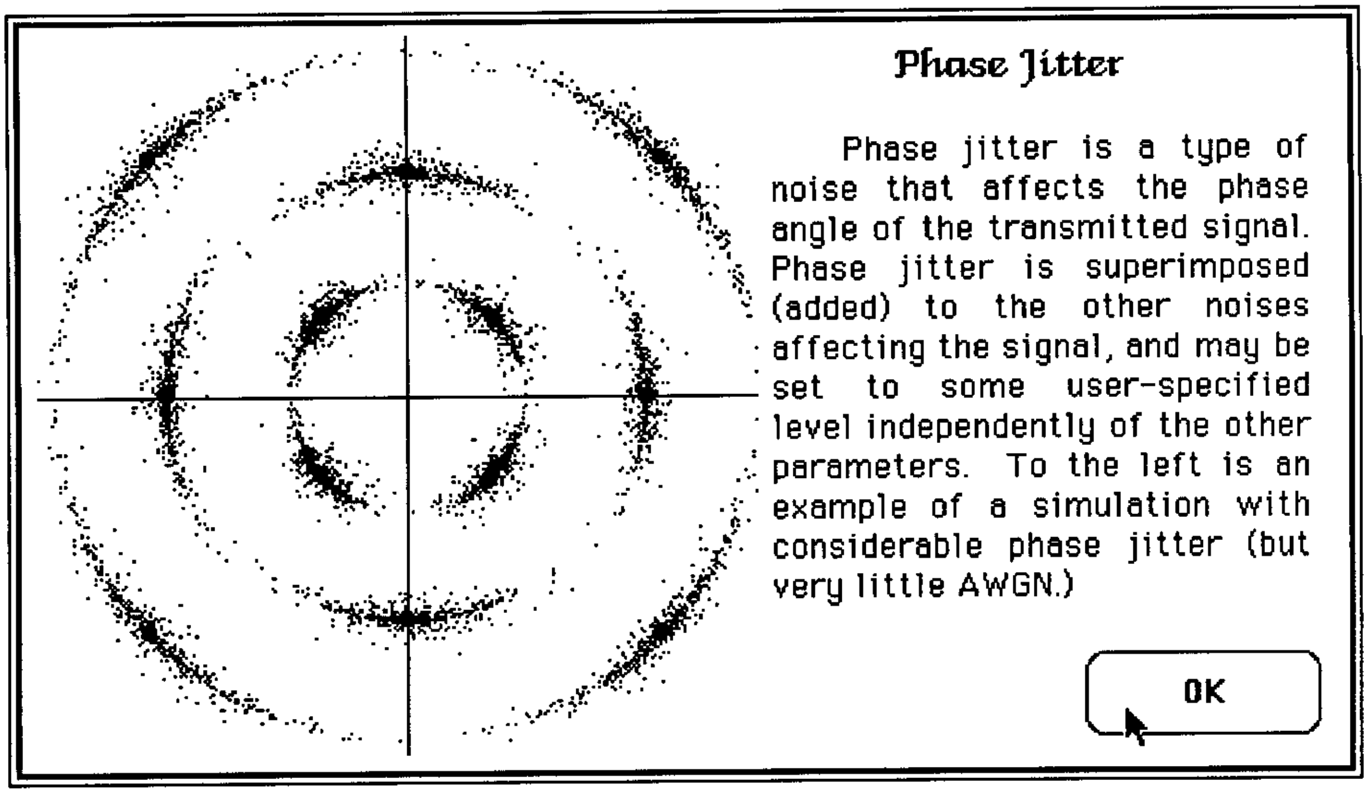
For convenience the user may turn on an editing grid in either polar or rectangular coordinates. This should make the placement of signal dots more precise. The user also may control the resolution of the grid. To the right is an example of a signal constellation embedded in a polar grid.

[☒] Activate Grid
rectangular [✓ Polar]

Circles: 8
Rays: 16

5.5. On-Line Help Screen for Phase Jitter

Here is the on-line help screen for the "phase jitter" item (note that when there is only one screen-full of help, there are no "Next" or "Previous" command buttons):

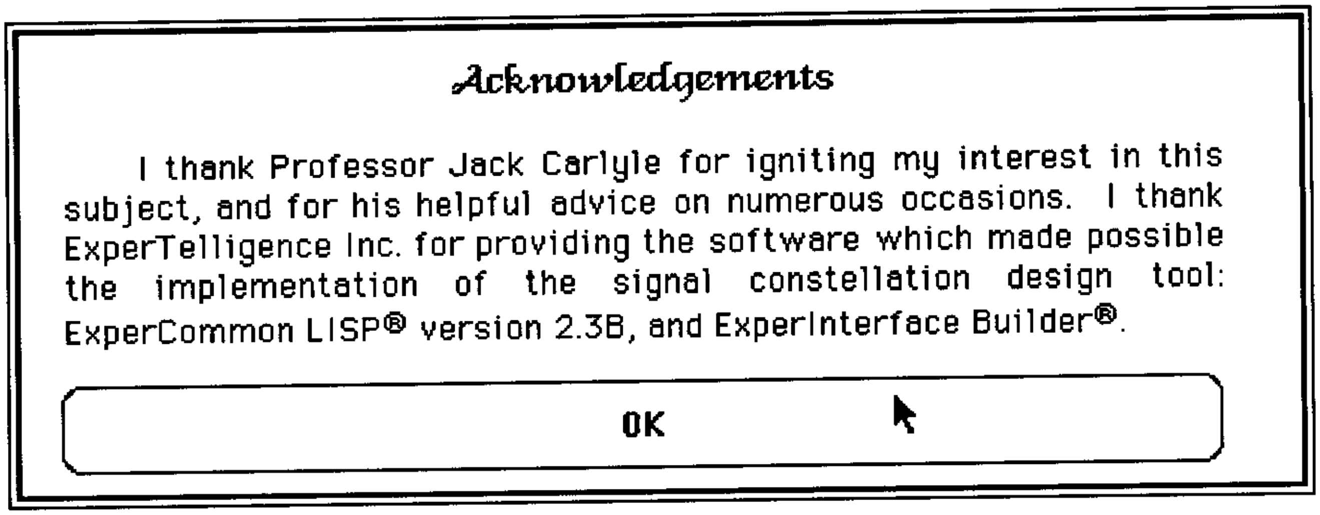


Phase Jitter

Phase jitter is a type of noise that affects the phase angle of the transmitted signal. Phase jitter is superimposed (added) to the other noises affecting the signal, and may be set to some user-specified level independently of the other parameters. To the left is an example of a simulation with considerable phase jitter (but very little AWGN.)

5.6. On-Line Help Screen for the Acknowledgements

Here is the on-line help screen for the "acknowledgements" item:

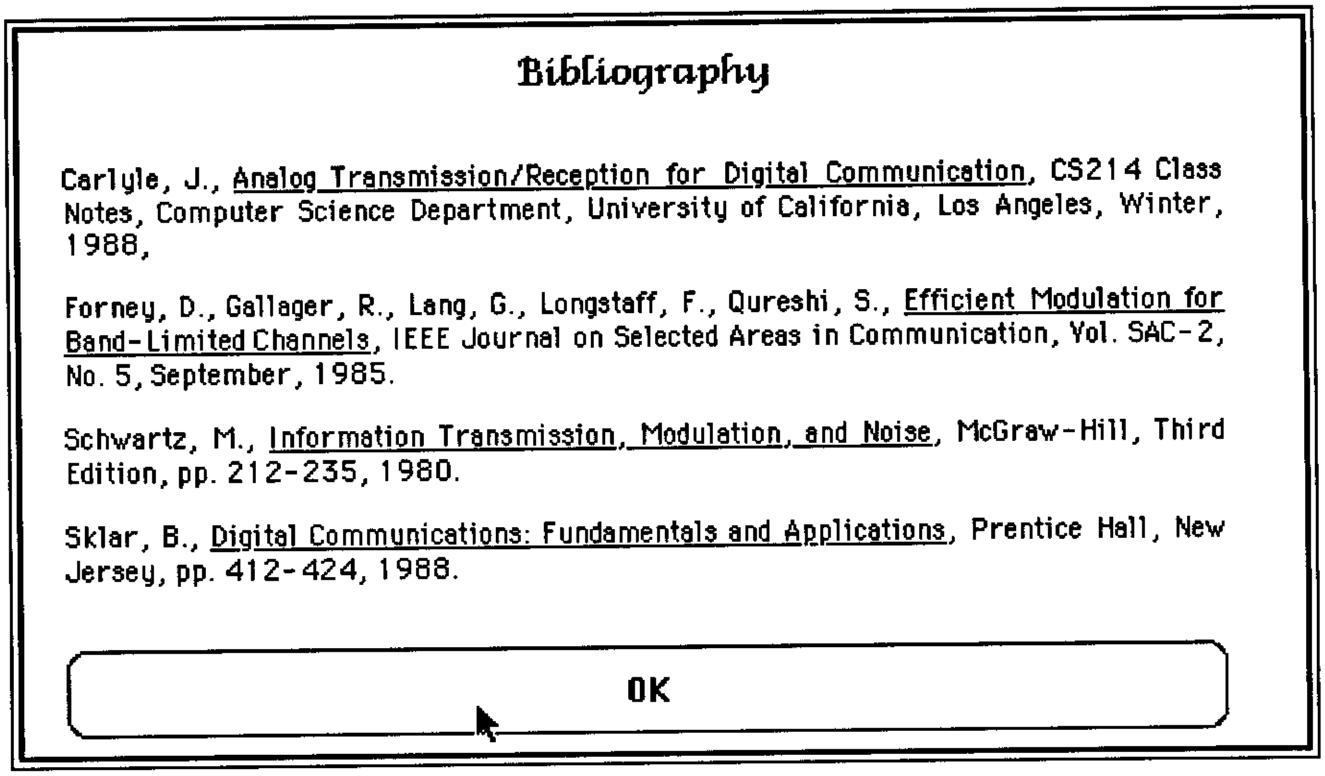


Acknowledgements

I thank Professor Jack Carlyle for igniting my interest in this subject, and for his helpful advice on numerous occasions. I thank ExperTelligence Inc. for providing the software which made possible the implementation of the signal constellation design tool: ExperCommon LISP® version 2.3B, and ExperInterface Builder®.

