DESIRE REFERENCE MANUAL

by Granino A. Korn

G.A. AND T.M. KORN INDUSTRIAL CONSULTANTS
1311 Maple Street, No. 3, Wenatchee , WA 98801 (509) 667-2286
sites.google.com/site/gatmkorn

Chapter 1 (Linux, Unix, Cygwin)

Chapter 1 (Windows©)

Chapter 5. Vector/Matrix Operations and Fast Fourier Transforms

ARRAY DECLARATIONS

COMPILED VECTOR OPERATIONS IN DYNAMIC PROGRAM SEGMENTS

DOT PRODUCTS, VECTOR NORMS, AND SUMS IN DYNAMIC SEGMENTS

VECTOR/MATRIX DIFFERENTIAL EQUATIONS AND DIFFERENCE EQUATIONS

TIME-HISTORY STORAGE IN ARRAYS

INTERPRETED VECTOR AND MATRIX OPERATIONS, FAST FOURIER TRANSFORMS, AND CONVOLUTIONS

Chapter 6. Time-history Output, Storage, and Recovery. SHOW Output

TIME-HISTORY OUTPUT IN DYNAMIC PROGRAM SEGMENTS

FAST BINARY-FILE TIME-HISTORY STORAGE AND RECOVERY

Chapter 1. Desire Operation under Windows©

TO GET STARTED

(a) Install Desire. Desire is supplied as a zipped installation archive **mydesire.zip**, or as a folder **\mydesire** in the **\WINDOWS** folder of a distribution CD. ***To install and uninstall Desire you simply copy or erase the installation folder.*** You can even install Desire on a flash stick. Desire ***never*** uses the troublesome Windows registry!

Proceed as follows:

- ***Create an installation folder***, say **d:\mydesire**, and **cd** to it.
- ***Copy and unzip the installation files – and you immediately have a complete, ready-to-go modeling/simulation system***

(b) Start Desire. To start Desire, double-click on **Wdesire.bat** in your installation folder. Desire starts and displays the date and time, plus some help information. ***We suggest that you place a shortcut to*** **Wdesire.bat** and also one to **Wdesire.exe** ***on your Windows desktop.***[1]

You will see (Fig. 1)

- A scrollable ***Explorer (file manager) Window*** listing subfolders like **SIM**, **Monte Carlo**, **Control**, … that contain user-program files with the extension **.src** or **.lst**. Open one of these folders, or a user-program folder of your own.[2]
- a ***Desire Editor Window***, with menus including an editor help menu.
- a ***Desire Command Window*** (a "console window"), ready to accept typed commands and program lines terminated with **Return**.

Position and size Desire's windows as you like. You can also conveniently change the Command-window color, fonts, and initial position by right-clicking the title bar and using the **Properties** screen. Desire then remembers your command-window and editor-window arrangement for future sessions.

Windows can run multiple copies of Desire (and thus multiple simulations with multiple Command Windows and the same or different Editor Windows) concurrently.

Now ***activate the Command Window by clicking on it and type*** **erun** (or more conveniently **zz**), followed by **Return**). ***Your simulation (or a multi-run simulation study) will compile and run immediately.*** The ***Desire Graph Window*** opens to show solution graphs as needed. You can move the Graph Window to any convenient position

---

[1] If the Command Window should crash due to a Windows glitch or insufficient memory, ***you can then reconstitute it immediately*** by double-clicking the **Wdesire.exe** icon.

[2] Path names of Desire user-program files (like **c:\samples\cannon.src**) are currently limited to 125 characters, plenty for most purposes.

(c) Run an Example Program. Drag-and-drop a Desire user program, say* .\sim\lorenz.src*, from its Explorer Window into the Editor Window.[3]** Edit the program if you like. Then click the red **OK** button; this transfers your edited program to Desire.

on your desktop with the mouse, or set its position with the command-mode or programmed statement

display W ***x,y***

In this way you can quickly try, modify, and run stored user programs. To ***quit*** Desire type **bye** in the Command Window if you want to save your last program, and click the command-window exit button.

Typing **ctrl-c** closes the Command Window but leaves the Editor Window intact You can always open a new Command Window by double-clicking **\mydesire\ Wdesire.exe**, or a shortcut icon for it.

> Desire is a large program with many advanced features. Good application software always hides its advanced features until you need them, but look at the ***Desire Reference Manual***, which is included with your distribution. There are also several textbooks that describe applications to control systems, biological systems, Monte Carlo simulation, and neural networks.

---

[3] If **.src** files have been associated with **deditor1.exe** as in earlier versions of Desire; double-clicking on a **.src** file still shows the file in an Editor Window. But now this Editor Window's **OK** button will ***not*** produce the desired result.

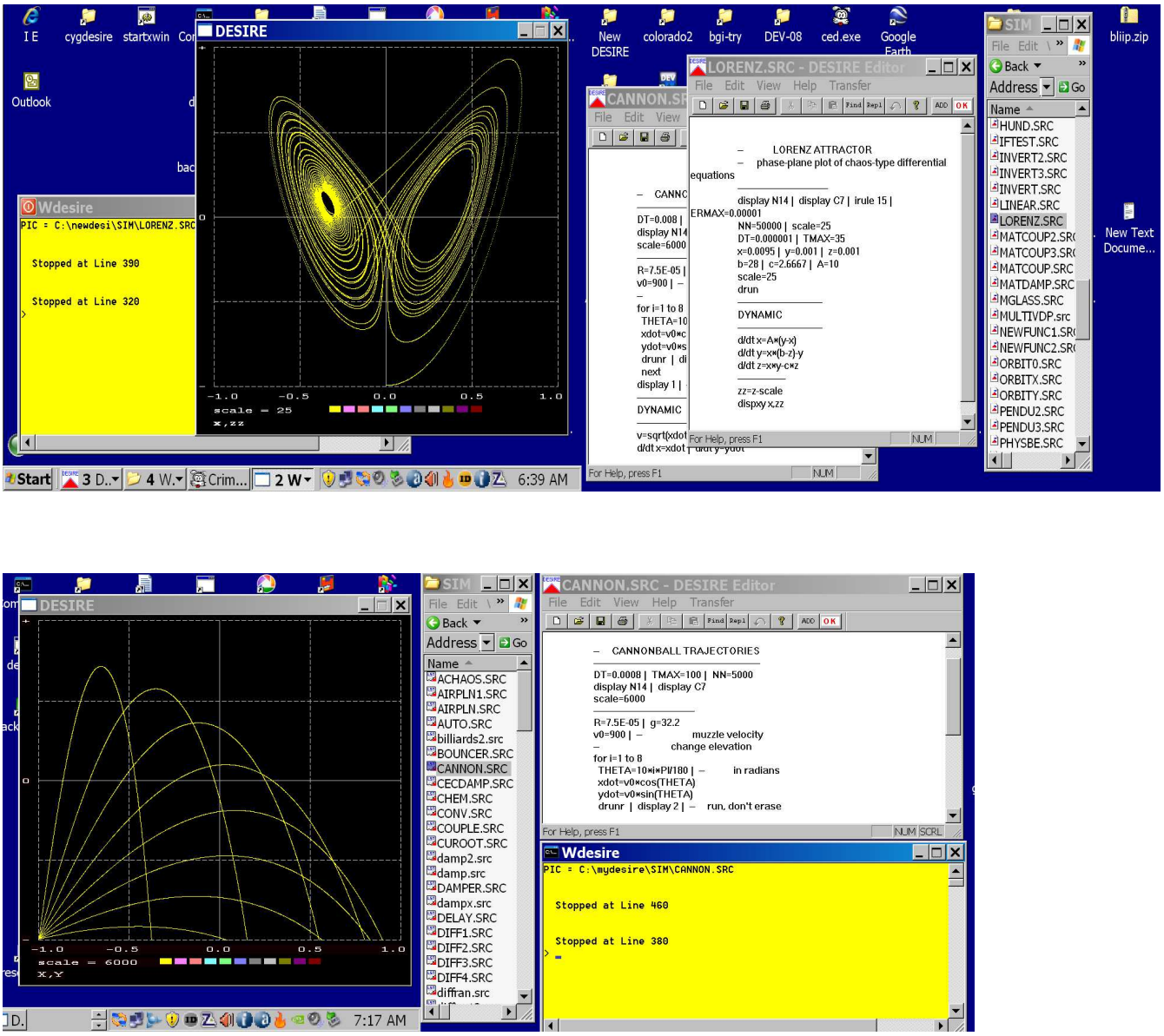


Fig. 1. Windows dual-screen displays showing the Desire Command Window (yellow), the Graph Window, an Editor Window or Windows, and a filemanager window. Windows can be rearranged as you like.

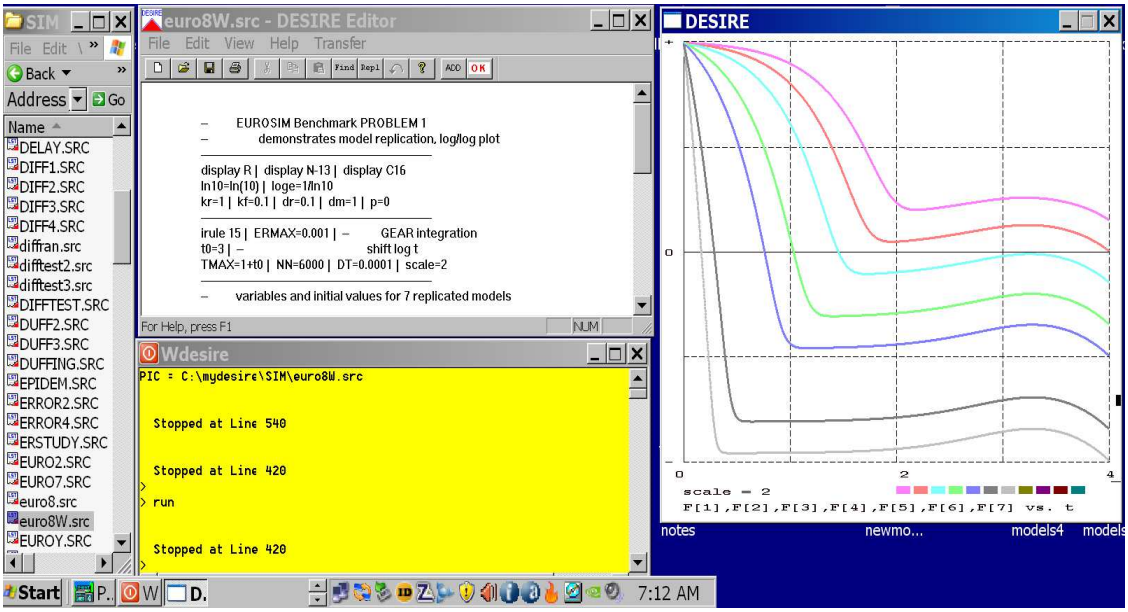


Fig. 2. This display has a Graph Window with a white background, nice for publications.