5.7. On-Line Help Screen for the References

Here is the on-line help screen for the "references" item:



Bibliography

Carlyle, J., Analog Transmission/Reception for Digital Communication, CS214 Class Notes, Computer Science Department, University of California, Los Angeles, Winter, 1988,

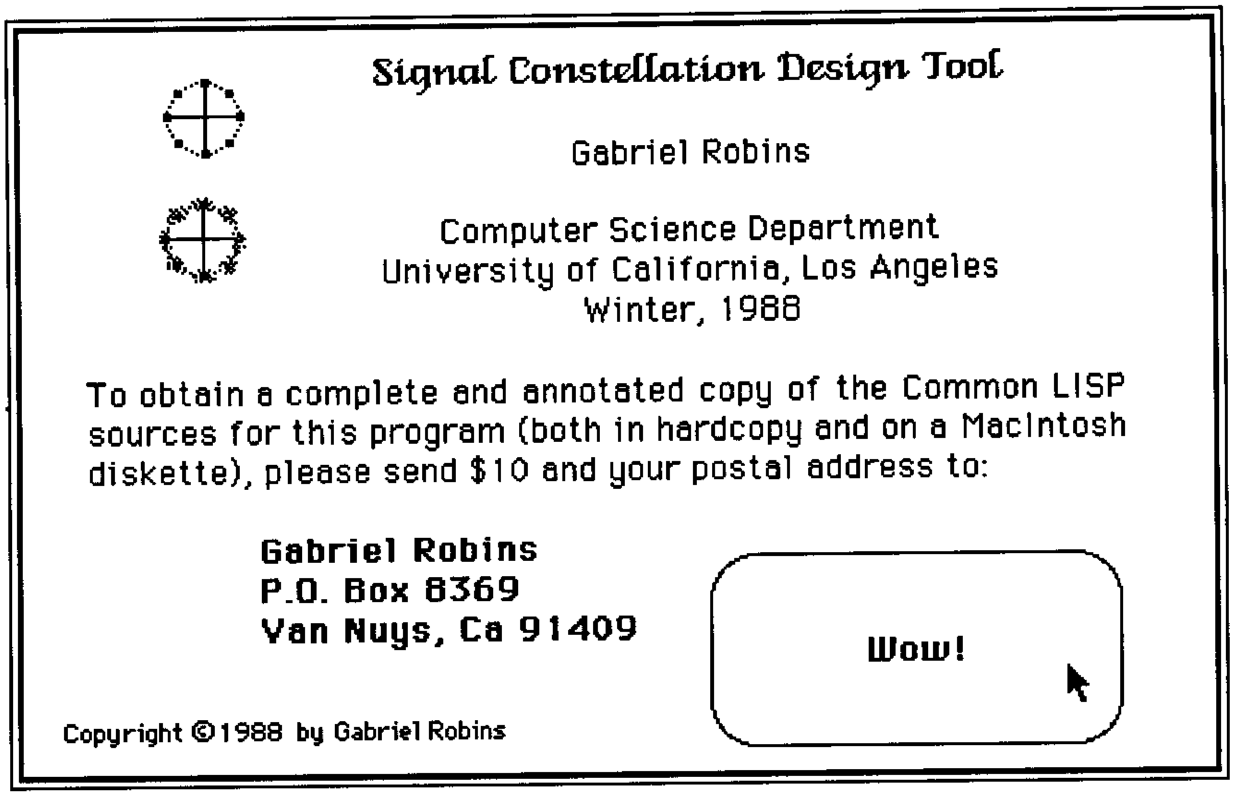
Forney, D., Gallager, R., Lang, G., Longstaff, F., Qureshi, S., Efficient Modulation for Band-Limited Channels, IEEE Journal on Selected Areas in Communication, Vol. SAC-2, No. 5, September, 1985.

Schwartz, M., Information Transmission, Modulation, and Noise, McGraw-Hill, Third Edition, pp. 212-235, 1980.

Sklar, B., Digital Communications: Fundamentals and Applications, Prentice Hall, New Jersey, pp. 412-424, 1988.

5.8. On-Line Help Screen for the "About..." Item

Here is the on-line help screen for the "About..." item; this screen contains some general information regarding the Signal Constellation Design Tool and is displayed when the user selects the "About..." menu item from the main menu anytime during execution:



Signal Constellation Design Tool

Gabriel Robins

Computer Science Department
University of California, Los Angeles
Winter, 1988

To obtain a complete and annotated copy of the Common LISP sources for this program (both in hardcopy and on a MacIntosh diskette), please send $10 and your postal address to:

Gabriel Robins
P.O. Box 8369
Van Nuys, Ca 91409

Copyright ©1988 by Gabriel Robins

6. The Software/Hardware Used

The Signal Constellation Design Tool is implemented in ExperCommon LISP® (version 2.3B), marketed by ExperTelligence, Inc. [Bollay, McConnell, Reali, and Ritz]. ExperCommon LISP is a production LISP system that runs on the MacIntosh family of machines. The user interface was constructed with ExperInterface Builder®, an interactive package that allows a user to quickly and easily design a graphical (menu and icon-based) user interface from scratch on an object-oriented paradigm [Hullot]. The hardware used was the MacIntosh Plus with 2 megabytes of memory and a 20-megabyte hard disk. ExperCommon LISP® and ExperInterface Builder® may be purchased directly from ExperTelligence Inc., 5638 Hollister Avenue, 3rd Floor, Goleta, California 93117, U.S.A., (805) 967-1797..

7. Obtaining the sources

The annotated Common LISP sources for the Signal Constellation Design Tool are available upon request. Although this tool was developed on the MacIntosh, it should be portable to any system which supports Common LISP and reasonable window and graphics conventions. To obtain the sources, both in hardcopy and on a MacIntosh diskette, please send $10 to Gabriel Robins, UCLA Computer Science Department, Los Angeles, California, 90024.

8. Summary

Signal constellation design essentially entails trading off error frequency against information throughput, a chief occupation of modem designers. We proposed and implemented an interactive tool for designing and simulating arbitrary signal constellations. While the actual code that simulates signal constellations is rather trivial in itself, the user interface to this code is not.

To design and construct the user interface we have used Interface Builder, a new interactive tool that greatly facilitates the synthesis of user interfaces through an object-oriented methodology. Using the Interface Builder package and the Signal Constellation Design Tool as the target prototype, we showed how an order-of-magnitude improvement can be achieved in the effort required to produce a complex user interface.

We hope that we have helped to dispel some of the mystique surrounding user interface synthesis on state-of-the-art workstations by showing that given the proper tools and methodology, the synthesis of complex user interfaces could be rather trivial. In particular, designing and implementing the user interface specified here took only a few days, and that includes the overhead to read the user manuals and learn (from scratch) how to use the software.

9. Acknowledgements

I thank Professor Jack Carlyle for igniting my interest in this subject, and for his helpful advice on numerous occasions. I thank ExperTelligence Inc. for providing the software which made possible the implementation of the Signal Constellation Design Tool.

10. Bibliography

Bollay, D., McConnell, J., Reali, R., Ritz, D., ExperCommon LISP Documentation: Volume I, II, and III, The ExperTelligence Press, Santa Barbara, California, 1987.

Carlyle, J., Analog Transmission/Reception for Digital Communication, CS214 Class Notes, Computer Science Department, University of California, Los Angeles, Winter, 1988,

Forney, D., Gallager, R., Lang, G., Longstaff, F., Qureshi, S., Efficient Modulation for Band-Limited Channels, IEEE Journal on Selected Areas in Communication, Vol. SAC-2, No. 5, September, 1985.

Goodman, D., The Complete HyperCard Handbook, Bantham Books, New York, 1987.

Hullot, J., ExperInterface Builder Documentation, The ExperTelligence Press, Santa Barbara, California, 1987.

Kaczmarek, T., Mark, W., & Wilczynski, D., The CUE Project, Proceedings of SoftFair, July, 1983.

Robins, G. The ISI Grapher: A Portable Tool for Displaying Graphs Pictorially, Invited Talk in Symboliikka '87, Helsinki, Finland, August, 17-18, 1987. Reprinted in Multicomputer Vision, Levialdi, S., Chapter 12, Academic Press, London, 1988.

Schwartz, M., Information Transmission, Modulation, and Noise, McGraw-Hill, Third Edition, pp. 212-235, 1980.

Sklar, B., Digital Communications: Fundamentals and Applications, Prentice Hall, New Jersey, pp. 412-424, 1988.

11. Appendix I: The Source Code

Signal Constellation Design Tool

by Gabriel Robins

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024, USA
gabriel@vaxb.isi.edu

Copyright © 1988 by Gabriel Robins

This is a signal constellation design tool, conceived, written, and implemented by Gabriel Robins. This program was written in Common LISP and is running under the ExperCommon LISP and Interface Builder products of ExperTelligence Inc.

To obtain the annotated source code both in hardcopy and on a MacIntosh diskette, please send $10 to Gabriel Robins, P.O. Box 8369, Van Nuys, CA 91409. The sources may be used for all educational and research purposes, so long as appropriate credit is cited; on the other hand, the use of this code or the ideas contained therein for any commercial purpose, either direct or indirect, is strictly prohibited without the prior written consent of the author.