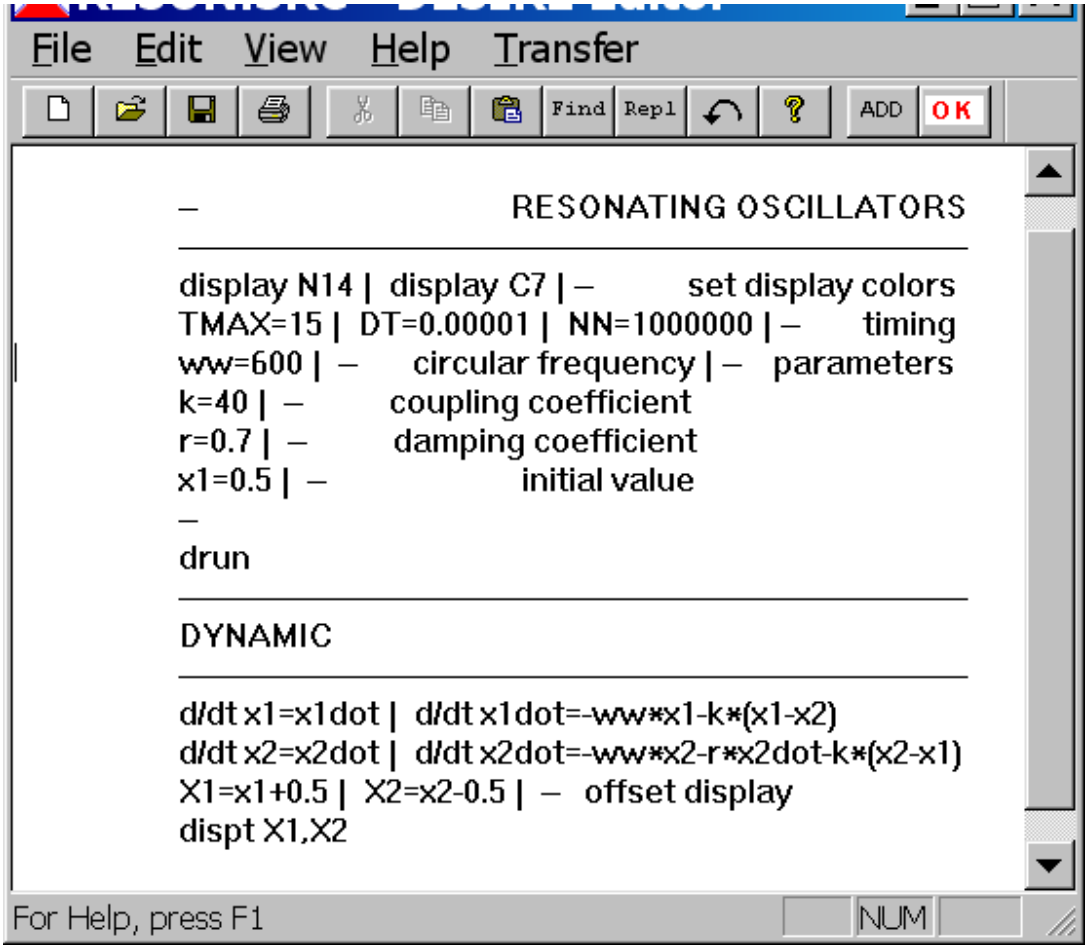


Fig. 3. Desire Editor Window for a small simulation program. The interpreted experiment-protocol script at the top exercises the runtime-compiled DYNAMIC program segment.

A SYSTEM FOR INTERACTIVE MODELING

1-1. Introduction

(a) To recap, the Windows version of Desire displays a ***Desire Command Window*** and a ***Desire Editor Window***, plus optional Explorer (file manager) windows for user programs (Fig. 1 to 3)***. To open additional Editor Windows*** type **ed** in the Command Window.

The Desire installation directory contains:

- ***executable files*** such as **Wdesire.exe**, **deditor1.exe**, **Wdesire.bat**.
- ***subfolders containing user programs*** (e.g. **\SIM**, **\Control**). Desire user program files normally have the extension **.src**. They are ordinary text files - you can, if you wish, read and print them with any editor or word processor.
- An (optional) subdirectory **\help** containing ***help-screen files***. These are ordinary text with the extension **.txt**.files; they can be dropped into Desire Editor Windows for display. ***You can easily make your own help screens*** by writing more **.txt** files.

You can ***enter a new user program*** in an Editor Window, or drag an existing user program into an Editor Window and edit if you wish. Click the **OK** button to transfer the edited program to Desire. Typing **erun** or **zz** in the activated Command Window then causes immediate very fast compilation, executes the simulation program, and displays solution graphs in color. ***Such "direct execution" permits truly interactive modeling.***

(b) To enter a new program use **File | new** in an Editor Window and then name and save the program as a named **.src** file with **Save as**.[4]

(c) With different programs, or different versions of the same program, in their own Editor Windows you can select any one of them for execution by clicking its **OK** button. This makes it easy to compare programs and models.

(d) User-program Files. Desire user programs are ordinary text files with the extension **.src** or **.lst**. **.lst** files are rarely needed (see below). File names are currently limited to 25 characters.

To inspect **.src** or **.lst** files drag them into a Desire Editor Window. If you associate **.src** and **.lst** files with a standard text editor like **Notepad** you can also display them by double-clicking on their icons. But do ***not*** associate **.src** and **.lst** files with **deditor1.exe** (as in some older versions of Desire); the resulting editing windows would ***not*** transfer programs to Desire.

(d) The Desire Command Window displays ***error messages*** and selected runtime output. ***Runtime error messages*** appear at the top of the Desire Command Window. Maximize

---

[4] You can, instead, type **ed** in the Command Window and supply a location and name in the **PIC** line. Enter your program between the slashes. Subsequent **erun** (or **load**) and **keep** commands will save the program in the specified location.

the window as needed to read messages. A runtime error message normally reproduces ***the numbered program line*** (Sec. 1-7) ***containing the error***. You can then find and correct the guilty line in your Editor Window.

(e) Desire handles up to 40,000 first-order ordinary differential equations with a variety of fixed- and variable-step Runge-Kutta integration rules, or up to 1000 differential equations with variable-step/variable-order Gear and Adams rules, ***plus*** difference equations and neural networks. You can combine simulations of dynamic systems, neural networks, and fuzzy-logic controllers. Desire programs can use up to 500,000 real variables or parameters. All scalar and matrix computations use 64-bit double-precision format.

(f) Desire does not use canned dialog boxes for data entry but lets you program ***your own interactive-dialog windows*** for individual user programs (Sec. 1-10). These windows can contain explanatory text and also fields for data entry and function-table modification.

1-2. Editor-window File Manipulation

(a) File and Edit Commands. Desire Editor Windows have ***standard Windows* File-*menu commands***

File New	**File Open**	**File Save**	**Save As**
Print	**Print preview**	**Print setup**	**Exit**

The Editor Window has ***toolbar buttons*** for the **File**-menu commands **New**, **Open**, **Save**, and **Print** and also for the standard **Edit**-menu commands

Cut	**Copy**	**Paste**	**Undo**	**Find**	**Replace**

Cut, **Copy**, and **Paste** can also be normally accessed with the right-hand mouse button. You can now ***load***, ***enter***, ***print***, ***edit***, and ***save*** Desire program files and, in fact, any text file, much as in the Windows **Notepad** editor. The **File Open** command always starts with the last-used directory. The **File** menu also lets you click directly to load one of the 4 most recently used program files.

The fastest and most convenient way to load stored user programs (**.src** or **.lst** files) is from a scrollable **Explorer** (filemanager) window. Simply ***drag the file into an Editor Window***. As already noted, you can open several Editor Windows and select different programs to run.

(b)* Transfer *Menu and Buttons. When you are satisfied with your edited text, ***transfer your program lines to Desire*** with a **Transfer OK** or **Transfer ADD** command:

- a mouse click on the **OK** Button transfers the editor-window text to Desire ***as a new program***, or
- the **ADD** Button ***adds*** your edited text to any existing Desire program, as discussed in Sec. 1-10. This is nice for combining programs or subprograms, or for adding library files.

1-3. How to Quit and Uninstall Desire

To exit Desire, type **bye** in the Command Window. The program then ***saves your current user program*** in the ASCII source file **SYSPIC.lst** and disconnects (closes) all open files and devices. To quit, click the Command Window's **exit** button. Close all unwanted Desire Editor Windows and **Explorer** windows by clicking their **exit** buttons.

To ***uninstall*** Desire simply delete the installation folder. Unlike most Windows programs, Desire leaves no garbage on your disk.

1-4. Getting Help

Desire ***Editor Windows*** have standard Windows **F1**, **shift-F1**, and **Help**-menu help (refer to your Windows Manual). In addition, clicking the toolbar **Help** Button (**?**) produces context-based help for each item on the Editor-window toolbar.[5]

Help files for Desire itself are ordinary text files, so that users can easiy write their own help files or special instructions. Helpt files for different purposes can be collected in special folders. The easiest way to display such files is to drag-and-drop them into a Desire Editor Window.

A text file **filespec** located in the **\mydesesire** folder also display in the Command Window when you type **help filespec** in the Command Window..

COMMAND LINES

1-5. Desire Command Lines and Operating-system Commands

(a) Commands and Experiment-protocol Scripts. Desire commands permit both ***interactive*** and ***programmed*** control of simulation experiments. Commands entered as part of a program form its ***experiment-protocol script***. The Desire command language described in Chap. 2 is a complete computer language in its own right.

Commands like **run** or **keep** typed into the Command Window execute at once. You can, if you wish, display their effects immediately with **write** commands. Such command interpretation is precisely what you need for interactive computer-aided experimentation and for program debugging (Chapter 7). Simulation-model code in DYNAMIC program segments, on the other hand, is ***compiled*** for fast execution (Chaps. 3 and 4).

Desire commands and program lines may contain multiple statements separated by the ***statement delimiter*** **|** , e.g.

```
x = 15  |  write x
```

Anything following a ***comment delimiter*** **--** and a subsequent space is a ***comment***.

(b) Operating-system Commands. Programmed or command-mode statements prefixed with **sh** (followed by a space) are ***DOS-type operating-system commands***:

```
sh dir
```

[5] Deditor1 can access the Windows XP help-display program winhlp32.exe (all pertinent information is contained in this manual chapter, though). Vista normally uses a different help-display program; to download winhlp32.exe, google winhlp32.exe and follow instructions on the Microsoft web page.

```
sh copy foo.dat moo.dat
sh xprogram
```

You can exit the DOS-type shell with **exit**.

Desire can thus access the full power of DOS-type commands. You can copy, rename, and edit files, create directories, or format media without leaving Desire. You can run batch files, and even Fortran or c programs like special graphics or optimization programs. Desire communicates with other programs by reading and writing files (Chapter 2). DOS-type commands prefaced by **sh** can also be used in experiment-protocol scripts.

(c) Deleting and Renaming Files. In particular, DOS-type commands can ***delete*** and ***rename*** files:

```
sh del kotz.src
sh del /mydir/bippy.*
sh ren /castle/prince.src  /castle/frog.src
```

1-6. **run**, **STOP**, and **go**

(a) The command **run** ***executes ("runs") the program currently in memory***. A mouse click in the Graph Window ***pauses execution*** of a program involving a single simulation run***;*** you can ***resume execution*** by typing **Enter** in the Command Window (see also Sec. 3-1b). You can edit the program and then continue the run with **drun**. Typing any character (e.g. **space**) followed by **Enter** ***terminates*** a paused run.

(b) Experiment-control-script execution stops when the program reaches a programmed **STOP** statement (Sec. 7-3). Typed commands can then modify the program and/or continue it with **go**.

1-7. Miscellaneous Commands

(a) The* time *Command. The command

time

displays ***the current date and time***. Used as a program line, **time** produces a ***date stamp***.

(b) Program Listings. Ordinarily Editor Windows are much more convenient than long program listings, but the commands

list and **list+**

can be useful for debugging (Chap. 7). They respectively list the current program without and with Desire line numbers (Sec. 1-8).

You can ***list specified lines, and/or ranges of lines*** with commands like

list 213, 245, 300- 410 **list+ 322, 410**

list '*filespec*' and list+ '*filespec*' list the program in a specified file.

(c) Clearing the Computer Memory. The command

new or **new *'filespec'***

closes any open editor window, ***erases the current program*** (if any), and sets the problem

identification code respectively to **no_name.prc** or to ***filespec***. Desire asks you for confirmation before erasing. The command **NEW** clears the computer memory ***without*** aasking for confirmation. **clear** erases all variable, array, function, etc. definitions.

DESIRE LINE NUMBERS

1-8. Program-line Numbering. The keep *and* keep+ *Commands*

(a) Desire Line numbering and Program Modifications. Desire program lines may be entered with or without ***line numbers***. Line numbers are rarely needed. They can be useful for identifying lines containing errors, or lines in special dialog windows (Sec. 1-10).

> Line numbers permit temporary program modifications. Desire processes program lines in numerical order, so that one can add numbered program lines in any order anywhere in a numbered program either in an EditorWindow. A new numbered program line replaces a preceding line with the same number. If the new line has only a carriage return, then the old line is erased. Saving your program on disk with **keep** or **keep+** preserves all editing changes.