The following class defines the main user-interface panel/editor. It contains various fields/parameters that the user may modify and other that are computed and stored away for future use.

```
(defclass (SC-Tool Editor)
  (ivs  typed-noise-level        ; noise level (AWGN) typed in by the user
        typed-phase-jitter       ; phase jitter level (AWGN) typed in by the user
        main-canvas              ; main panel drawing area
        width                    ; width of the main drawing area
        height                   ; height of the main drawing area
        canvas-right             ; right corner of drawing area
        canvas-left              ; left corner of drawing area
        canvas-top               ; top corner of drawing area
        canvas-bottom            ; bottom corner of drawing area
        percent-error-box        ; field displaying error percentage
        nop-box                  ; field displaying number of signals generated
        misses-box               ; field displaying number of missed signals
        nop                      ; number of signals generated
        misses                   ; number of signals missed
        N                        ; signal constellation N parameter
        M                        ; signal constellation M parameter
        SC-Type                  ; type of the current signal constellation
        Constellation-Points     ; list of displayed constellation points
        Dot-Size                 ; size (in pixels) of the signal dot
        ConstEd                  ; constellation (sub)editor
        DistEd                   ; distribution (sub)editor
```

```
      AWGNEd1                        ; panel 1 of the AWGN help
      AWGNEd2                        ; panel 2 of the AWGN help
      AWGNEd3                        ; panel 3 of the AWGN help
      AboutConstEd1                  ; panel 1 of the constellation editor help
      AboutConstEd2                  ; panel 2 of the constellation editor help
      AboutConstEd3                  ; panel 3 of the constellation editor help
      AboutDistEd1                   ; panel 1 of the distribution editor help
      AboutDistEd2                   ; panel 2 of the distribution editor help
      AboutDistEd3                   ; panel 3 of the distribution editor help
      AboutDistEd4                   ; panel 4 of the distribution editor help
      simulation-on                  ; boolean flag: simulation status
      actual-coordinates             ; normalized (on unit disk) constellation points
      canvas-region                  ; clipping region for the main drawing area
      infinite-region                ; used to turn clipping off
    ))
```

The following code is executed whenever the main user interface panel is created, i.e. when execution commences. Various fields are initialized here and the various subeditors are instantiated and saved.

```
(defmethod (SC-Tool DoInit) (&aux p)
   (wait-cursor)
  (setq typed-noise-level 0
       typed-phase-jitter 0
     N 8
     M 1
      Constellation-Points (cp N)
      actual-coordinates nil
     SC-Type 'nc
      Dot-Size 6
      main-canvas (self 'FindNamedItem 'main-canvas)
        percent-error-box (self 'FindNamedItem 'percent-error-box)
      nop-box (self 'FindNamedItem 'nop-box)
      misses-box (self 'FindNamedItem 'misses-box)
     nop 0
     misses 0
     ConstEd (GetNewEditor 'ChooseConstellation)
      DistEd (GetNewEditor 'ChooseDistribution)
     AWGNEd1 (GetNewEditor 'AboutAWGN1)
     AWGNEd2 (GetNewEditor 'AboutAWGN2)
     AWGNEd3 (GetNewEditor 'AboutAWGN3)
      AboutConstEd1 (GetNewEditor 'AboutConstellationEditor1)
      AboutConstEd2 (GetNewEditor 'AboutConstellationEditor2)
      AboutConstEd3 (GetNewEditor 'AboutConstellationEditor3)
       AboutDistEd1 (GetNewEditor 'AboutDistributionEditor1)
       AboutDistEd2 (GetNewEditor 'AboutDistributionEditor2)
       AboutDistEd3 (GetNewEditor 'AboutDistributionEditor3)
       AboutDistEd4 (GetNewEditor 'AboutDistributionEditor4)
      simulation-on nil
      canvas-region (!newrgn)
       infinite-region (!newrgn)
      rectangle (Rect 'new 0 0 500 500))
  (setq width (- (setq canvas-right (main-canvas 'right))
             (setq canvas-left (main-canvas 'left))))
  (setq height (- (setq canvas-bottom (main-canvas 'bottom))
             (setq canvas-top (main-canvas 'top))))
  (!SetRectrgn canvas-region canvas-left canvas-top canvas-right canvas-bottom)
  (!SetRectrgn infinite-region 0 0 999 999)
 (self 'SetAboutString "About SC Design Tool...")
 (self 'AddSon ConstEd)
 (self 'AddSon DistEd)
 (self 'AddSon AWGNEd1)
 (self 'AddSon AWGNEd2)
 (self 'AddSon AWGNEd3)
 (self 'AddSon AboutConstEd1)
 (self 'AddSon AboutConstEd2)
 (self 'AddSon AboutConstEd3)
 (self 'AddSon AboutDistEd1)
 (self 'AddSon AboutDistEd2)
 (self 'AddSon AboutDistEd3)
 (self 'AddSon AboutDistEd4)
   (normal-cursor))
```

The following code turns the clipping on inside the main drawing area this is essential to do before any graphics operations so that other areas of the main panel are not affected by the graphics operations.

```
(defmethod (SC-Tool clip) ()
  (!SetClip canvas-region))
```

The following code turns the clipping off.

```
(defmethod (SC-Tool unclip) ()
   (!SetClip infinite-region))
```

The following code is executed when the main panel is closed; the various editors are garbage-collected and control is returned to the LISP system.

```
(defmethod (SC-Tool DoClose) ()
  (DoFreeEditor ConstEd)
  (DoFreeEditor DistEd)
  (DoFreeEditor AWGNEd1)
  (DoFreeEditor AWGNEd2)
  (DoFreeEditor AWGNEd3)
  (DoFreeEditor AboutConstEd1)
  (DoFreeEditor AboutConstEd2)
  (DoFreeEditor AboutConstEd3)
  (DoFreeEditor AboutDistEd1)
  (DoFreeEditor AboutDistEd2)
  (DoFreeEditor AboutDistEd3)
  (DoFreeEditor AboutDistEd4)
   (self 'exittolisp)
  (DoFreeEditor self))
```

The following code executes when the user selects the "About SC Tool..." help item from the menu. This code invokes the subeditor that gives the user some information about this program. When the user clicks "OK" control is returned to the calling/father editor and the execution continues.

```
(defmethod (SC-Tool DoAbout) (&aux aboutEd)
  (setq aboutEd (GetNewEditor 'AboutSCTool))
  (self 'AddSon aboutEd)
   (PromptModalEditor aboutEd)
   (DoFreeEditor aboutEd))
```

```
(defclass (AboutSCTool Editor))

(defmethod (AboutSCTool DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutSCTool OK) (&rest l) (self 'ExitToLisp))
```

The following code executes when the user selects the AWGN help item from the menu. This code invokes the subeditor that gives the user some information about AWGN. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the AWGN help are displayed. The AWGN help screens number 3 in total and each one is modelled here as a separate subeditor.

```
(defmethod (SC-Tool DoAboutAWGN1) (&rest 1)
  (PromptModalEditor AWGNEd1))

(defclass (AboutAWGN1 Editor))

(defmethod (AboutAWGN1 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutAWGN1 OK) (&rest l)
   (self 'ExitToLisp))

(defmethod (AboutAWGN1 NEXT) (&rest l)
  (send edFather 'DoAboutAWGN2)
   (self 'ExitToLisp))
```

The following code executes when the user selects the AWGN help item from either the menu or by a click on the corresponding string in the main panel. This code invokes the second AWGN help subeditor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the AWGN help are invoked.

```
(defmethod (SC-Tool DoAboutAWGN2) (&rest 1)
  (PromptModalEditor AWGNEd2))

(defclass (AboutAWGN2 Editor))

(defmethod (AboutAWGN2 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutAWGN2 OK) (&rest l)
   (self 'ExitToLisp))

(defmethod (AboutAWGN2 PREVIOUS) (&rest l)
  (send edFather 'DoAboutAWGN1)
   (self 'ExitToLisp))

(defmethod (AboutAWGN2 NEXT) (&rest l)
  (send edFather 'DoAboutAWGN3)
   (self 'ExitToLisp))
```

The following code executes when the user selects the AWGN help item from either the menu or by a click on the corresponding string in the main panel. This code invokes the third AWGN help subeditor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the AWGN help are invoked.

```
(defmethod (SC-Tool DoAboutAWGN3) (&rest 1)
  (PromptModalEditor AWGNEd3))

(defclass (AboutAWGN3 Editor))

(defmethod (AboutAWGN3 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutAWGN3 OK) (&rest l)
   (self 'ExitToLisp))

(defmethod (AboutAWGN3 PREVIOUS) (&rest l)
  (send edFather 'DoAboutAWGN2)
   (self 'ExitToLisp))
```

The following code executes when the user selects the constellation editor help item from the menu. This code invokes the subeditor that gives the user some information about the constellation editor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the constellation editor help are displayed. The constellation editor help screens number 3 in total and each one is modelled here as a separate subeditor.

```
(defmethod (SC-Tool DoAboutConstellationEditor1) (&rest 1)
  (PromptModalEditor AboutConstEd1))

(defclass (AboutConstellationEditor1 Editor))

(defmethod (AboutConstellationEditor1 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutConstellationEditor1 OK) (&rest l)
   (self 'ExitToLisp))

(defmethod (AboutConstellationEditor1 NEXT) (&rest l)
  (send edFather 'DoAboutConstellationEditor2)
   (self 'ExitToLisp))
```

The following code executes when the user selects the constellation editor help item from the menu. This code invokes the second constellation-editor help subeditor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the AWGN help are invoked. The constellation editor help screens number 3 in total and each one is modelled as a separate subeditor.

```
(defmethod (SC-Tool DoAboutConstellationEditor2) (&rest 1)
  (PromptModalEditor AboutConstEd2))

(defclass (AboutConstellationEditor2 Editor))

(defmethod (AboutConstellationEditor2 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutConstellationEditor2 OK) (&rest l)
   (self 'ExitToLisp))

(defmethod (AboutConstellationEditor2 PREVIOUS) (&rest l)
  (send edFather 'DoAboutConstellationEditor1)
   (self 'ExitToLisp))
```

```
(defmethod (AboutConstellationEditor2 NEXT) (&rest l)
  (send edFather 'DoAboutConstellationEditor3)
   (self 'ExitToLisp))
```

The following code executes when the user selects the constellation editor help item from the menu. This code invokes the third constellation-editor help subeditor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the constellation-editor help are invoked. The constellation editor help screens number 3 in total and each one is modelled as a separate subeditor.

```
(defmethod (SC-Tool DoAboutConstellationEditor3) (&rest 1)
  (PromptModalEditor AboutConstEd3))

(defclass (AboutConstellationEditor3 Editor))

(defmethod (AboutConstellationEditor3 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutConstellationEditor3 OK) (&rest l)
   (self 'ExitToLisp))

(defmethod (AboutConstellationEditor3 PREVIOUS) (&rest l)
  (send edFather 'DoAboutConstellationEditor2)
   (self 'ExitToLisp))
```

The following code executes when the user selects the distribution editor help item from the menu. This code invokes the subeditor that gives the user some information about the distribution editor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the distribution editor help are displayed. The distribution editor help screens number 4 in total and each one is modelled here as a separate subeditor.

```
(defmethod (SC-Tool DoAboutDistributionEditor1) (&rest 1)
  (PromptModalEditor AboutDistEd1))

(defclass (AboutDistributionEditor1 Editor))

(defmethod (AboutDistributionEditor1 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutDistributionEditor1 OK) (&rest l)
   (self 'ExitToLisp))

(defmethod (AboutDistributionEditor1 NEXT) (&rest l)
  (send edFather 'DoAboutDistributionEditor2)
   (self 'ExitToLisp))
```

The following code executes when the user selects the distribution editor help item from the menu. This code invokes the second distribution-editor help subeditor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the distribution editor help are invoked. The distribution editor help screens number 4 in total and each one is modelled as a separate subeditor.

```
(defmethod (SC-Tool DoAboutDistributionEditor2) (&rest 1)
  (PromptModalEditor AboutDistEd2))
```

```
(defclass (AboutDistributionEditor2 Editor))

(defmethod (AboutDistributionEditor2 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutDistributionEditor2 OK) (&rest l)
  (self 'ExitToLisp))

(defmethod (AboutDistributionEditor2 PREVIOUS) (&rest l)
  (send edFather 'DoAboutDistributionEditor1)
  (self 'ExitToLisp))

(defmethod (AboutDistributionEditor2 NEXT) (&rest l)
  (send edFather 'DoAboutDistributionEditor3)
  (self 'ExitToLisp))
```

The following code executes when the user selects the distribution editor help item from the menu. This code invokes the third distribution-editor help subeditor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the distribution editor help are invoked. The distribution editor help screens number 4 in total and each one is modelled as a separate subeditor.

```
(defmethod (SC-Tool DoAboutDistributionEditor3) (&rest 1)
  (PromptModalEditor AboutDistEd3))

(defclass (AboutDistributionEditor3 Editor))

(defmethod (AboutDistributionEditor3 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutDistributionEditor3 OK) (&rest l)
  (self 'ExitToLisp))

(defmethod (AboutDistributionEditor3 PREVIOUS) (&rest l)
  (send edFather 'DoAboutDistributionEditor2)
  (self 'ExitToLisp))

(defmethod (AboutDistributionEditor3 NEXT) (&rest l)
  (send edFather 'DoAboutDistributionEditor4)
  (self 'ExitToLisp))
```

The following code executes when the user selects the distribution editor help item from the menu. This code invokes the forth distribution-editor help subeditor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the distribution editor help are invoked. The distribution editor help screens number 4 in total and each one is modelled as a separate subeditor.

```
(defmethod (SC-Tool DoAboutDistributionEditor4) (&rest 1)
  (PromptModalEditor AboutDistEd4))

(defclass (AboutDistributionEditor4 Editor))

(defmethod (AboutDistributionEditor4 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutDistributionEditor4 OK) (&rest l)
  (self 'ExitToLisp))
```

```
(defmethod (AboutDistributionEditor4 PREVIOUS) (&rest l)
  (send edFather 'DoAboutDistributionEditor3)
  (self 'ExitToLisp))
```

The following code executes when the user selects the phase jitter help item from the menu. This code invokes the subeditor that gives the user some information about phase jitter. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the phase jitter help are displayed.

```
(defmethod (SC-Tool DoAboutPhaseJitter) (&rest 1)
  (prog (AboutEd)
      (setq aboutEd (GetNewEditor 'AboutPhaseJitter))
     (self 'AddSon aboutEd)
      (PromptModalEditor aboutEd)
      (DoFreeEditor aboutEd)))

(defclass (AboutPhaseJitter Editor))

(defmethod (AboutPhaseJitter DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutPhaseJitter OK) (&rest l) (self 'ExitToLisp))
```

The following code executes when the user selects the SC Design tool help item from the menu. This code invokes the subeditor that gives the user some information about the SC Design tool. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the SC Design tool help are displayed. The SC Design Tool help screens number 2 in total and each one is modelled here as a separate subeditor.

```
(defmethod (SC-Tool DoAboutSCDesign1) (&rest 1)
  (prog (AboutEd)
      (setq aboutEd (GetNewEditor 'AboutSCDesign1))
     (self 'AddSon aboutEd)
      (PromptModalEditor aboutEd)
      (DoFreeEditor aboutEd)))

(defclass (AboutSCDesign1 Editor))

(defmethod (AboutSCDesign1 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutSCDesign1 OK) (&rest l) (self 'ExitToLisp))

(defmethod (AboutSCDesign1 NEXT) (&rest l)
  (send edFather 'DoAboutSCDesign2)
   (self 'ExitToLisp))
```

The following code executes when the user selects the SC Design tool help item from the menu. This code invokes the second SC Design tool help subeditor. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the SC Design Tool help are invoked. The SC Design tool help screens number 2 in total and each one is modelled as a separate subeditor.

```
(defmethod (SC-Tool DoAboutSCDesign2) (&rest 1)
  (prog (AboutEd)
      (setq aboutEd (GetNewEditor 'AboutSCDesign2))
      (self 'AddSon aboutEd)
       (PromptModalEditor aboutEd)
       (DoFreeEditor aboutEd)))

(defclass (AboutSCDesign2 Editor))

(defmethod (AboutSCDesign2 DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutSCDesign2 OK) (&rest l)
  (send edFather 'ExitToLisp)
   (self 'ExitToLisp))

(defmethod (AboutSCDesign2 PREV) (&rest l)
 (send edFather 'DoAboutSCDesign1)
   (self 'ExitToLisp))
```

The following code executes when the user selects the acknowledgements help item from the menu. This code invokes the subeditor that gives the user some information about the acknowledgements. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the acknowledgements help are displayed.

```
(defmethod (SC-Tool DoAboutAcknowledgements) (&rest 1)
  (prog (AboutEd)
      (setq aboutEd (GetNewEditor 'AboutAcknowledgements))
      (self 'AddSon aboutEd)
       (PromptModalEditor aboutEd)
       (DoFreeEditor aboutEd)))

(defclass (AboutAcknowledgements Editor))

(defmethod (AboutAcknowledgements DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutAcknowledgements OK) (&rest l) (self 'ExitToLisp))
```

The following code executes when the user selects the references help item from the menu. This code invokes the subeditor that gives the user some information about the references. When the user clicks "Quit" control returns to the calling/father editor, otherwise if "Next" or "Previous" are clicked then the corresponding continuations of the references help are displayed.

```
(defmethod (SC-Tool DoAboutReferences) (&rest 1)
  (prog (AboutEd)
      (setq aboutEd (GetNewEditor 'AboutReferences))
      (self 'AddSon aboutEd)
       (PromptModalEditor aboutEd)
       (DoFreeEditor aboutEd)))

(defclass (AboutReferences Editor))

(defmethod (AboutReferences DoEvent) (theEvent)
  (self 'DoModalDialogEvent theEvent))

(defmethod (AboutReferences OK) (&rest l) (self 'ExitToLisp))
```

The following code allows the user to type in a noise level, or alternatively to modify the noise level by clicking or dragging a scroll bar. When a new noise level is typed in the scroll bar updates itself accordingly, and conversely, if the scroll bar is modified then the (decimal) noise level updates is updated on the screen as well. Thus we achieve the effect of two indicators locked with each other on the same value.

```
(defmethod (SC-Tool type-noise-level) (itm)
  (!setCtlValue ((self 'FindNamedItem 'scrolled-noise-level) 'itemControl)
                ((self 'FindNamedItem 'typed-noise-level) 'GetNum))
  ((self 'FindNamedItem 'scrolled-noise-level) 'Display))

(defmethod (SC-Tool scroll-noise-level) (bar)
  (setq typed-noise-level (!GetCtlValue (bar 'itemControl)))
  ((self 'FindNamedItem 'typed-noise-level) 'SetNum typed-noise-level)
  ((self 'FindNamedItem 'typed-noise-level) 'Display))
```

The following code allows the user to type in a phase jitter level, or alternatively to modify the phase jitter level by clicking or dragging a scroll bar. When a new phase jitter level is typed in the scroll bar updates itself accordingly, and conversely, if the scroll bar is modified then the (decimal) phase jitter level updates is updated on the screen as well. Thus we achieve the effect of two indicators locked with each other on the same value.

```
(defmethod (SC-Tool type-phase-jitter) (itm)
  (!setCtlValue ((self 'FindNamedItem 'scrolled-phase-jitter) 'itemControl)
                ((self 'FindNamedItem 'typed-phase-jitter) 'GetNum))
  ((self 'FindNamedItem 'scrolled-phase-jitter) 'Display))

(defmethod (SC-Tool scroll-phase-jitter) (bar)
  (setq typed-phase-jitter (!GetCtlValue (bar 'itemControl)))
  ((self 'FindNamedItem 'typed-phase-jitter) 'SetNum typed-phase-jitter)
  ((self 'FindNamedItem 'typed-phase-jitter) 'Display))
```

The following code turns on the simulation; from now on signals will be generated at random and plotted, yielding the error-rate for the current constellation with the specified noise parameters.

```
(defmethod (SC-Tool Simulate) (&rest 1)
  (setq Simulation-on t))
```

The following code turns off and stops the ongoing simulation.

```
(defmethod (SC-Tool suspend-simulation) (&rest 1)
  (setq Simulation-on nil))
```

The following code (re)draws the signal constellation.

```
(defmethod (SC-Tool redraw-main-canvas) (&rest 1)
  (let (ds1 ds2 x y image)
        (wait-cursor)
      (!EraseRect main-canvas)
       (self 'clip)
      (!EraseRect main-canvas)
        (!MoveTo (+ canvas-left (truncate (/ width 2))) canvas-top)
        (!LineTo (+ canvas-left (truncate (/ width 2))) (+ canvas-top height))
        (!MoveTo canvas-left (truncate (+ canvas-top (/ height 2))))
        (!LineTo (+ canvas-left width) (truncate (+ canvas-top (/ height 2))))
       (setq ds1 (truncate (/ Dot-Size 2)))
       (setq ds2 (round (/ Dot-Size 2)))
      (cond ((not actual-coordinates)
          (setq actual-coordinates (main-canvas-affine
                              Constellation-Points
                                 canvas-left canvas-top width height))))
      (dolist (p actual-coordinates)
          (setq x (car p))
          (setq y (cadr p))
             (!SetRect rectangle (- x ds1) (- y ds1) (+ x ds2) (+ y ds2))
           (!FillOval rectangle black))
      (self 'unclip)
        (normal-cursor)))
```