NOTE: You can insert definitions of functions, procedures and/or submodels ahead of your program. You can do this by typing them in, or you can make a library file using numbered lines (Sec. 1-10) and use the **ADD** Button (Sec. 1-9).

(b) Programs Without Line Numbers. It is normally most convenient to enter, save, and print Desire programs entirely without line numbers.

When you enter a program without line numbers Desire automatically supplies ***internal line numbers*** that start with **210** and increment by 10. To see these line numbers, type **list+** (Sec.1-7).

(c) The typed command **keep** saves the program currently in memory in a text file ***filename*.src**, without line numbers, in the folder last used by the Editor Window. The command **keep+** saves the program with line numbers in ***filename*.lst**.

1-9. Transfer-ADD *Operations in Editor Windows*

When you transfer edited text from an Editor Window by clicking on the **Transfer-ADD** Menu, or more conveniently on the **ADD** Button, ***numbered lines*** will be automatically inserted in the correct numerical order. ***Lines without line numbers*** are appended at the end of any existing text, with line numbers normally incrementing by 10. You can thus add an extra program segment at the end of an existing program.

1-10. Programming Special Editor Windows and Dialog Boxes

(a) Simulation experiments typically require repeated interactive changes of just a few program lines, say Lines **110** to **134**, **200**, and **206**. Usually such lines assign data values, but they can also define functions, procedures, differential equations, or input and output files for a series of experiments. Although it is easy enough to modify a program in the main Editor Window, it is nice to avoid repeated scrolling through large programs. This is a useful application for numbered program lines.

A programmed or command-mode **edit** line like

1200 edit 110-134, 136-139, 200, 205

produces an Editor Window that displays only the listed program lines for editing. When you click **OK**, Desire inserts the resulting numbered program lines into their correct places in the program currently in memory.

When the edited program runs to your satisfaction, you can make the change permanent by typing **keep** or **keep+.**

(b) You can place your special **edit** lines into a program segment starting with **label eee** (Sec. 7-2) and ending with **STOP** (Sec. 7-3), as in

```
label eee
   edit  110-134, 136-139, 200, 206
   STOP
```

You can then call this labeled routine by typing **go to eee**.

Such program segments can include appropriate comment lines, including help for the user and even character graphics. This lets you ***program special dialog boxes***.

ADDITIONAL COMMANDS

1-11. More File Manipulation

In the following,

filespec = {drive:}{\directory\...} filename{.extension}

specifies the ***program identification code (PIC)*** of a program, i.e. the path to its storage location. The most important file-manipulation commands are

reload (or **rld**) '***filespec***'	loads a source program (erases program in memory)
load '***filespec***'	loads or adds a source program (does not erase)
keep '***filespec***'	saves the program, without line numbers (file extension defaults to **.src)**
keep+ '***filespec***'	saves the program, with line numbers (file extension defaults to **.lst)**
new	erases program in memory (asks for confirmation)
new '***filespec***'	erases program in memory, specifies PIC
NEW	erases program in memory (no confirmation query)
PIC	displays current PIC
PIC '***filespec***'	sets new PIC

When no file extension is specified, **keep** and **reload** default to the file extension **.src**, and **keep+** and **load** default to the file extension **.lst.**

1-12. Listing Programs in the Command Window

The commands

list and **list+**

respectively ***list the current program*** without and with line numbers, in the Command Window. Commands like

list 2100, 2300 - 2340

similarly list selected program lines (see also Sec. 1-8). This can be useful for debugging (Sec. 7-3)..

1-13. Running a Simulation in the Background

It can be useful to run a large statistical or optimization study, or train a neural network, ***in the background*** while you use Windows to do something else (even if it is only a solitaire game). You can do this if you suppress all runtime graphics with a **display 0** statement (Sec. 6-4), or better by omitting **dispt**, **dispxy**, and **DISPXY** display requests in your DYNAMIC segments.

You can still watch the progress of your simulation in a suitably sized Command Window by using **write** statements in the experiment protocol, or **type** statements in a DYNAMIC program segment. The simulation program will continue even while you work in other windows.

Once your Command Window shows that the simulation program is done, you can edit the program to include graphics showing, say, the time history of the last simulation run, or a crossplot and start the new program. You can also run another program reading results of the earlier simulation from **.dat** or **.tim** files.

1-14. Multiple Desire Simulations in Multiple Command Windows

It is possible to ***run two or more Desire simulations concurrently***, each in a separate Command Window.

Do ***NOT*** simply call **\mydesire\Wdesire** more than once. For each extra instance of Desire and its command window, you must copy the entire installation folder **\mydesire** to a new folder, say **\mydesi**, and then replace **\mydesire** in the batch file **Wdesire.bat** with **\mydesi**. Then **\mydesi\Wdesire** produces a new Command Window and new Editor Windows; note that but each set of Editor Windows works ***only*** with its own Command Window. You would typically reposition at least one of the Graph Windows with **display W *x,y*** (Sec. $$) in the corresponding Command Window.

APPENDIX A: Rarely Used Features

The following program features are rarely used. Since they cost little memory and no speed they were kept for compatibility with the earlier versions of Desire used in pre-2007 textbooks.

Command-window Program Entry and Editing Using Desire Line Numbers.

We normally edit programs in editor windows, but the experiment-protocol interpreter alone can also do simple program entry and editing in the Desire Command Window. This may be useful for small interactive program changes when Desire has returned a numbered program line with an error message.

(a) With the memory cleared, you can ***enter or add new program lines with line numbers***, say

```
50      X = 15
55      yak2 = 7  |  variab16 = - 15.55 + X
60      write (X + yak2) * cos(variab16)
```

Unlike un-numbered command lines, such numbered program lines execute only when you type **run**.

Program or command lines may contain ***multiple statements*** separated by |. A line can have up to 244 characters, which "wrap around" on the display screen. DESIRE program listings then show _ as a ***continuation-line symbol***.

NOTE: Remember that you can change drives and directories and copy, rename, and delete files without leaving DESIRE by using shell commands prefixed by **sh**.

(b) To see what you have actually entered, type **list+** to list all numbered lines. You can, instead, ***list selected lines and/or line-number ranges*** with commands like

list 434-439, 210, 300- 400

(c) While you enter a line, you can edit it by using the **backspace** key and entering or deleting text before you type **Return**.

(d) Desire lets you

1. ***overwrite a program line*** by typing a new line with the same line number.
2. ***delete a program line*** by typing its line number followed by **Return**.

To ***delete more than one program line***, use commands of the form

erase 332- 339, 378, 400- 500

(this would delete Lines 332 through 339, Line 378, and Lines 400 through 500). **keep** saves the edited program.

Auto-line-number Mode and Renumbering

(a) Auto-line-number Mode. When you type **/** (followed by **Return**), ***numbered prompts*** invite you to enter program lines with ***automatically supplied line numbers***, starting with

210>

Successive line numbers then increment by 10. Lines with errors are not entered. You can restart with any desired line number by simply including it on your line, e.g.

450> **85** **y = 7 | z = 0-19.1**

or by typing the line number followed by **Return**. To produce line-number increments other than 10, say 50, start or restart auto-line-number mode with

auto 50 or **/auto 50**

Reusing any line number overwrites the previously-entered line with the same number, so that you can erase or edit lines.

To leave auto-line-number mode, type a slash **/** followed by **Return**. To ***execute commands while you are in auto-line-number mode***, prefix any typed command with a slash **/**, e.g.

/list or **/run**

This will also cause Desire to quit auto-line-number mode.

(b) Renumbering Program Lines. To ***renumber the lines*** of a program currently in memory as **210, 220,** … , use

keep 'dummy'	(creates **dummy.src**, without line numbers)
rld 'dummy.src'	(line numbers are now **210**, **220, . . .**)
keep or **keep+**	

You can, instead, start the new line numbers with, say, **2000** by using

keep 'dummy'	(creates **dummy.src,** without line numbers)
NEW	(erases the program)
1990	(empty first line)
load 'dummy'	(append the program; **load** targets defaults to **.src**)
keep or **keep+**	(saves renumbered program)

Line numbers still increment by 10; the typed command **auto** ***increment*** changes the increment if desired.

Loading and Saving Programs in Condensed Form

The command

save '*filespec*'

saves a program in condensed("precompiled") form (for later retrieval by **old** or **chain**). If you do not specify a file extension, the extension defaults to **.prc**. The path defaults to the current default path. Given without a file specfication, **save** saves the program currently in memory in the current default directory under its current program name, with the default extension **.prc**.

The command

old 'pa'

loads a user program **pa.prc** stored in condensed form with the default **.prc** extension into memory.

Program Chaining

Program chaining loads and runs two or more successive **.prc** (not **.src** or **.lst**) rograms or program segments from disk; common data can be preserved. Nowadays we prefer to use programs with multiple DYNAMIC segments.

The command

chain '*filespec*',*n*

loads the user program ***filespec*** saved earlier ***in precompiled form*** (**.prc** format, see above) The program executes immediately, starting at Line Number ***n***. If ,***n*** is omitted then execution starts with the first line of the new program. The old program is overwritten, but ***all variable and array names and data are preserved for use in the new program***. If you do *not* want the old data, use a **clear** statement before **chain**, or in the new program. The problem identification code (PIC, Sec.$$) changes to that of the new program. The default file extension for **chain** is **.prc**.

User-defined functions and procedures will not survive chaining. In fact, ***DESIRE keeps you from re-using their names*** (and those of their parameters) until you erase all declarations with **clear** (Sec. 2-9). Chaining causes recompilation of DYNAMIC program segments.

APPENDIX B: SOURCE-CODE COMPILATION AND LINKING

Desire for Windows is compiled and linked with the MINGW version of the open-source GNU system. The program employs the open-source Borland-type graphics library developed by Professor M. Main at the University of Colorado.[6] MINGW installation, and also the correct compiler and linker strings used with the graphics library, are quite complicated but become child's play if you install the University of Colorado's excellent **cs1300** workbench.

Download **cs1300-setup.msi**[7] to install the workbench. The resulting console window lets you **cd** to your source-file directory and call a large number of built-in open-source programs like **gcc**, **gbd**, **c++**, **g77**, **make**, **emacs**, and many others.[8] In particular, **bgi++** calls **g++** with pre-built include and library switches for Borland-type graphics.[9]

To create Desire for Windows (**Wdesire.exe**) ***simply use***

```
bgi++ -c –O3 *.c          (compile command)
bgi++  –o Wdesire *.o     (link command)
```

The writer is grateful to Professor M. Main for his invaluable help with **wbgim** and **cs1300**.

[6] **http://www.codecutter.net/tools/winbgim/V6_0/doc/bgi.html**
[7] **http://www.cs.colorado.edu/~main/cs1300/#installation**
[8] This is almost as good as a Linux or Unix system!

[9] Since the graphics library is written in c++, Desire compilation uses **g++** rather than **gcc,** even though Desire itself is written in standard c.