The following code clears the signal constellation. This is useful when the user wishes to clear all the data points plotted so far.

```
(defmethod (SC-Tool clear-main-canvas) (&rest 1)
  (setq nop 0)
  (setq misses 0)
   (nop-box 'SetString (format nil "~A" 0))
   (nop-box 'display)
    (percent-error-box 'SetString (format nil "~A" 0.0))
     (percent-error-box 'display)
   (misses-box 'SetString (format nil "~A" 0.0))
   (misses-box 'display)
    (self 'redraw-main-canvas))
```

The following code translates the signal constellation from normalized coordinates (on the unit disk) to absolute screen coordinates.

```
(defun main-canvas-affine (pts canvas-left canvas-top width height)
   (let ((ans nil))
       (dolist (p (reverse pts))
          (setq ans
                (cons (list
                      (+ canvas-left
                          (truncate (+ (/ width 2)
                                       (* (car p) (/ width 2.2)))))
                      (+ canvas-top
                          (truncate (+ (/ height 2)
                                       (* (cadr p) (/ height 2.2))))))
                    ans)))
     ans))
```

The following code is executed whenever there is no other activity going on. It is used to carry on the simulation by generating random signals and plotting them against the current signal constellation. The various fields and indicators are also updated to reflect the resulting error statistics.

```
(defmethod (SC-Tool DoIdle) ()
    (normal-cursor)
  (cond (simulation-on
          (setq selected-signal (random (length actual-coordinates)))
          (setq actual-pt (nth selected-signal actual-coordinates))
           (setq original-pt (nth selected-signal Constellation-Points))
           (setq vector-plus-noise (add-noise original-pt actual-pt self DistEd))
           (setq x (round (car vector-plus-noise)))
           (setq y (round (cadr vector-plus-noise)))
         (self 'clip)
```

```
        (!SetRect rectangle x y (+ x 1) (+ y 1))
        (!FillOval rectangle black)
        (self 'unclip)
          (setq dist (+ (square (- (car actual-pt) x))
                        (square (- (cadr actual-pt) y))))
        (setq min-dist dist)
         (dolist (other-pt actual-coordinates)
                (cond ((not (eq actual-pt other-pt))
                   (setq min-dist
                             (min (+ (square (- (car other-pt) x))
                                     (square (- (cadr other-pt) y)))
                              min-dist)))))
          (cond ((< min-dist dist) (setq misses (1+ misses))))
        (setq nop (1+ nop))
        (cond ((eq 0 (rem nop 10))
               (nop-box 'SetString (format nil "~A" nop))
             (nop-box 'display)
               (misses-box 'SetString (format nil "~A" misses))
             (misses-box 'display)
               (percent-error-box 'SetString
                           (format nil "~A"
                                   (* 100 (/ misses nop))))
                  (percent-error-box 'display))))))
```

The following code invokes the signal constellation editor as a subeditor.

```
(defmethod (SC-Tool DoChangeConstellation) (&rest 1)
  (PromptModalEditor ConstEd)
  (cond ((eq 'do-it (ConstEd 'action))
        (setq N (val (ConstEd 'n)))
        (setq M (val (ConstEd 'm)))
        (setq Constellation-Points (ConstEd 'pointset))
        (setq Dot-Size (val (ConstEd 'ds)))
        (setq SC-Type (ConstEd 'mode)))))
```

The following code invokes the distribution editor as a subeditor.

```
(defmethod (SC-Tool DoChangeDistribution) (&rest 1)
  (PromptModalEditor DistEd)
  (cond ((eq 'do-it (DistEd 'action))
         (assign (self 'FindNamedItem 'typed-noise-level)
               (case (DistEd 'dist-type)
                   (g (DistEd 'variance))
                     (u (DistEd 'interval))))
           (self 'type-noise-level (self 'FindNamedItem 'typed-noise-level)))))
```

The following code corrupts a given signal by adding random noise and phase jitter to it according to the parameters specified by the user. The AWGN and the phase jitter are superimposed (added) onto the signal.

```
(defun add-noise (original-pt actual-pt SC-Ed DistEd)
  (setq AWGN-vector (case (DistEd 'dist-type)
                     (g (gd (SC-Ed 'typed-noise-level)))
                      (u (ud (SC-Ed 'typed-noise-level)))))
   (setq phase-jitter-angle
        (* (cond ((< (rnd) 0.5) 1) (t -1))
         (rad (case (DistEd 'dist-type)
                  (g (abs (* (SC-Ed 'typed-phase-jitter) (log (rnd)))))
                   (u (- (random (1+ (SC-Ed 'typed-phase-jitter))) 1))))))
  (setq cos-angle (cos phase-jitter-angle))
  (setq sin-angle (sin phase-jitter-angle))
  (setq old-x (- (- (car actual-pt) (SC-Ed 'canvas-left)) (/ (SC-Ed 'width) 2)))
  (setq old-y (- (- (cadr actual-pt) (SC-Ed 'canvas-top)) (/ (SC-Ed 'height) 2)))
  (setq new-x (- (* old-x cos-angle) (* old-y sin-angle)))
  (setq new-y (+ (* old-x sin-angle) (* old-y cos-angle)))
  (list (+ new-x (car AWGN-vector) (SC-Ed 'canvas-left) (/ (SC-Ed 'width) 2))
      (+ new-y (cadr AWGN-vector) (SC-Ed 'canvas-top) (/ (SC-Ed 'height) 2))))
```

The following class defines the constellation editor panel. It contains various fields/parameters that the user may modify and other that are computed and stored away for future use.

```
(defclass (ChooseConstellation Editor)
 (IVS mode
      n                         ; constellation N parameter
      m                         ; constellation M parameter
      h                         ; horizontal spacing for the rectangular grid
      v                         ; vertical spacing for the rectangular grid
      c                         ; number of circles for the polar grid
      r                         ; number of rays for the polar grid
      grid                      ; field toggling the grid on/off
      rectangular               ; field indicating rectangular grid
      polar                     ; field indicating polar grid
      grid-type                 ; type of the grid
      grid-on                   ; boolean flag indicating whether grid is on
      p1                        ; first grid parameter field
      p2                        ; second grid parameter field
      name-p1                   ; name of the p1 field instance
      name-p2                   ; name of the p2 field instance
      ds                        ; the size of the signal dot
      n-on-a-circle             ; field indicating constellation type N on a circle
      n-by-m-rectangle          ; field indicating constellation type N by M rectangle
      n-on-m-circles            ; field indicating constellation type N on M circles
      user-specified            ; field indicating constellation is user specified
      canvas                    ; main drawing area for the constellation editor
      width                     ; width of the main drawing area
      height                    ; height of the main drawing area
      canvas-right              ; right corner of drawing area
      canvas-left               ; left corner of drawing area
      canvas-top                ; top corner of drawing area
      canvas-bottom             ; bottom corner of drawing area
      pointset                  ; the constellation points
      action                    ; boolean indicator for OK vs. CANCEL click
      rectangle                 ; a temporary record that defines a rectangular region
      canvas-region             ; clipping region for the main drawing area
      infinite-region           ; used to turn clipping off
      highlighted-point         ; the currently highlighted constellation point
      highlighted-location      ; actual screen location of highlighted point
   ))
```

The following code is executed when the constellation editor is instantiated. Various fields are initialized here, and some constants are computed as well.

```
(defmethod (ChooseConstellation DoInit) (&aux secs strDate group)
  (setq m (self 'FindNamedItem 'm)
      n (self 'FindNamedItem 'n)
      ds (self 'FindNamedItem 'ds)
      pointset (cp 8)
         n-by-m-rectangle  (self 'FindNamedItem 'n-by-m-rectangle)
         n-on-a-circle (self 'FindNamedItem 'n-on-a-circle)
         n-on-m-circles (self 'FindNamedItem 'n-on-m-circles)
        user-specified (self 'FindNamedItem 'user-specified)
       rectangular (self 'FindNamedItem 'rectangular)
       polar (self 'FindNamedItem 'polar)
       p1 (self 'FindNamedItem 'p1)
       p2 (self 'FindNamedItem 'p2)
       name-p1 (self 'FindNamedItem 'name-p1)
       name-p2 (self 'FindNamedItem 'name-p2)
      h 8
      v 8
      c 8
      r 16
       mode 'n-on-a-circle
       grid-type 'rectangular
       grid-on nil
      canvas (self 'FindNamedItem 'canvas)
          group (list n-on-a-circle n-by-m-rectangle
                   n-on-m-circles user-specified)
        group2 (list rectangular polar)
       rectangle (Rect 'new 0 0 500 500)
       canvas-region (!newrgn)
        infinite-region (!newrgn)
       highlighted-point nil
        highlighted-location nil)
   (n-on-a-circle 'itemLinks group)
   (n-by-m-rectangle 'itemLinks group)
   (n-on-m-circles 'itemLinks group)
   (user-specified 'itemLinks group)
   (rectangular 'itemlinks group2)
   (polar 'itemlinks group2)
   (self 'TextActivate n)
   (setq width (- (setq canvas-right (canvas 'right))
              (setq canvas-left (canvas 'left))))
   (setq height (- (setq canvas-bottom (canvas 'bottom))
              (setq canvas-top (canvas 'top))))
   (!SetRectrgn canvas-region canvas-left canvas-top canvas-right canvas-bottom)
   (!SetRectrgn infinite-region 0 0 999 999)
   (assign ds 6))
```

The following code is executed when the constellation editor panel is closed; control is returned to the LISP system.

```
(defmethod (ChooseConstellation DoClose) ()
   (self 'do-it))
```

The following code allows the user to delete from the current signal constellation the point that is currently highlighted.

```
(defmethod (ChooseConstellation delete-point) (&rest 1)
  (let (x y ds1 ds2)
      (self 'clip)
      (cond (highlighted-point
            (self 'unhighlight highlighted-location (+ 1 (val ds)))
            (setq ds1 (round (/ (val ds) 2)))
            (setq ds2 (truncate (/ (val ds) 2)))
            (setq x (car highlighted-location))
            (setq y (cadr highlighted-location))
            (!SetRect rectangle (- x ds1) (- y ds1) (+ x ds2) (+ y ds2))
            (!InvertOval rectangle)
            (setq pointset (delq highlighted-point pointset))
            (setq highlighted-point nil)
            (setq highlighted-location nil)))
      (self 'unclip)))
```