Table 1-1. PARTIAL LIST OF Desire FILES

1. HELP SCREENS

There are duplicate copies in the default installation directory and in the **.\help** subdirectory, if any. ***You can easily add your own help-screen files***; any ASCII text file works!

howe	**if**	**program**	**next**	**filename**
files	**fkeys**	**proc**	**why**	**types**
edit	**menu**	**loops**	**nonlin**	**irule**
fun	**stiff**	**display**	**keywords**	**programs**
vector	**show**	**examples**		

(neural-network help screens keyed to **.lst** examples with similar names):

parity3	**creepmin**	**xor**	**art**	**parity**
bpmap	**cpmap**	**pay**	**vquant**	**nquant**
entropy	**babybam**	**code**	**sweeptst**	**hquant**

(Table 1-1 continued)

2. USER EXAMPLE PROGRAMS.

Examples are in **.src** format (without line numbers) or **.lst** format (with line numbers). In the following partial list of examples, **?** and ***** are standard Unix or Windows wildcard metacharacters which stand, respectively, for any legal character and any legal string.

sincos	graph-plotting demonstration
scalef	automatic graph scaling
fft?	FFT demos
conv	convolution via FFT
invert?	matrix inversion (single precision)
freq	complex numbers, amplitude and phase
plot?	conformal mapping
signal	signal generators
vdp, multivdp	Van der Pol's equation
lorenz	chaos-type problems
scroll	
achaos	
mglass	Mackay-Glass time series (chaos)
quant	quantization (a/d converter simulation)
prob	histogram (unsupported feature)
avg	signal averaging demo
minmax	running max/min, function storage
fun? , newfu*	function generators
store1, get1	function storage demo
delay, tdelay	delay demos
backl	backlash

oscilltr, damp, couple, signal oscillators

cannon?	18th-century ballistic-weapon system
bouncer	bouncing ball
orbit?	translunar space-vehicle orbits
to*	torpedo simulations, with and without control, Monte Carlo
airpln	flight simulation with "**stash**"
airpln1	flight simulation without "**stash**"
billiards?	billiard simulation
pilot?	pilot-ejection benchmark, crossplot
pendu?	Pendulum simulations
del1500	delay line, 1499 differential equations
del2000	- same with 1999 differential equations
line?	- same in vector/matrix notation, up to 6000 equations
gluc	glucose-tolerance test
epidem	epidemic propagation
physb?	blood-circulation model
host?, prey?	Volterra/Lotka equations (predator-prey ecology)
world	Forrester-type world model
orbit*	space-orbit simulations
matcoup?	vector differential equations
matdamp	vector differential equations
***x**	stiff-integration examples
stiff?	stiff-integration examples
bangstep*	bang-bang servo, variable-step integration
triga*, step	nuclear-reactor simulation, stiff
gear?, fuel?	stiff-integration examples (nuclear reactor)
satser?	servo simulation
howe	satellite roll controller (demonstrates submodels, delay)
servo*	servo impulse and frequency response (uses fft)
lib	transfer-function submodels
optrun	iterative multi-run optimization of a servo
fuzz*	fuzzy-logic control simulations

(Neural-Network Programs)

xor*	XOR with backpropagation, with and without batching
xormin	XOR, creeping random search
parit*	parity learning, mixed backpropagation tricks
cruupity	parity learning, creeping random search
ch*	Cottrell/Zipser encoding, various types of backpropagation, Chebyshev and nonquadratic criteria
sweeptst	different activation functions and their derivative
bpmap*	backpropagation function-mapping networks
bpxy*	4-layer backpropagation

soft	different softmax classifiers with probability output, associative memory
rdf*, pdf*	different radial-basis-function networks
narendra?	Narendra control-system optimizer
autoas*	linear classifiers and associative memory
statgood	vector quantization
vquant?	vector quantization with and without conscience (FSCL)
pquant*	competitive learning for parttern recognition
multiwin	competitive learning with multiple winners
hquant?	hierarchical vector quantization
entropy	entropy and histogram
cohen*, herault	time-history analysis
qual1	competitive learning of nonlinear regression
predict*	linear and nonlinear predictors
nconv	2-dimensional correlation detector
hry*	principal components and related topics
bam	bidirectional associative memories (BAMs)
cbam	competitive-layer BAM
hobam	BAM with nonlinear feedbacl
olambam	BAM using optimal linear transformation
alpha*, art*	pseudo-ART, alphabet and spiral learning
jetsx	new Jets/Sharks demo
neuron*	pulsed neuron
hop*	Hopfield-type networks
?feed	Grossberg-type signal enhancement
cp*	counterpropagation mapping networks
spiral*	spiral-learning counterpropagation networks
qiral?	
piral?	
artmap?	
fuzz*	fuzzy-logic examples
fizz*, fazz,* fozz*	
autoas*	these character-recognition programs are two-layer and three-layer autoassociators, illustrate thresholding and competitive noise reduction techniques
vq*	competitive template matching used to measure entropy and mu tual information
tessel*	demonstrate Voronoi and Haken-type tessellations

(Partial Differential Equations, Method of Lines)

heat*	heat conduction experiments
hexch*	heat-exchanger simulation
pde, advec*	hyperbolic PDEs

euro* EUROSIM comparison problems

(Monte Carlo Simulations)

rwalk*, timcan*, to22x*, toparz1, tozz?, Vnoise, minmax,

(Parzen-window Measurements)

parz*, porz*, exp3d, tri*, unif*, sin3d

(Nuclear Reactor Simulations)

triga*, fuel*, step

Revised 10/14/11

Chapter 1. Desire under Linux or Unix

Getting Started

1-1. Desire Installation

(a) We normally use a GNOME desktop for Linux; KDE also works. Desire runs in a ***terminal window*** controlled with keyboard commands terminated with the **Return** key. Set up a terminal window and create a directory called **/desire** in your user area (**/home/username/desire**) and go to this directory:

mkdir desire
cd desire

Copy the distribution file **desire.tgz** and unzip it with

tar xvzf desire.tgz

Your* /desire *folder now contains a complete ready-to-run Desire system with many examples (**.src** files) in different subdirectories like **/desire/sim** and **/desire/control** (Fig. 1).[1]

(b) Decide next which program editor you want to use with Desire. GNOME's default editor **gedit** works nicely, but **gedit** cannot produce the very useful multiple editor windows discussed in Secs. 1-5 and 1-6.

Currently, the most suitable editor is **kwrite**. This installs automatically under Fedora 14-up if you activate KDE in the installation script (***even when you normally use GNOME, as we do***). Under other Linux distributions (like Ubuntu), you can fetch and install **kwrite** with the SYNAPTIC package manager. You can, instead, install **nedit**; an **nedit** package file is included in the Desire distribution. Please refer to Sec. 1-6 before you use **kwrite** or **nedit**).

NOTE: Currently, the distribution file **desire.tgz** is configured to use **kwrite**. You can edit the three files **editcfg1**, **ekiller.old**, and **mk_syspict.sh** in the **/desire** folder to replace **kwrite** with **nedit** or another editor.

1-2. Try Desire: Load and Run a User Program

(a) Starting Desire. **cd** to the **/desire** directory. Type

./name

where ***name*** is the name of a Desire executable file, say **ydesire64**. Desire then announces itself with the current date/time, and the prompt **>** invites you to type commands in the terminal window.

Some currently available Desire executables are

ydesire64	fast, uses 3 global hardware register variables (replaces **udesire64**)
xdesire64	an older version
Xdesire32	obsolete - will not work on 32-bit Linux systems
cygdesire.exe	for Cygwin (Unix under Windows)

[1] If you have a Desire CDROM, installation is even easier. Simply copy the **/desire** folder to the hard disk, open a terminal window, and **cd** to **/desire**.

> **NOTE:** We no longer use 32-bit Linux. If you still have a 32-bit Linux system and need a Desire executable, simply go to the folder **xtry32** and type **make** to produce **desire**. To create a 32-bit executable on a 64-bit Linux system you must use compiler and linker command strings containing **-m32** explicitly.

(b) Load a User Program. It is most convenient to load an existing user programs by simply clicking on its file icon in a filemanager window or on the desktop (Sec. 1-5). But to try Desire quickly you can load an example source program, say **./sim/lorenz.src**, with a keyboard command

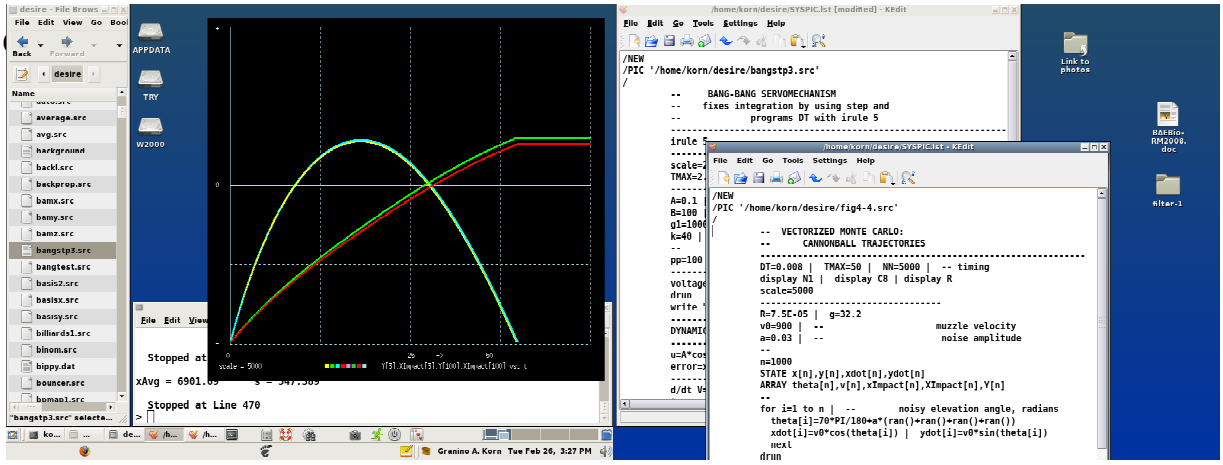


Figure 1. Desire on a Linux dual-screen display. From left to right you can see a filemanager window, the Desire terminal window, the Graph Window, and two Desire Editor Windows for two different user programs.

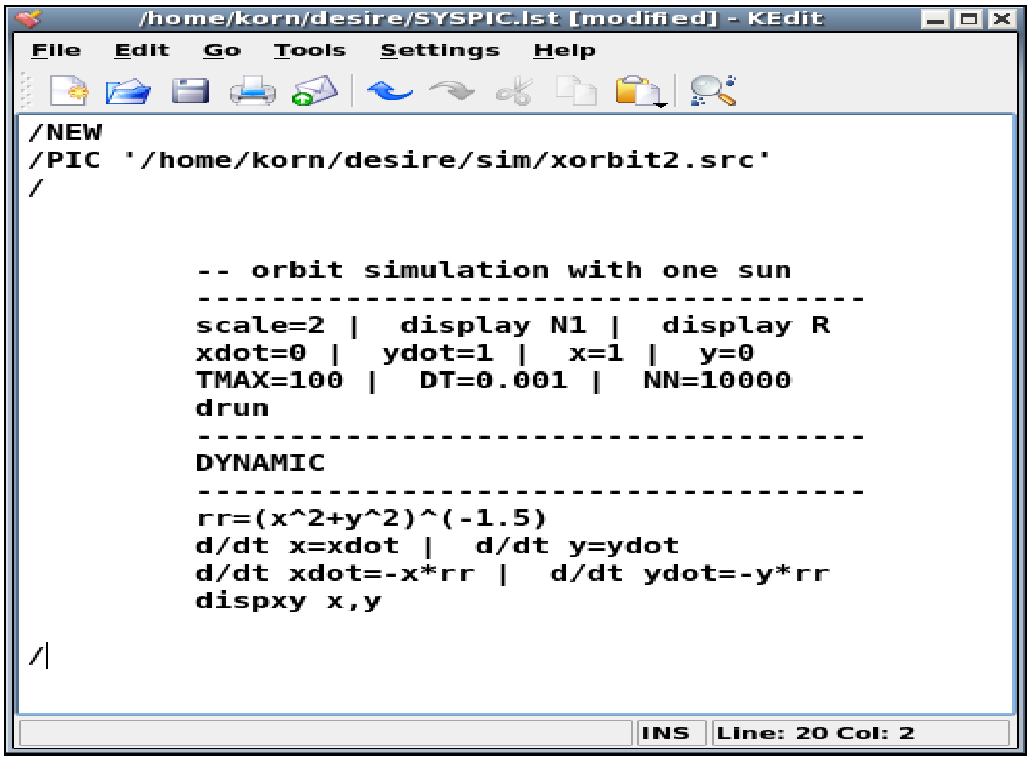


Figure 2. A Desire editor window displays a source program for editing and immediate execution. The program text is automatically placed between two slashes (***/***) and prefaced with its correct location code (PIC) and the command **NEW** which erases any previously loaded program. Note the editor-window save button. Note that you can also change the name and location of your user program on this screen.

rld './sim/lorenz.src' or **reload './sim/lorenz.src'**

The file extension for **rld** and **reload** defaults to **.src**, so that you could simply enter

rld ''/mydirectory/sim/lorenz' or **reload '/mydirectory/sim/lorenz'**

(c) Run the Program. Once loaded, the user application program ***compiles and runs immediately*** with graphics or listings when you type

run

Desire displays graphics in a fixed-size (656 by 600) ***Graph Window***, which appears automatically when you program or command graphics operations. New graphics displays automatically replace earlier ones. To ***remove the Graph Window***,[2] enter **Return**. Note that mouse operations will ***not*** move the Graph Window.

We like to use a screen resolution of 1280 by 960 or 1024 by 768 pixels. You might like to use different resolutions on the two screens of a dual-monitor display. With higher screen resolutions the Graph Window is smaller. Desire, like most programs, is at its best with dual-monitor displays. Dual-monitor displays install automatically with new versions of Linux.

To ***change the graph-window location*** use the programmed or command-mode display statement

display W *i, k*

where ***i, k*** are the desired screen coordinates of the window origin (upper left corner of Graph Window). You will want to set the graph-window location to your taste for different display sizes and single- or dual-monitor displays. To change the graph-window ***size*** change your screen resolution.

(d) You can work in other windows while Desire runs. You can even run multiple instances of Desire installed in new folders (say **/desire2**), each with its own **mk_syspict.sh** and **dscript.sh** script files. Use **display W *i, k*** to separate multiple Graph Windows by changing their locations.

1-3. Entering, Naming, and Saving a New User Program

To *enter a new program*, you could type numbered program lines in the terminal window (see below), but we do this only very rarely, e.g. to test short programs. It is much easier to type and edit your program in an ***editor window*** obtained with the keyboard command **ed** (Fig. 2).

Enter your text between the two lines starting with slashes (**/**). The editor-window ***save button*** then transfers the edited program to Desire. To ***execute the edited program***, enter **erun** (or **zz**, which is easier to type) in the terminal window. When the program runs the way you want enter

keep '*your directroy path/your filename*'

to save the edited program in a specified file. The file extension again defaults to **.src**.

[2] If a glitch, possibly in another program, should ever keep you from removing the Graph Window, simply log out and log in again.

1-4. File Specifications and Program Identification Codes

Linux ***file specifications*** have the form

filespec = directory path/filename.extension

Desire file specifications are currently limited to 100 characters. When you type only a file name in a Desire terminal window the directory path defaults to the current path. Desire accepts Unix "dot" path specifications, as in

rld './miffle' or **rld '../grunch.src'**

and also “multiple-dot” path specifications like **../../nut.src**.

Loading a user program with **rld** '***filespec***' or **reload** '***filespec***' assigns the program in memory the location code or ***program identification code (PIC) filespec***. A new un-named program entered into memory from an editor window is given the default PIC **no_name.src**.

The command-line or programmed statement **PIC** displays the currently active PIC, while

PIC '/mydir/kotz.ddd'

assigns a new PIC to the program in memory. The file extension defaults to **.src**.

Interactive Modeling with Desire Editor Windows

1-5. Desire Editor Windows

(a) Desire user programs are normally stored as plain-text files with the extension **.src**, e.g. **lorenz.src**. User-program files can be located in any folder or on your desktop. You can create project directories containing user-program files together with notes (text files), screenshots, spreadsheets with results, reports, special shell scripts, or any other files. ***A new Desire feature makes it dramatically easier to load Desire user programs from any directory without the need to type* rld *with a possibly long file specification***.

When you right-click a user-program **.src** icon, and select **Open with** GNOME gives you a choice of several ***file associations*** (programs opening the file). We associate Desire user programs (extension **.src**) with the shell scripts **dscript.sh** and **mk_syspict.sh** located in the **/desire** directory.[3] .

(b) We use **dscript.sh** as the default association. Then ***clicking or double-clicking a* .src *file icon automatically transfers the file to a system file (* SYSPIC.lst) *in your /*desire *directory***.[4] Now ***the simulation program executes immediately*** when you type **erun** (or more conveniently **zz**) followed by **RETURN** in the Desire terminal window.

---

[3] The writer is grateful to Dr. R. Wieland of ZALF/Muencheberg for his help in writing the useful **mk_syspict.sh** script.

[4] ***Watch this!*** The convenient scripts **dscript.sh** and **mk_syspict.sh** furnished with your distribution and listed in Appendix B are configured to pass your file to the Desire program in the folder **/home/username/desire, *and nowhere else***! If you install Desire in a different installation directory you must edit the script files to replace **~/desire** . You may also want to edit **mk_syspict.sh** to substitute a diffferent editor for **nedit**.

(c) Opening a **.src** file with **mk_syspict.sh** also ***transfers the program file to Desire via*** **SYSPIC.lst;** in addition it ***displays the file text in editor window*** (Fig. 2).

You can now execute the user program by typing **erun** or **zz**, or else first ***edit the program and then click the editor-window*** **save** ***command or button to transfer the edited file to*** **SYSPIC.lst** ***for execution.***

Desire simulations run immediately without noticeable compiler delay, so that ***repeated editing and execution permits very convenient interactive modeling.***

Note that your original user-program file is still safe in its folder. The command **rld** reloads it into memory; you can then view and edit it again by typing **ed**. The file you had been editing is also still available in its own editor window. The command **reload** reloads the original file and closes all open Desire editor windows.

After you have modified and tested a program you can save it in the location specified in the editor window with the terminal command

keep

You can still do more editing if you like. If you do not want to overwrite your original file, edit the location code (PIC) in the editor window before loading the program, or use the terminal command

keep '*filespec*'

This assigns the new address code (PIC) to the program in memory but leaves the editor window unchanged, so you can still modify the program.

NOTE: Editor windows called with **mk_syspict.sh**, **ed**, or **edit** ***must NOT use*** the editor's **Save as** menu entry to save files; for that prevents Desire from loading the edited program. If you have used **Save as** by mistake, use **Save as** with **SYSPIC.lst**, and all will be well again.

1-6. Using Multiple Editor Windows

You can open two or more **.src** files with **mk_syspict.sh** to display each in a new editor window.

NOTE: At this point, **kwrite** and **nedit** display a nag screen asking whether you want to overwrite the old editor window with the new one. ***Always click on*** "**ignore**".

Select the program in any one of the Editor Windows for execution by clicking its **save** command or button. ***Multiple editor windows make it especially convenient to edit, run, and compare different simulation models, or different versions of the same model.***

DETAILS ON USING EDITOR WINDOWS UNDER LINUX

1. The Commands

(a) To run or inspect a Desire user program file (**.src** file), select the file with the right mouse button. Then

- click on **dscrpt.sh** to load the program into Desire ready to run on a typed **erun** or **zz** command. Typing **ed** or **edit** then opens the file in an editor window.
- click on **mk_syspict.sh** to load the file into Desire ***and*** open it in an editor window.

In either case you can then run the program by typing **erun** (or more conveniently **zz**).

You can ***edit the program*** and then ***load the edited program into Desire*** by clicking on the editor's **save** button. Use **erun** (or **zz**) to run the edited program.

(b) You can load ***multiple editor windows*** by repeating this procedure. Kill the editor's multi-window nag screen by clicking on **ignore**.

(c) For interactive modeling and simulation you will run, modify, and rerun programs many times. To ensure that no useful old or intermediate code is destroyed in this process, Desire simulations keep program code in ***several different places***, viz.

- on disk
- in editor windows
- in the system file **SYSPIC.lst** in the installation folder.
- in the computer memory

Using an editor window to change and run a program does not affect the original program on disk.

Clicking on **dscrpt.lst**, **mk_syspict.sh**, or the editor'**save** button loads the soure code and its program identifiation code (PIC) into **SYSPIC.lst**. PIC is simply the path string ***filespec*** that specifies the source-file location on disk. When **mk_syspict.sh**, **ed**, or **edit** create an editor window for a program they automatically add its program identification code at the top. ***You can edit this proram identification code*** if you want to create a new file later. Next,

- **erun** (or **zz**) loads the source code and the program identification into memory and starts a sximulation run
- **load** only loads source code and program identification into memory

Note here that **rld *'filespec'*** and **reload *'filespec'*** load source code nd program identification into both **SYSPIC.lst** and memory, ready to run.

(d) The typed command **run** runs the program currently in memory. If you need to find out which progam is currently in memory, type **PIC**. **keep** saves this progam in the location specified by the program identification code. You can also store the program elsewhere by typing **keep 'filespec'**.

(d) You can open ***muliple editor windows*** with the same or different programs and edit and run any of thse programs to compare results. ***You will want to give diffeent versions of the same program new program identifications to prevent overwriting when you save programs with*** **keep**.

DETAILS ON USING EDITOR WINDOWS UNDER LINUX

(a) To run or inspect a Desire user program file (**.src** file), select the file with the right mouse button. Then

- click on **dscrpt.sh** to load the program into Desire ready to run on a typed **erun** or **zz** command. Typing **ed** or **edit** then opens the file in an editor window.
- click on **mk_syspict.sh** to load the file into Desire ***and*** open it in an editor window.

In either case you can then run the program by typing **erun** (or more conveniently **zz**).

You can ***edit the program*** and then ***load the edited program into Desire*** by clicking on the editor's **save** button. Use **erun** (or **zz**) to run the edited program.

(b) You can load ***multiple editor windows*** by repeating this procedure. Kill the editor's multi-window nag screen by clicking on **ignore**.

(c) For interactive modeling and simulation you will run, modify, and rerun programs many times. To ensure that no useful old or intermediate code is destroyed in this process, Desire simulations keep program code in ***several different places***, viz.

- on disk
- in editor windows
- in the system file **SYSPIC.lst** in the installation folder.
- in the computer memory

Using an editor window to change and run a program does not affect the original program on disk.

Clicking on **dscrpt.sh**, **mk_syspict.sh**, or the editor'**save** button loads the soure code and its program identifiation code (PIC) into **SYSPIC.lst**. PIC is simply the path string ***filespec*** that specifies the source-file location on disk. When **mk_syspict.sh**, **ed**, or **edit** create an editor window for a program they automatically add its program identification code at the top. ***You can edit this proram identification code*** if you want to create a new file later. Next,

- **erun** (or **zz**) loads the source code and the program identification into memory and starts a sximulation run
- **load** only loads source code and program identification into memory

Note here that **rld *'filespec'*** and **reload *'filespec'*** load source code nd program identification into both **SYSPIC.lst** and memory, ready to run.

(d) The typed command **run** runs the program currently in memory. If you need to find out which progam is currently in memory, type **PIC**. **keep** saves this progam in the location specified by the program identification code. You can also store the program elsewhere by typing **keep *'filespec'***.

(e) You can open ***muliple editor windows*** with the same or different programs and edit and run any of thse programs to compare results. ***You will want to give different versions of the same program new program identifications to prevent overwriting when you save programs with*** **keep**.

(f) Typing **reload** in the terminal window closes all Desire editor windows and ***reloads the disk version of the last-run source file*** into both **SYSPIC.lst** and memory, ready to run on a **run** command. This is a cnvenient way to undo all changes you may have made..You can then type **ed** or **edit** to re-edit the file.

1-7. Other Possibilities

The terminal command **rld** reloads the disk file corresponding to the current PIC (***not*** the edited file) for execution; note again that recently edited versions are still available in their editor windows. Typing **reload** does the same thing as **rld** but closes all editor windows. **reload '*filespe*c'** loads a new user program and also kills all editor windows.

To identify different versions of a program you can, if you like, change the program identification at the top of an editor window (Fig. 2). You can then run or **load** the program and save it with **keep**. The commands **rld** and **reload** return an error message if no file with the new identification exists.

Note also that you can open ***user-written help screens***, or in fact any text file, in an ordinary editor window while Desire runs. You can either click on the file icon and use **Open with *text editor***, or enter **sh gedit *filespec* &** in the terminal window. You can also drag the text file into a Desire Editor Window.