The following code allows the user to add to the current signal constellation a new point; the very next mouse click will determine the position of this new point.

```
(defmethod (ChooseConstellation add-point) (&rest 1)
  (let (x y ds1 ds2 mouse-horiz mouse-vert MouseLocationPoint)
      (self 'clip)
      (normal-cursor)
      (do nil ((!button)) nil)
      (!GetMouse (setq MouseLocationPoint (Point 'new)))
      (setq mouse-horiz (MouseLocationPoint 'horizontal))
      (setq mouse-vert  (MouseLocationPoint 'vertical))
      (cond ((and (< canvas-left mouse-horiz) (< mouse-horiz canvas-right)
                  (< canvas-top mouse-vert) (< mouse-vert canvas-bottom))
             (setq x (* (- (- mouse-horiz canvas-left) (/ width 2))
                        (/ 2.2 width)))
             (setq y (* (- (- mouse-vert canvas-top) (/ height 2))
                        (/ 2.2 height)))
             (setq pointset (cons (list x y) pointset))
             (self 'unhighlight highlighted-location (+ 1 (val ds)))
             (setq highlighted-point nil)
             (setq highlighted-location nil)
             (setq ds1 (round (/ (val ds) 2)))
             (setq ds2 (truncate (/ (val ds) 2)))
             (!SetRect rectangle (- mouse-horiz ds1) (- mouse-vert ds1)
                        (+ mouse-horiz ds2) (+ mouse-vert ds2))
             (!FillOval rectangle black)))
      (self 'unclip)))
```

The following code is invoked whenever the main drawing area of the constellation editor is clicked; the closest signal point to the click is determined and is highlighted for future operations upon it.

```
(defmethod (ChooseConstellation Click-Canvas) (itm pt wn mod)
  (let ((max-dist 999999) (target nil) (actual nil) x y)
      (dolist (p pointset)
            (setq x (+ canvas-left (truncate (+ (/ width 2)
                                          (* (car p) (/ width 2.2))))))
            (setq y (+ canvas-top (truncate (+ (/ height 2)
                                          (* (cadr p) (/ height 2.2))))))
            (cond ((< (setq tmp (+ (square (- (pt 'horizontal) x))
                                   (square (- (pt 'vertical) y))))
                      max-dist)
                   (setq max-dist tmp)
                   (setq target p)
                   (setq actual (list x y)))))
      (self 'unhighlight highlighted-location (+ 1 (val ds)))
     (setq highlighted-point target)
     (setq highlighted-location actual)
      (self 'highlight highlighted-location (+ 1 (val ds)))
       (normal-cursor)))
```

The following code highlights the given point in the current constellation.

```
(defmethod (ChooseConstellation highlight) (pt ds)
  (cond (pt
        (self 'clip)
         (ellipse (car pt) (cadr pt) ds ds t)
          (self 'unclip))))
```

The following code un-highlights the given point in the current constellation.

```
(defmethod (ChooseConstellation unhighlight) (pt ds)
  (cond (pt
        (self 'clip)
         (ellipse (car pt) (cadr pt) ds ds t)
          (self 'unclip))))
```

The following code turns the clipping on inside the main drawing area; this is essential to do before any graphics operations so that other areas of the main panel are not affected by the graphics operations.

```
(defmethod (ChooseConstellation clip) ()
   (!SetClip canvas-region))
```

The following code turns the clipping off.

```
(defmethod (ChooseConstellation unclip) ()
    (!SetClip  infinite-region))
```

The following code turns on the rectangular editing grid.

```
(defmethod (ChooseConstellation do-rectangular) (itm)
  (cond
     ((= (!GetCtlValue (itm 'itemControl)) 1) (setq grid-type 'rectangular))
     (t (!SetCtlValue (itm 'itemControl) 1)))
   (name-p1 'SetString "    Horiz:")
    (name-p1 'Display)
   (name-p2 'SetString "     Vert:")
    (name-p2 'Display)
    (p1 'SetString (format nil "~A" v))
    (p1 'Display)
    (p2 'SetString (format nil "~A" h))
    (p2 'Display)
    (self 'redraw-canvas))
```

The following code turns on the polar editing grid.

```
(defmethod (ChooseConstellation do-polar) (itm)
  (cond
     ((= (!GetCtlValue (itm 'itemControl)) 1) (setq grid-type 'polar))
     (t (!SetCtlValue (itm 'itemControl) 1)))
  (name-p1 'SetString "Circles:")
  (name-p1 'Display)
 (name-p2 'SetString "   Rays:")
  (name-p2 'Display)
  (p1 'SetString (format nil "~A" c))
  (p1 'Display)
  (p2 'SetString (format nil "~A" r))
  (p2 'Display)
  (self 'redraw-canvas))
```

The following code is invoked when the user clicks the "Redraw" button.

```
(defmethod (ChooseConstellation do-redraw-grid) (&rest 1)
  (self 'redraw-canvas))
```

The following code initiates the redrawing of the editing grid according to the spacing parameters specified by the user.

```
(defmethod (ChooseConstellation redraw-grid) ()
  (self 'clip)
   (wait-cursor)
  (cond (grid-on
       (case grid-type
            (polar (draw-polar-grid c r width height
                         canvas-right canvas-left
                         canvas-top canvas-bottom))
            (rectangular (draw-rectangular-grid v h width height
                              canvas-right canvas-left
                              canvas-top canvas-bottom))
           (t nil))))
  (self 'unclip)
   (normal-cursor))
```

The following code actually draws the polar editing grid according the the circle and ray counts/parameters specified by the user.

```
(defun draw-polar-grid (c r width height canvas-right canvas-left canvas-top canvas-bottom)
  (dotimes (rad c)
          (ellipse (truncate (+ canvas-left (/ width 2)))
                (truncate (+ canvas-top (/ height 2)))
                 (round (* (/ (1+ rad) c) (/ width 2)))
                  (round (* (/ (1+ rad) c) (/ height 2)))))
  (dotimes (ray r)
          (!MoveTo (truncate (+ canvas-left (/ width 2)))
                (truncate (+ 1 canvas-top (/ height 2))))
          (!LineTo (truncate (+ canvas-left (truncate (/ width 2))
                         (* width (cos (* 2 (pi) (/ ray r))))))
                (truncate (+ 1 canvas-top (truncate (/ height 2))
                          (* height (sin (* 2 (pi) (/ ray r)))))))))
```

The following code actually draws the rectangular editing grid according to the horizontal and vertical counts/parameters specified by the user.

```
(defun draw-rectangular-grid (v h width height canvas-right canvas-left canvas-top canvas-
bottom)
  (dotimes (x (1+ h))
          (!MoveTo (round (+ canvas-left (* (- width 1) (/ x h)))) canvas-top)
          (!LineTo (round (+ canvas-left (* (- width 1) (/ x h))))
              (+ canvas-top height)))
  (dotimes (y (1+ v))
          (!MoveTo canvas-left (round (+ canvas-top (* (- height 1) (/ y v)))))
        (!LineTo (+ canvas-left width)
                (round (+ canvas-top (* (- height 1) (/ y v)))))))
```

The following code toggles the editing grid on/off.

```
(defmethod (ChooseConstellation do-grid-on) (itm)
  (setq grid-on (not grid-on))
  (self 'redraw-canvas))
```

The following code makes the cursor appear like a watch; this is useful to indicate to the user that some time-consuming computation is taking place.

```
(defun wait-cursor () (!SetCursor (!GetCursor 4)))
```

The following code resets the cursor to its normal appearance.

```
(defun normal-cursor () (!InitCursor))
```

The following code (re)draws the main drawing area of the signal constellation editor.

```
(defmethod (ChooseConstellation redraw-canvas) (&rest 1)
  (let (ds1 ds2 x y image)
        (wait-cursor)
      (!EraseRect canvas)
        (setq highlighted-point nil)
        (setq highlighted-location nil)
        (self 'redraw-grid)
       (self 'clip)
        (!MoveTo (+ canvas-left (truncate (/ width 2))) canvas-top)
        (!LineTo (+ canvas-left (truncate (/ width 2))) (+ canvas-top height))
        (!MoveTo canvas-left (truncate (+ 4 (/ height 2))))
        (!LineTo (+ canvas-left width) (+ 4 (truncate (/ height 2))))
       (setq ds1 (truncate (/ (val ds) 2)))
       (setq ds2 (round (/ (val ds) 2)))
      (dolist (p pointset)
              (setq x (+ canvas-left (truncate (+ (/ width 2)
                                        (* (car p) (/ width 2.2))))))
              (setq y (+ canvas-top (truncate (+ (/ height 2)
                                       (* (cadr p) (/ height 2.2))))))
              (!SetRect rectangle (- x ds1) (- y ds1) (+ x ds2) (+ y ds2))
            (!FillOval rectangle black))
      (self 'unclip)
        (normal-cursor)))
```

The following code is invoked when the "N on a circle" constellation is chosen by the user via clicking the associated radio button. The main drawing area is then updated to reflect the new chosen constellation.

```
(defmethod (ChooseConstellation Mode-n-on-a-circle) (itm)
 (cond
    ((= (!GetCtlValue (itm 'itemControl)) 1) (setq mode 'n-on-a-circle))
    (t (!SetCtlValue (itm 'itemControl) 1)))
  (setq pointset (cp (val n)))
  (self 'redraw-canvas))
```

The following code is invoked when the "N by M rectangle" constellation is chosen by the user via clicking the associated radio button. The main drawing area is then updated to reflect the new chosen constellation.

```
(defmethod (ChooseConstellation Mode-n-by-m-rectangle) (itm)
 (cond
    ((= (!GetCtlValue (itm 'itemControl)) 1) (setq mode 'n-by-m-rectangle))
    (t (!SetCtlValue (itm 'itemControl) 1)))
  (setq pointset (rnm (val n) (val m)))
  (self 'redraw-canvas))
```