MISCELLANEOUS COMMANDS AND PROGRAM FEATURES

1-8. Desire Command Lines and Operating-system Commands

(a) Commands and Experiment-protocol Scripts (see also Chap. 7). Desire commands permit both ***interactive*** and ***programmed*** control of simulation experiments. Commands entered as part of a program form its ***experiment-protocol script***. The Desire command language described in Chap. 2 is a complete computer language in its own right.

Commands like **run** or **keep** typed into the Desire terminal window execute at once. Their effects can immediately displayed with **write** commands. Such command interpretation is precisely what you need for interactive computer-aided experimentation and for program debugging (Chapter 7). Simulation-model code in DYNAMIC program segments, on the other hand, is ***compiled*** for fast execution (Chaps. 3 and 4).

Desire commands and program lines may contain multiple statements separated by the ***statement delimiter*** **|** , e.g.

```
x = 15   |   write x
```

Anything following a ***comment delimiter*** **- -** and a subsequent space is a ***comment***.

(b) Operating-system Commands. Programmed or command-mode statements prefixed with **sh** (followed by a space) are ***operating-system commands***:

```
sh ls
sh cp foo.dat moo.dat
sh mcopy –t a:boffle.txt
sh xprogram
```

Your terminal-window text will change to reverse video to indicate that Linux or Unix is temporarily in charge.

In particular, the command-line or programmed statements

sh sh and **sh csh**

respectively give you ***Bourne (bash) and C command shells***. You can exit the Bourne shell with **crtl-D** and the C shell with **exit**.

Desire can thus access the full power of Linux or Unix. You can copy, rename, and edit files, create directories, or format media without leaving Desire. You can run batch files (shell scripts), and even Fortran or C programs such as special graphics programs. Desire communicates with other programs by reading and writing files (Chapter 2). Unix commands prefaced by **sh** can also be used in experiment-protocol scripts.

(c) Deleting and Renaming Files. In particular, UNIX commands can ***delete*** and ***rename*** files:

```
sh rm kotz.src
sh rm /mydir/bippy.*
sh mv /castle/prince.src  /castle/frog.src
```

1-9. Miscellaneous Commands

(a) The* time *Command. The command

time

displays ***the current date and time***. Used as a program line, **time** produces a ***date stamp***.

(b) Program Listings. Normally, editor windows are more convenient than long program listings, but the commands

list and **list+**

can be useful for debugging (Chap. 7). They respectively list the current program without and with Desire line numbers (Sec. 1-7).

Linux terminals can scroll through long listings. You can ***"pause" a long listing*** in the terminal window, by typing **ctrl-c**. You can then ***resume listing*** by typing **Return** ***twice***. To ***terminate a paused listing***, type any character followed by **Return**. But Desire listings are so fast that you rarely have time to pause.

You can ***list specified lines, and/or ranges of lines*** with statements like

list 213, 245, 300-410 **list+ 322, 410**

list '*filespec*' and list+ '*filespec*' list the program in a specified file, respectively without and with Desire line numbers..

(c) Clearing the Computer Memory. The command

new or **new '*filespec*'**

closes any open editor window, ***erases the current program*** (if any), and sets the problem identification code respectively to **no_name.prc** or to ***filespec***. Desire asks you for confirmation before erasing. The command **NEW** clears the computer memory ***without*** confirmation. **clear** erases all variable, array, function, etc. definitions.

(d) The* run *Command. The command **run** ***executes ("runs") the program currently in memory***. Typing **ctl-c** ***pauses program execution;*** you can then ***resume execution*** by typing **Return** (see also

Sec. 3-1b). Typing any character (e.g. **space**) followed by **Return** ***terminates*** a paused run. Programs also pause when they reach a programmed **STOP** statement (see also Sec. 7-3). The terminal command **go** restarts program execution after **STOP**.

1-10. Desire Line Numbers and Special Editor Windows

Editor windows called with **edit** instead of **ed** show ***Desire line numbers***. These line numbers are created by Desire; ***they are not the same*** as the line numbers optionally supplied by editors such as **kedit** or **gedit**. Every Desire program has internal line numbers, even though you rarely see them. Explicit Desire line numbers are used in error messages, for debugging (Chap. 7), and for combining program segments (Sec. 1-11). The command **list+** produces a program listing showing explicit line numbers.

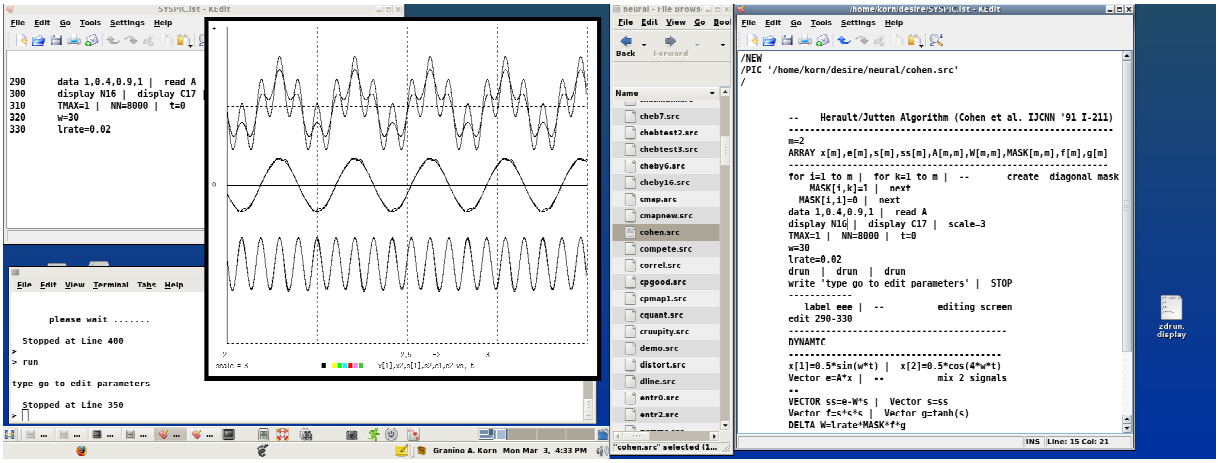


Figure 3. Linux screen showing a Special Editor Window (upper left), as well as terminal, graph, filemanager, and Desire editor windows.

keep+ creates a text file ***filespec*.lst** containing the current program listed with explicit Desire line numbers. **keep+ '*newfilespec*'** or **list+ '*newfilespec*'** saves the current program, with line numbers, as a new text file; the file extension defaults to **.lst**.

Desire line numbers are also used in commands-mode or program lines like

edit 100, 230 - 240

to create ***special editor windows*** that display selected program and comment lines for editing. A special editor window's **save** button passes the selected lines to Desire (i.e. inserts them into **SYSPIC.lst**). A subsequent command **erun** (or **zz**) overlays only the selected lines of the program in memory and then runs the revised program.

Special editor windows

- let you ***edit only selected program lines***, so that you do not need to scroll through long programs.
- can display not only program lines but also comments or instructions for the user.

You can again have multiple editor windows on the screen and load and execute their contents. Make sure, though, that the lines from a special editor window do not overlay the wrong program. Editor warning dialog boxes are discussed in Appendix A.

Note that statements like **ed 100, 230 - 240** are ***illegal***.

COMBINING PROGRAM SEGMENTS

1-11. Internal Line Numbers and load *Operations*

(a) Internal Line Numbers. Even Desire programs ostensibly without line numbers have ***internal line numbers*** that normally start with 210 and increment by 10. You can see them by entering the command **list+.**

(b)* load *Operations. The command

load '*filespec*'

loads a source file into memory without erasing the current program. Drive and directory default to the current default setting, and the file extension defaults to **.src**.

If the source file has explicit line numbers (e.g. if it was created with **keep+**), they are preserved. If there are no explicit line numbers, **load** creates internal line numbers starting with **210** and incrementing by **10**. Before a **load** operation the initial line number and the line-number increment can be changed with ***line-renumbering commands*** like

300 (line numbers now start with **300**) **auto 20 (line-number increment becomes 20)**

The command sequence

```
new
load 'file1'
load 'file2'
```

erases the current program and then ***combines program files***. Normally, **file2** is a program with line numbers starting above 200, and **file1** is a library file containing **FUNCTION**, **PROCEDURE**, and/or **SUBMODEL** definitions with line numbers below 200.

Source files created with an editor window (not with **list+**) may contain ***commands*** prefixed with a slash ***/*** ; such commands ***execute immediately*** as they are **load**ed.

You can also use an editor window's **save** button followed by the Desire command **load** to load an edited program into memory without executing it.

RTFM

Desire is a large program with many advanced features (in fact, several textbooks describe applications to control systems, biological systems, Monte Carlo simulation, and neural networks). You ought to look at the entire ***Desire Reference Manual***, which is included with your distribution.

APPENDIX A. LISTINGS OF DESIRE SCRIPT FILES

mk_syspict.sh (passes a program file to Desire and opens an editor window)

```
DIRECTORY=~/desire/SYSPIC.lst
echo "/NEW" > $DIRECTORY
echo -n "/PIC '" >> $DIRECTORY
echo "$1'" >>$DIRECTORY
echo "/" >> $DIRECTORY
cat $1 >>$DIRECTORY
echo "/" >> $DIRECTORY

kwrite  ~/desire/SYSPIC.lst
```

dscript.sh omits the last line of **mk_syspict.sh** and thus passes the program file to Desire without opening an editor window

ekiller.old (kills all **nedit** windows)

```
kill -KILL `ps -e | egrep nedit | mawk '{print $1}'`
```

NOTE!!! Some Linux distributions ***use "gawk" instead of "mawk"***

editcfg1

```
kedit SYSPIC.lst
```

APPENDIX B. OBSOLETE FEATURES

The following program features are no longer used. Since they cost little memory and no speed they were kept for compatibility with earlier versions of the program.

Console-terminal Program Entry, Listing, and Editing Using Explicit Line Numbers.

We normally edit programs in editor windows, but the experiment-protocol interpreter alone can also do simple program entry and editing in a terminal window. This can still be useful for small interactive program changes when Desire has returned a numbered program line with an error message.

(a) With the memory cleared, you can ***enter or add new program lines with line numbers***, say

```
50      X = 15
55      yak2 = 7   |   variab16 = -15.55 + X
60      write (X + yak2) * cos(variab16)
```

Unlike command lines, such numbered program lines execute only after you type **run**.

Program lines may contain ***multiple statements*** separated by |. A line can have up to 244 characters, which "wrap around" on the CRT screen. Desire program listings then show _ as a ***continuation-line symbol***.

NOTE: Remember that you can change drives and directories and copy, rename, and delete files without leaving Desire by using shell commands prefixed by **sh**.

(b) To see what you have actually entered, type **list** to list all numbered lines. You can, instead, ***list selected lines and/or line-number ranges*** with commands like

```
list 434-439, 210, 300- 400
```

(c) While you enter a line, you can edit it by using the **backspace** key and entering or deleting text before you type **Return**.

(d) Desire lets you

1. ***overwrite a program line*** by typing a new line with the same line number.
2. ***delete a program line*** by typing its line number followed by **Return**.

To ***delete more than one program line***, use commands of the form

erase 332- 339, 378, 400- 500

(this would delete Lines 332 through 339, Line 378, and Lines 400 through 500).

keep saves the edited program.

Auto-line-number Mode and Renumbering

(a) When you type **/** (followed by **Return**), numbered prompts invite you to enter program lines with ***automatically supplied line numbers***, starting with

210>

Successive line numbers then increment by 10. Lines with errors are not entered. You can restart with any desired line number by simply including it on your line, e.g.

450> **85** **y = 7 | z = 0-19.1**

or by typing the line number followed by **Return**. To produce line-number increments other than 10, say 50, start or restart auto-line-number mode with

auto 50 or **/auto 50**

Reusing any line number overwrites the previously-entered line with the same number, so that you can erase or edit lines.

To leave auto-line-number mode, type a slash / followed by **Return**. To ***execute commands while you are in auto-line-number mode***, prefix any typed command with a slash /, e.g.

/list or **/run**

This will also cause Desire to quit auto-line-number mode.

Renumbering Program Lines

To ***renumber the lines*** of a program currently in memory as **210, 220,. . .,** use

keep+ 'dummy' (creates **dummy.src** without line numbers)
reload 'dummy.src' (line numbers are now **210, 220, . . .**)
keep

You can, instead, start the new line numbers with, say, **2000** by using

keep+ 'dummy' (creates **dummy.src** without line numbers)
1990 (Return)
load 'dummy' (note that **load** defaults to **.src**)

Line numbers still increment by 10; you can use **auto** to change the increment if you wish.

Loading and Saving Programs in Condensed Form

The command

save '*filespec*'

saves a program in condensed precompiled form (for later retrieval by **old** or **chain**). If you do not specify a file extension, the extension defaults to **.prc**. The path defaults to the current default path. Given without a file specfication, **save** saves the program currently in memory in the current default directory under its current program name, with the default

extension **.prc**.

The command

old 'pa'

loads a user program **pa.prc** stored in condensed form with the default **.prc** extension into memory.

Program Chaining

Program chaining loads and runs two or more successive **.prc** programs or program segments from disk; common data can be preserved. Nowadays we prefer to use programs with multiple DYNAMIC segments.

The command

chain '*filespec*',*n*

loads the user program ***filespec*** saved earlier ***in precompiled form*** (**.prc** format, Sec. 1-8) The program executes immediately, starting at Line Number ***n***. If ,***n*** is omitted then execution starts with the first line of the new program. The old program is overwritten, but ***all variable and array names and data are preserved for use in the new program***. If you do *not* want the old data, use a **clear** statement before **chain**, or in the new program. The problem identification code (PIC, Sec. 1-15) changes to that of the new program. The default file extension for **chain** is **.prc**.

User-defined functions and procedures will not survive chaining. In fact, ***Desire keeps you from re-using their names*** (and those of their parameters) until you erase all declarations with **clear** (Sec. 2-9). Chaining causes recompilation of DYNAMIC program segments.

Table 1-1. A PARTIAL LIST OF DESIRE FILES

1. HELP SCREENS

There are duplicate copies in the default installation directory and in the **.\help** subdirectory, if any. Note that y***ou can easily add your own help-screen files***; any ASCII text file works!

howe	**if**	**program**	**next**	**filename**
files	**proc**	**why**	**types**	
edit	**menu**	**loops**	**nonlin**	**irule**
fun	**stiff**	**display**	**keywords**	**programs**
vector	**show**	**examples**		

(neural-network help screens keyed to **.src** examples with similar names):

parity3	**creepmin**	**xor**	**art**	**parity**
bpmap	**cpmap**	**pay**	**vquant**	**nquant**
entropy	**babybam**	**code**	**sweeptst**	**hquant**

2. EXAMPLE USER PROGRAMS

All examples are **.src** files. In the following partial list of examples, **?** and ***** are standard Unix or Windows wildcard metacharacters standing, respectively, for any legal character and any legal string.

sincos	graph-plotting demonstration
scalef	automatic graph scaling
fft?	FFT demos
conv	convolution via FFT
invert?	matrix inversion (single precision)
freq	complex numbers, amplitude and phase
plot?	conformal mapping
signal	signal generators
vdp, multivdp	Van der Pol's equation
lorenz	chaos-type problems
scroll	
achaos	
mglass	Mackay-Glass time series (chaos)
quant	quantization (a/d converter simulation)
prob	histogram (unsupported feature)
avg	signal averaging demo
minmax	running max/min, function storage
fun? , newfu*	function generators

store1, get1	function storage demo
delay, tdelay	delay demos
backl	backlash
oscilltr, damp, couple, signal	oscillators
cannon?	18th-century ballistic-weapon system
bouncer	bouncing ball
orbit?	translunar space-vehicle orbits
to*	torpedo simulations, with and without control, Monte Carlo
airpln	flight simulation with "**stash**"
airpln1	flight simulation without "**stash**"
billiards?	billiard simulation
pilot?	pilot-ejection benchmark, crossplot
pendu?	Pendulum simulations
euro*	EUROSIM comparison problems
del1500	delay line, 1499 differential equations
del2000	- same with 1999 differential equations
line?	- same in vector/matrix notation, up to 6000 equations
gluc	glucose-tolerance test
epidem	epidemic propagation
physb?	blood-circulation model
host?, prey?	Volterra/Lotka equations (predator-prey ecology)
world	Forrester-type world model
orbit*	space-orbit simulations
matcoup?, reson	vector differential equations
matdamp	vector differential equations
***x**	stiff-integration examples
stiff?	stiff-integration examples
bangstep*	bang-bang servo, variable-step integration
triga*, step	nuclear-reactor simulation, stiff
gear?, fuel?	stiff-integration examples (nuclear reactor)
satser?	servo simulation
howe	satellite roll controller (demonstrates submodels, delay)
servo*	servo impulse and frequency response (uses fft)
lib	transfer-function submodels
optrun	iterative multi-run optimization of a servo
fuzz*	fuzzy-logic control simulations

(Neural-Network Programs)

in /*neural *directory

xor*	XOR with backpropagation, with and without batching
xormin	XOR, creeping random search
parit*	parity learning, mixed backpropagation tricks
cruupity	parity learning, creeping random search
ch*	Cottrell/Zipser encoding, various types of learning, Chebyshev and nonquadratic criteria
sweeptst	different activation functions and their derivative
bpmap*	backpropagation function-mapping networks
bpxy*	
soft	different softmax classifiers with probability output, associative memory
rdf*, pdf*	different radial-basis-function networks
narendra?	Narendra control-system optimizer
autoas*	linear classifiers and associative memory
statgood	vector quantization
vquant?	vector quantization with and without conscience (FSCL)
pquant*	competitive learning for parttern recognition
multiwin	competitive learning with multiple winners
hquant?	hierarchical vector quantization
entropy	entropy and histogram
cohen*, herault	time-history analysis
qual1	competitive learning of nonlinear regression
predict*	linear and nonlinear predictors
nconv	2-dimensional correlation detector
hry*	principal components
bam?	bidirectional associative memories (BAMs)
cbam	competitive-layer BAM
hobam	BAM with nonlinear feedbacl
olambam	BAM using optimal linear transformation
alpha*, art*	pseudo-ART, alphabet and spiral learning
jetsx	new Jets/Sharks demo
neuron*	pulsed neuron
hop*	Hopfield-type networks
?feed	Grossberg-type signal enhancement
cp*	counterpropagation mapping networks

spiral* spiral-learning counterpropagation networks
qiral?
piral?
artmap?

fuzz* fuzzy-logic examples
fizz*, fazz,* fozz*
autoas* these character-recognition programs are two-layer and three-layer autoassociators, illustrate thresholding and competitive noise reduction techniques

vq* competitive template matching used to measure entropy and mu tual information

tessel* demonstrate Voronoi and Haken-type tessellations

(Partial Differential Equations, Method of Lines)

in* /pde *directory

heat* heat conduction experiments
hexch* heat-exchanger simulation
pde, advec* hyperbolic PDEs

(Monte Carlo Simulations and Statistics Examples)

- *in* /montecarlo *directory*

(Parzen-window Measurements)

- ***in* /parzen *directory***
-

(Nuclear Reactor Simulations)

triga*, fuel*, step

(filter models)

- *in* /filter *directory*

Chapter 2. Interpreted Experiment-protocol Scripts

SIMPLE INTERPRETER OPERATIONS

2-1. Interpreter Language

Desire experiment-protocol statements are ***interpreted*** line by line as they execute. Statements entered ***without line numbers*** are ***commands*** and execute immediately. Numbered lines, on the other hand, form a ***program (experiment-protocol script)*** that executes when you type **run**. Anything to the right of a ***comment delimiter*** **--** is a ***comment***, and is ignored by the program. In particular, lines of dashes **------** are comments.

Every Desire program has an ***experiment*** protocol script, which sets up parameters and arrays and can then call simulation runs to "exercise" models defined in runtime-compiled ***DYNAMIC program segments***. A simple example is

```
NN = 300  | TMAX = NN |  t = 1
gain = 1.25
drun
```

Desire interpreter programs can also stand alone to do useful general-purpose computation:

```
bippy = 23.9 + sin(PI/3)
CHEESE = (MILK + grease) * dye
```

Desire's scripting language is a complete mathematical language. It acts much, like a Basic dialect augmented by complex-number, vector, and matrix operations. As noted in Chap. 1, you can enter or list Desire programs ***without line numbers***; the system internally numbers lines automatically when you **load** or **reload** a program.

Good software lets you ignore advanced language features until they are actually needed. In particular, all Desire identifiers refer to REAL 64-bit floating-point numbers and functions unless a symbol is specifically ***declared*** to mean something else like **ARRAY**, **COMPLEX**, **INTEGER**, **FUNCTION**, **PROCEDURE**, **SUBMODEL**, or **label**. Desire displays a reference list of all Desire keywords on the monitor screen when you type

```
help keywords
```

or double-click on the relevant help-screen icon.

2-2. Expressions and Library Functions

(a) Desire ***expression assignments*** resemble those in Fortran or Basic but are slightly more flexible.

Symbolic identifiers must start with a letter and may have up to 20 upper- and/or lower-case alphanumeric characters and **$** signs. ***Desire is case-sensitive***, so that **dwarf7**, **DWARF7**, and **Dwarf7** are three different identifiers. Expressions may contain blanks:

Elevator = 3 * coeff7 * - 14.4 * (sin(w * t) + Y * a$bcdE) - 1.2E-07

Program listings automatically eliminate redundant blanks and indent ("prettyprint") loops and subprograms to make them easier to read.

(b) Desire expressions admit **PI = 3.14159...** and ***octal*** and ***hexadecimal*** as well as decimal integers (Sec. 2-13). Rules for operator precedence and the use of parentheses are those of normal mathematics or Fortran. Note that

2^2^3 = 2^(2^3) = 256

just as in Fortran (some Basic systems would return 64). But you need not to remember the convention; simply use parentheses!

Like most computer languages, Desire implements **x^y** to compute x^y for any exponent **y** other than **y** = 2 using logarithms. This will return an error **if y ≤ 0**.

(c) Desire normally returns an error when the program encounters an undefined identifier. But recursive assignments like

X = X + 5000

are legal because **X** is defined on the left-hand side. If **X** was not previously defined, it is initialized to 0 on the right and is then assigned the value **5000**.

(d) Desire library functions (e.g. **sin(x), cos(x), . . .)** are listed in Table 2. Type

help fun and **help nonlin**

or double-click help-screen icons to display library-function definitions on the monitor screen. In addition, you can easily define new functions (Sec. 2-10).

2-3. Console, File, and Device Output

(a) Programmed or command-mode **write** statements display, print, or store the results of computations.

```
xvar = 222.22 * 3 -  4.7
write xvar
```

or

```
write 222.22 * 3 - 4.7
```

produces the result on the monitor screen. Like a Basic **PRINT** statement, the Desire statement

```
write 'The moon is pink'
```

displays a ***text string***. You can write ***multiple items on a line:***

```
write 'A = ';A,'B = ';B,'C = ';C
```

A semicolon produces the next item immediately following the last one, and a comma inserts a tab between substrings. Completion of each **write** statement causes a carriage return/line feed (newline), unless the **write** statement ends with a semicolon.
Specifically, **write** alone causes a newline. **write ;** (note the blank) would return an error message.

Desire **write** statements ***correctly recognize different previously declared data types*** including **INTEGER**s, **COMPLEX** numbers, and arrays and produce the correct output, as in

```
AA= - 231.77
ARRAY vector[800]
.................
write AA, vector
```

(b) To write to an ASCII data file or device (serial printer, data link), you must first **connect** (open) a "channel" (really a buffer in memory) for the file or device. Programmed or command-mode statements like

```
connect 'BRIE.dat' as output 4
connect 'lpt2' as output 5
```

associate named files or devices with ***channel numbers*** between 0 and a maximum value (currently 10). Matched double quotes may replace single quotes. Once a file or device is connected, you can write to it with a command-mode or programmed **write** statement referencing the numbered channel:

```
write #4,'x14 has the value ';B
```

A comma must follow the channel number. You can write more than one such line.