The following code is invoked when the "N on M circles" constellation is chosen by the user via clicking the associated radio button. The main drawing area is then updated to reflect the new chosen constellation.

```
(defmethod (ChooseConstellation Mode-n-on-m-circles) (itm &aux itemVal)
 (cond
    ((= (!GetCtlValue (itm 'itemControl)) 1) (setq mode 'n-on-m-circles))
    (t (!SetCtlValue (itm 'itemControl) 1)))
  (setq pointset (cnm (val n) (val m)))
  (self 'redraw-canvas))
```

The following code is invoked when the "user specified" constellation is chosen by the user via clicking the associated radio button. The user is next expected to define his own constellation by manually adding and deleting specific points to the constellation.

```
(defmethod (ChooseConstellation Mode-user-specified) (itm &aux itemVal)
 (cond
    ((= (!GetCtlValue (itm 'itemControl)) 1) (setq mode 'user-specified))
    (t (!SetCtlValue (itm 'itemControl) 1)))
  (self 'redraw-canvas))
```

The following code recomputes the current signal constellation from scratch; this is useful when the user has modified the current signal constellation by adding and deleting points, and would like to "undo" the modifications and revert to the original "canned" constellation with which the editing session was started.

```
(defmethod (ChooseConstellation Recompute-Const) (&rest 1)
 (selectq mode
              (n-on-a-circle (self 'Mode-n-on-a-circle n-on-a-circle))
              (n-by-m-rectangle (self 'Mode-n-by-m-rectangle n-by-m-rectangle))
              (n-on-m-circles (self 'Mode-n-on-m-circles n-on-m-circles))
            (user-specified (self 'Mode-user-specified user-specified))))
```

One of the following code is invoked when the user selects a particular "canned" signal constellation to manipulate. This functionality may also be invoked from the menu, as well as by clicking a radio button.

```
(defmethod (ChooseConstellation menu-n-on-a-circle) ()
   (n-on-a-circle 'twist)
   (self 'mode-n-on-a-circle n-on-a-circle))

(defmethod (ChooseConstellation menu-n-by-m-rectangle) ()
   (n-by-m-rectangle 'twist)
   (self 'mode-n-by-m-rectangle n-by-m-rectangle))
```

```
(defmethod (ChooseConstellation menu-n-on-m-circles) ()
   (n-on-m-circles 'twist)
   (self 'mode-n-on-m-circles n-on-m-circles))

(defmethod (ChooseConstellation menu-user-specified) ()
  (user-specified 'twist)
  (self 'mode-user-specified user-specified))
```

One of the following code is invoked when the selects a particular type of editing grid. This functionality may also be invoked from the menu, as well as by clicking a radio button.

```
(defmethod (ChooseConstellation menu-rectangular) ()
  (rectangular 'twist)
  (self 'do-rectangular rectangular))

(defmethod (ChooseConstellation menu-polar) ()
  (polar 'twist)
  (self 'do-polar polar))
```

The following code toggles the editing grid on/off; it may also be invoked from the menu, as well as by clicking a radio button.

```
(defmethod (ChooseConstellation menu-grid-on) ()
  ((self 'FindNamedItem 'Grid-Toggle) 'twist)
  (self 'do-grid-on (self 'FindNamedItem 'Grid-Toggle)))
```

The following code returns the values of the various parameters that may be set by the user.

```
(defmethod (ChooseConstellation ReturnN) (itm)
  (self 'TextActivate 'm)
  (m 'SelectText 0 32000))

(defmethod (ChooseConstellation ReturnM) (itm)
  (self 'TextActivate ds)
  (ds 'SelectText 0 32000))

(defmethod (ChooseConstellation ReturnDs) (itm)
  (self 'TextActivate n)
  (n 'SelectText 0 32000))
```

The following code is executed whenever there is no other activity going on. It is used to update the values of some user-settable parameters.

```
(defmethod (ChooseConstellation DoIdle) ()
  (case grid-type
      (polar (setq c (p1 'getnum))
           (setq r (p2 'getnum)))
      (rectangular (setq h (p1 'getnum))
              (setq v (p2 'getnum))))
  (cond ((< (val m) 1)
      (m 'SetNum 1)
       (m 'Display)))
  (cond ((< (val n) 1)
      (n 'SetNum 1)
       (n 'Display)))
  (cond ((< (val ds) 2)
    (ds 'SetNum 2)
      (ds 'Display))))
```

The following code distinguishes between a "CANCEL" and an "OK".

```
(defmethod (ChooseConstellation do-it) (&rest 1)
  (setq action 'do-it)
  (edFather 'actual-coordinates nil)
  (self 'ExitToLisp))

(defmethod (ChooseConstellation dont-do-it) (&rest 1)
  (setq action 'dont-do-it)
  (self 'ExitToLisp))

(defmethod (ChooseConstellation OK) (&rest 1)
   (self 'do-it))
```

The following code invokes the help screens for the constellation editor.

```
(defmethod (ChooseConstellation DoAboutConstellationEditor) (&rest 1)
  (edFather 'DoAboutConstellationEditor1))
```

The following class defines the distribution editor panel. It contains various fields/parameters that the user may modify and other that are computed and stored away for future use.

```
(defclass (ChooseDistribution  Editor)
  (IVS dist-type                  ; the type of the current distribution
       variance-box               ; panel field containing the variance
       variance-bar               ; panel scroll bar representing the variance
       variance                   ; variance of the Gaussian distribution
       interval                   ; interval of the uniform distribution
       nop                        ; fields containing the number of points
       trial-number               ; number of points
       canvas                     ; main drawing area for distribution editor
       canvas-region              ; clipping region for the main drawing area
       infinite-region            ; used to turn clipping off
       width                      ; width of the main drawing area
       height                     ; height of the main drawing area
       canvas-right               ; right corner of drawing area
       canvas-left                ; left corner of drawing area
       canvas-top                 ; top corner of drawing area
       canvas-bottom              ; bottom corner of drawing area
       distribution-on            ; boolean flag telling whether the simulation is on
       rectangle                  ; temporary rectangular region used for drawing
       action                     ; distinguishes between "OK" and "CANCEL" clicks
       total-x                    ; total of all the X coordinates
       total-y                    ; total of all the Y coordinates
       ave-x                      ; average of the X coordinates
       ave-y                      ; average of the Y coordinates
    ))
```

The following code is executed when the distribution editor is instantiated. Various fields are initialized here, and some constants are computed as well.

```
(defmethod (ChooseDistribution DoInit) (&aux secs strDate group)
  (setq nop (self 'FindNamedItem 'nop)
      canvas (self 'FindNamedItem 'canvas)
       u (self 'FindNamedItem 'u)
       g (self 'FindNamedItem 'g)
       nop (self 'FindNamedItem 'nop)
        ave-x (self 'FindNamedItem 'ave-x)
        ave-y (self 'FindNamedItem 'ave-y)
         variance-box (self 'FindNamedItem 'variance-box)
         variance-bar (self 'FindNamedItem 'variance-bar)
      variance 0
       interval 0
       group (list u g)
        rectangle (Rect 'new 0 0 500 500)
         distribution-on nil
         trial-number 0
       total-x 0
       total-y 0
       dist-type 'g
        canvas-region (!newrgn)
          infinite-region (!newrgn))
  (setq width (- (setq canvas-right (canvas 'right))
                 (setq canvas-left (canvas 'left))))
  (setq height (- (setq canvas-bottom (canvas 'bottom))
                 (setq canvas-top (canvas 'top))))
  (!SetRectrgn canvas-region canvas-left canvas-top canvas-right canvas-bottom)
  (!SetRectrgn infinite-region 0 0 999 999)
  (u 'itemLinks group)
  (g 'itemLinks group))
```

The following code turns the clipping on inside the main drawing area; this is essential to do before any graphics operations so that other areas of the main panel are not affected by the graphics operations.

```
(defmethod (ChooseDistribution clip) ()
  (!SetClip canvas-region))
```

The following code turns the clipping off.

```
(defmethod (ChooseDistribution unclip) ()
   (!SetClip infinite-region))
```

The following code is executed when the distribution editor panel is closed; control is returned to the LISP system.

```
(defmethod (ChooseDistribution DoClose) ()
   (self 'do-it))
```

The following code distinguishes between a "CANCEL" and an "OK".

```
(defmethod (ChooseDistribution do-it) (&rest 1)
  (setq action 'do-it)
  (self 'ExitToLisp))

(defmethod (ChooseDistribution dont-do-it) (&rest 1)
  (setq action 'dont-do-it)
  (self 'ExitToLisp))

(defmethod (ChooseDistribution OK) (&rest 1)
  (self 'noop))
```

One of the following code is invoked when the user selects a uniform distribution type. This functionality may also be invoked from the menu, as well as by clicking a radio button.

```
(defmethod (ChooseDistribution Mode-u) (itm)
 (cond
    ((= (!GetCtlValue (itm 'itemControl)) 1) (setq dist-type 'u))
    (t (!SetCtlValue (itm 'itemControl) 1)))
  ((self 'FindNamedItem 'parameter-name) 'SetString "Interval:")
  ((self 'FindNamedItem 'parameter-name) 'Display)
  (assign (self 'FindNamedItem 'variance-box) interval)
  ((self 'FindNamedItem 'variance-box) 'Display)
  (!setCtlValue ((self 'FindNamedItem 'variance-bar) 'itemControl) interval)
  ((self 'FindNamedItem 'variance-bar) 'Display)
  (self 'clear))
```

One of the following pieces of code is invoked when the user selects a Gaussian distribution type. This functionality may also be invoked from the menu, as well as by clicking a radio button.

```
(defmethod (ChooseDistribution Mode-g) (itm)
 (cond
    ((= (!GetCtlValue (itm 'itemControl)) 1) (setq dist-type 'g))
    (t (!SetCtlValue (itm 'itemControl) 1)))
  ((self 'FindNamedItem 'parameter-name) 'SetString "Variance:")
  ((self 'FindNamedItem 'parameter-name) 'Display)
  (assign (self 'FindNamedItem 'variance-box) variance)
  ((self 'FindNamedItem 'variance-box) 'Display)
  (!setCtlValue ((self 'FindNamedItem 'variance-bar) 'itemControl) variance)
  ((self 'FindNamedItem 'variance-bar) 'Display)
  (self 'clear))
```

One of the following code permits the distribution type to also be selected from the menu, as well as by clicking a radio button.

```
(defmethod (ChooseDistribution menu-mode-u) ()
  (u 'twist)
  (self 'mode-u u))

(defmethod (ChooseDistribution menu-mode-g) ()
  (g 'twist)
  (self 'mode-g g))
```

The following code (re)draws the main drawing area of the distribution editor. All previously plotted points are erased from the display.

```
(defmethod (ChooseDistribution redraw-dist) (&rest 1)
   (wait-cursor)
  (!EraseRect canvas)
   (!MoveTo (+ canvas-left (truncate (/ width 2))) canvas-top)
   (!LineTo (+ canvas-left (truncate (/ width 2))) (+ canvas-top height))
   (!MoveTo canvas-left (+ canvas-top (truncate (/ height 2))))
   (!LineTo (+ canvas-left width) (+ canvas-top (truncate (/ height 2))))
   (nop 'SetString (format nil "~A" trial-number))
  (nop 'display)
   (ave-x 'SetString (format nil "~A" (cond ((= 0 trial-number) 0)
                                             (t (/ total-x trial-number)))))
   (ave-x 'display)
   (ave-y 'SetString (format nil "~A" (cond ((= 0 trial-number) 0)
                                            (t (/ total-y trial-number)))))
   (ave-y 'display))
```

The following code is executed whenever there is no other activity going on. It is used to carry on a simulation by generating random points and plotting them on the main drawing area. The various fields are also updated to reflect some statistics regarding the accumulating points. These points are generated according to the probability distribution (and parameters) specified by the user.

```
(defmethod (ChooseDistribution DoIdle) ()
   (let (pt)
        (normal-cursor)
       (cond (distribution-on
             (setq pt (selectq dist-type
                             (u (uniform-point width height))
                          (g (gaussian-point
                              variance width height))
                               (otherwise (uniform-point width height))))
             (self 'clip)
             (setq x (car pt))
             (setq y (cadr pt))
                 (setq total-x (- (+ x total-x) (/ (- width 1) 2)))
                 (setq total-y (- (+ y total-y) (/ (- height 1) 2)))
              (!SetRect rectangle (+ canvas-left x) (+ canvas-top y)
                    (+ canvas-left x 1) (+ canvas-top y 1))
             (!FillOval rectangle black)
             (self 'unclip)
                 (setq trial-number (1+ trial-number))
               (cond ((eq 0 (rem trial-number 10))
                     (nop 'SetString (format nil "~A" trial-number))
                 (nop 'display)
                   (ave-x 'SetString (format nil "~A"
                                     (/ total-x trial-number)))
                  (ave-x 'display)
                   (ave-y 'SetString (format nil "~A"
                                     (/ total-y trial-number)))
                     (ave-y 'display)))))))
```

The following code graphs the probability density function of the distribution specified by the user.

```
(defmethod (ChooseDistribution graph) (&rest 1)
  (let (xc yc xd yd)
        (self 'redraw-dist)
         (wait-cursor)
       (self 'clip)
      (selectq dist-type
               (u (!MoveTo canvas-left (+ canvas-top 10))
                   (!LineTo (+ canvas-left width) (+ canvas-top 10)))
             (g (dotimes (x width)
                        (setq xc (/ (- x (/ width 2)) (/ width 2)))
                      (setq yc (gauss xc (/ variance 100)))
                         (setq xd (round (+ (/ width 2) (* xc (/ width 2)))))
                      (setq yd (round (- (/ height 2)
                                (* yc (/ height 2)))))
                     (!SetRect rectangle (+ canvas-left xd)
                              (+ canvas-top yd)
                                 (+ canvas-left xd 1) (+ canvas-top yd 1))
                        (!FillOval rectangle black)
                       )))
       (self 'unclip)
         (normal-cursor)))
```

The following code returns a random 2-dimensional point selected out of a uniform probability distribution with the given range.

```
(defun uniform-point (width height)
   (list (my-random width) (my-random height)))
```

The following code returns a random 2-dimensional point selected out of a Gaussian probability distribution with the given variance.

```
(defun Gaussian-point (variance width height)
   (prog (pt)
        (setq pt (gd variance))
           (return (list (round (+ (/ width 2) (car pt)))
                        (round (+ (/ height 2) (cadr pt)))))))
```

The following code returns a random integer selected uniformly out of the given range.

```
(defun ud (interval)
   (let (angle r)
        (setq angle (* (rnd) 2 (pi)))
         (setq r (- (random (1+ interval)) 1))
        (list (* r (cos angle))
              (* r (sin angle)))))
```

The following code returns a random integer selected from a Gaussian distribution with the given variance.

```
(defun gd (variance)
   (let (angle r)
        (setq angle (* (rnd) 2 (pi)))
         (setq r (abs (* variance (log (rnd)))))
        (list (* r (cos angle))
              (* r (sin angle)))))

(defun gauss (n var)
   (/ (exp (/ (square (/ n var)) -2)) (* var (sqrt (* 2 (pi))))))
```

The following function squares its input value.

```
(defun square (x) (* x x))
```

The following function returns the value of pi.

```
(defun pi () 3.141592654)
```

The following code updates the appropriate fields and variables when the user types in a new value for the variance.

```
(defmethod (ChooseDistribution type-variance) (itm)
  (ISetCtlValue (variance-bar 'itemControl)
          (case dist-type
                (g (setq variance (val variance-box)))
                 (u (setq interval (val variance-box)))))
  (variance-bar 'Display))
```

The following code updates the appropriate fields and variables when the user scrolls the bar that represents the value for the variance.

```
(defmethod (ChooseDistribution scroll-variance) (bar)
  (case dist-type
       (g (setq variance (IGetCtlValue (bar 'itemControl))))
        (u (setq interval (IGetCtlValue (bar 'itemControl)))))
  (variance-box 'SetNum (case dist-type
                         (g variance)
                           (u interval)))
   (variance-box 'Display))

(defmethod (ChooseDistribution ReturnVar) (itm)
   (ISetCtlValue (variance-bar 'itemControl)
             (setq variance (val variance-box)))
   (variance-bar 'Display)
   (self 'TextActivate 'variance-box)
   (variance-box 'SelectText 0 32000))
```

The following code turns on the plotting of random points.

```
(defmethod (ChooseDistribution plot) (&rest 1)
   (setq Distribution-on t))
```

The following code turns off the plotting of random points.

```
(defmethod (ChooseDistribution suspend-distribution) (&rest 1)
   (setq Distribution-on nil))
```

The following code clears the main drawing area and resets the appropriate variables.

```
(defmethod (ChooseDistribution clear) (&rest 1)
   (setq Distribution-on nil)
   (setq trial-number 0)
   (setq total-x 0)
   (setq total-y 0)
    (self 'redraw-dist))
```

The following code is executed when the user wants some help on the distribution editor.

```
(defmethod (ChooseDistribution DoAboutDistributionEditor) (&rest 1)
   (edFather 'DoAboutDistributionEditor1))
```

The following code generates a "canned" signal constellation consisting of N points uniformly distributed on the unit circle.

```
(defun cp (N  &optional (sa 0.0))
   (let ((ans nil))
      (cond ((> N 0)
             (wait-cursor)
```

```
          (do ((x sa (+ x (/ 360.0 N))))
             ((> x (+ 359.9999 sa)))
              (setq ans (cons (list (cos (rad x)) (sin (rad x))) ans)))
          (normal-cursor)))
     ans))
```

The following code scales the given points by the given factor.

```
(defun shrink (pts factor)
  (let ((ans nil))
     (dolist (p pts)
          (setq ans (cons (list (* factor (car p))
                                (* factor (cadr p)))
                     ans)))
     ans))
```

The following code generates a "canned" signal constellation consisting of N points uniformly distributed on M circles; all their centers are the origin and their radii are an arithmetic progression.

```
(defun cnm (N M &optional (sa 0.0))
  (let (pc (ans nil))
       (wait-cursor)
        (cond ((not (eq N (* M (/ N M)))) (setq N (* M (truncate (1+ (/ N M)))))))
       (setq pc (truncate (/ N M)))
       (do ((k 1 (1+ k)))
         ((> k M))
           (setq ans (append (shrink (cp pc (* k (/ 180.0 pc))) (/ k M)) ans)))
       (normal-cursor)
     ans))
```

The following code generates a "canned" signal constellation consisting of NM points uniformly distributed in an M by M rectangle centered about the origin.

```
(defun rnm (N M)
  (let ((ans nil))
     (cond ((> N 0)
            (wait-cursor)
            (do ((x 0 (1+ x)))
             ((>= x M))
              (do ((y 0 (1+ y)))
               ((>= y N))
                  (setq ans (cons (list (- (/ (1+ (* 2 x)) M) 1)
                                        (- (/ (1+ (* 2 y)) N) 1))
                              ans))))
            (normal-cursor)))
     ans))
```

The following function converts its argument from degrees to radians.

```
(defun rad (x) (* (/ x 180.0) (pi)))
```

The following function returns the value associated with the given field.

```
(defun val (x) (x 'GetNum))
```

The following function assigns the value associated with the given field to be the given value.

```
(defun assign (x y) (x 'SetNum y))
```

The following function returns a random real number between 0 and 1.

```
(defun rnd () (/ (1+ (abs (!Random))) 32767))
```

The following function returns a random integer between 0 and n.

```
(defun my-random (n)
  (truncate (* (rnd) n)))
```

The following function draws an ellipse centered about the point (x,y) with major and minor axis rx and ry; optionally the ellipse may be shaded and the thickness of the line used to draw it may also be controlled.

```
(defun ellipse (x y rx ry &optional (fill nil) (tk 1))
  (let ((rectangle (Rect 'new)))
       (!SetRect rectangle (- x rx) (- y ry) (+ x rx) (+ y ry))
      (!InvertOval rectangle)
      (cond ((not fill)
             (!SetRect rectangle (+ (- x rx) tk) (+ (- y ry) tk)
                    (- (+ x rx) tk) (- (+ y ry) tk))
             (!InvertOval rectangle)))))
```

The following line must be the last line in this file when this program is converted into a double-clickable stand-alone application.

```
; (launchappl 'sc-tool)
```