NOTE: Channels 0 and 1 are no longer reserved for console input/output. Instead, they open for ***append*** rather than simple write. This lets you to append text to files after the file has been **disconnect**ed (closed); you simply reconnect it. We use this feature mainly to produce spreadsheet and database files.

(c) Desire **write** statements recognize previously declared array names and label each array with its type, name, and dimension or dimensions. This is nice for display and for printed listings, but ***files*** written this way are not easily machine-readable. For this reason, Desire also provides a modified **write** statement of the form

```
write ##3, array1, array2, . . .
write ##4, a, b, array, . . .
```

(note the double **##**), which simply writes successive array elements or scalar values separated by newlines to produce machine-readable ASCII files (e.g. for spreadsheet programs) without header labels. Conventional **write #** statements write as before.

(d) When you are finished with your file or device, **disconnect** (close) its channel with a statement like

disconnect 3 or **disconnect 3, 4, . . .** (disconnects multiple channels)

Each **disconnect** operation flushes data remaining in a buffer or buffers, releases the channel number and, for file output, makes the proper mass-storage directory entry.

The current default device and path are implied if none are specified. File extensions for **connect** statements default to **.dat.** As we noted, Desire no longer reserves Channels 0 and 1 for console input and output.

STOP, **new**, and **NEW** statements do ***not*** automatically **disconnect** (close) open files. But ***any error, or exit from Desire, automatically*** disconnect***s all open channels and thus writes output files.***

(e) **write** statements like

```
write %%100
```

write ***hexadecimal equivalents of decimal integers***. Statements like

```
write $27   |   write #4,$&33   |   write #5,$&&1B
```

(decimal, octal, or hexadecimal integers preceded by **$**) produce **the *ASCII characters denoted by the respective decimal, octal, or hexadecimal character code***. This is used, in particular, for printer and device control.

2-4. Console, File, and Device Input

(a) An ***input statement***,

```
input  x, y, z, . . .
```

produces successive console-terminal prompts like

```
x?
```

which asks you to type a value or expression for the variables **x, y, ...** Desire then assigns the resulting numbers to **x, y, z, ...** You need not wait for successive prompts; you can continue to enter successive values or expressions, separating them with carriage returns. You are prompted for the next item needed. Any error results in an error message and a new prompt for the same item. You can ***abort*** the input procedure by typing **!**.

If **X** is a (previously declared) ***array***, then **input X** produces successive prompts **A?** until the array is filled; note that such array prompts do not specify the array index.

(b) Assuming that a device or file was **connect**ed **for input**, the file/device **input** statement

input {#*channel number*,} *variable* {,*variable*, ...}

successively reads ASCII values or expressions for each variable from the specified channel. Successive file or device entries must be separated by line feeds, commas, semicolons, or horizontal tabs. Line-feed-delimited files must be terminated with a line feed. If end-of-file (EOF) is reached, Desire returns an error message.

You can read arrays from text files with statements like

```
input #3,gee
```

or

```
for i=1 to 10 |  input #3,gee[i] |  next
```

EXAMPLE:

```
connect 'GRUNT.dat' as input 3
input #3, Q, P, R, S
disconnect 3
```

Desire can thus read tab-delimited text files output by spreadsheet programs like Excel. Excel, in turn, can read text files created with Desire**.**

Note that different c systems may have different end-of file (EOF) markers; in case of problems, pad data files with extra 0 entries. Examples **input***, **output*** demonstrate different cases.

Files or devices **connect**ed **for input** must be **disconnect**ed after use, as in Sec. 2-3c.

(c) Desire input ***statements can read real and* INTEGER *expressions, subscripted variables, and arrays***, but ***not*** complex data. Arrays are read ***in row-major sequence***. One **input** statement can read data of several different types. File or device input should supply a sufficient number of array items. To permit readable grouping of input requests, ***you can replace any comma in an input list with a semicolon***.

2-5. Conditional Branching with if *Statements*

if statements let programs ***branch on conditions*** like

expression1 RELATION expression2

where ***RELATION*** stands for one of the operators **<, >, =, <=, >=, <>** (**<>** means "not equal"), as in

```
if Crit >= 0 then x19 = x * 1000   |  else proceed
```

The **then** or **else** clauses defining program branches can involve multiple statements:

```
if A21 < 0 then p = p + 1   |   q = q + 50
   else p = p-1   |   write 'exception!'
   proceed
```

Code between **then** and the corresponding **else** executes if and only if the specified condition is true. If the condition is not satisfied, the program continues on the far side of **else.**

proceed statements, which simply let the program continue, may look useless, but they neatly delimit ***nested* if *clauses***. There must always be exactly one **else** and one **proceed** for each **if** in the program. This syntax makes the small experiment-protocol interpreter very efficient.

In ***nested* if** statements, the outermost **if** always corresponds to the outermost **else** that is still free. The **else** clause terminates with the corresponding outermost **proceed**. Desire listings automatically indent ("prettyprint") **if** clauses to sort out nested statements.

EXAMPLE:

```
Q = 10 | P = 1
if Q = 13 then if P = 0 then write 'Q = 13, P = 0'
      else write 'Q = 13, P <> 0'   |   proceed
   else if P = 0 then write 'Q <> 13, P = 0'
      else write 'Q <> 13, P <> 0 ---  whee!'
      proceed
   proceed
write 'done'
```

When an **if ... then ...** construction does not fit on one line you can use a comment (**--**) as the first statement of the **then** clause.

Desire **if** statements admit only the simple (two-term) logical expressions ***condition1*** **or** ***condition2*** and ***condition1*** **and** ***condition2***. ***More general logical expressions are illegal.*** Legal examples are

```
if a <> b and Q = P then write 'whee'   |   else proceed

if FOO < 100 or FOO > 100 then write 'FOO is not 100!'
   else write 'FOO = 100'   |   proceed
```

2-6. Labels and Branching

(a) Desire permits ***unconditional branching to a numbered line***,

```
go to 1100
```

One can substitute a numerical expression ***in parentheses*** for the line number ("computed" **go to**).

It is usually best to use line-number branching only for debugging, typically using command mode. We prefer ***line-number-free programming***: we declare named ***symbolic labels*** and branch to them. Listings automatically offset labels to show them more clearly. A label declaration must be the only statement on its line, except for possible comments. The following example programs a simple loop

```
i=12
   label Wippy
write i   |   i = i + 40
 if i < 509 then go to Wippy
    else proceed
 write 'Whoop!'
```

go to in command mode can restart an experiment without repeating lengthy initial setup operations.

(b) Typing **go to eee** and **go to jjj** ***executes user routines following these special labels. This is used, in particular, to display user-written editing screens and menus.*** You can also use it to execute special test programs, to run alternate models, or to print something, e.g. selected data items.

2-7. **for, while,** *and* **repeat** *Loops*

Desire admits PASCAL-like ***loops***:

```
for y = 0 to PI/4 step PI/40
  write 'y= ';y,'sin(y)= ';sin(y)
  next
k=1
  while k <= 80
  write k, k * k, 1/k
  k = k + 1
```

```
  end while
k = 0
repeat  | write k, k * k, 1/k  |   k = k + 10  |  until k >= 100
```

Unlike Basic, Desire conveniently uses **next** rather than **next y**. **while...** and **until...** conditions may use a single **and** or a single **or**, exactly like **if** (Sec. 2-5). More general logical expressions are ***illegal***.

In ***command mode***, Desire admits only ***one-line*** loops like

```
for i = 1 to 10   |   a = 2 * i   |   write a   |   next
```

and prints a warning message.

Loops can be nested but, as in Fortran or Basic, they must not overlap other loops or **if/else** clauses. Be sure to provide a unique **else**, **proceed**, **next**, **end while**, and **until** corresponding to each **if**, **else**, **for**, **while**, and **repeat**. Desire returns error messages on loop overlaps and on duplicate **next**, **end while**, and **until** statements if the program actually reaches faulty code. An interpreter program, though, may never actually encounter errors in "dead" code.

go tos into **if/else** clauses, loops, or procedures cause trouble exactly as in Basic or Fortran. Desire error messages catch many such problems. Desire does admit **go to**, since this may be efficient; but go-to-less “structured” programming is also readily possible. Specifically, Desire's **exit** statement produces a go-to-less exit from the current loop. **exit** may be followed only by comments, **else** clauses, or **proceed** on the same line.

2-8. Subscripted Variables and Arrays

Before ***subscripted variables*** like **vectr1[12], A[7,4]** are used in assignments and expressions, the interpreter program must declare ***arrays*** with statements like

```
ARRAY vectr1[1200], A [m, n]
```

Arrays can have up to 10 dimensions, but only one-and two-dimensional arrays are used in **Vector**, **MATRIX**, **DOT**, etc. expressions (Chaps. 5 and 8). There are also **STATE** array declarations for vector/matrix differential equations (*q.v.*).

You can freely employ array ***elements*** as subscripted variables in scalar expressions, both in the interpreter program and in DYNAMIC program segments:

```
template[2, 3] = 23
alpha = cos(layer1[m + 2]) + beta
```

Subscripts vary between 1 (***not*** 0) and a dimension value ≥ 1. Dimensions and subscripts may be expressions. ***The same name must not be used for both a variable and an array***. recall that Desire stores array elements in ***row-major sequence*** (the last subscript changes most quickly) rather than in column-major sequence. This makes matrix printouts look more "natural".

Arrays are initialized to all 0s. They can be "filled" through assignments to individual array elements, or through **data/read** statements (Sec. 2-9), **input** statements (Sec. 2-4), or through **store** statements in the DYNAMIC segment.

EXAMPLE:

```
N =100
ARRAY vectr1[2 * N]
for i = 1 to N
  vectr1[i] = i
  vectr1[i + N] = cos(i)
  next
```

Desire also still recognizes the **dimension** keyword used instead of **ARRAY** in the earliest Desire systems.

2-9. The clear *Statement*

The programmed or command-mode statement **clear** clears and thus destroys all array definitions (and also definitions of variables, labels, user-defined function, and procedures) and reclaims their memory allocations. A **clear** statement must precede any attempt to ***redimension an array***.

2-10. data *Lists and* read *Assignments*

data *lists* like

data *expression1, expression2, ...*

supply numerical values for **read** assignments like

read *variable name*
read *array name*
read *variable or array name, variable or array name, ...*

Successive **read** statements assign values of **data** expressions sequentially to the variables in the **read** list. The system ***read pointer*** normally starts at the first expression of the first **data** list and searches for the next **data** list as each **data** list is exhausted. An error message indicates insufficient data.

EXAMPLE:

```
x = 117.222
data 11, 12, 13, 14
y = 100. 222
data 15, 16, x - y, 18
read x1, x2, x3, x4
read x5, x6, x7, x8
```

is equivalent to

x1 = 11, x2 = 12, x3 = 13, x4 = 14, x5 = 15, x6 = 16, x7 = 7, x8 = 18

Desire* read *operations, like* input *statements, can read real and* INTEGER *expressions, subscripted variables, and arrays, but not **COMPLEX** data. Arrays are read in row-major sequence. A single **read** statement can read data of several different types.

For convenience in separating data for different arrays, or to indicate array rows or columns, you can substitute semicolons for commas anywhere in a **data** or **read** list.

EXAMPLE:
```
ARRAY Amatrix[3, 3]
x = 27.5  |  y = - 20.5
data 200.19 ; 10, 20,3 0 ; 40, 50, 60 ; x + y, 80, 90
read velocity, Amatrix
```

is equivalent to **velocity = 200.19** and

```
Amatrix[1,1] = 10    Amatrix[1,2] = 20   Amatrix[1,3] = 30
Amatrix[2,1] = 40    Amatrix[2,2] = 50   Amatrix[2,3] = 60
Amatrix[1,3] = 6.9   Amatrix[2,3] = 80   Amatrix[3,3] = 90
```

The programmed or command-mode statement

restore

resets the data pointer to the first expression in the first **data** list, much as in Basic. But Desire also has

restore ***labelname***

and **restore** ***line-number expression***

which reset the data pointer to the first **data** item on the program line referenced by a **label** (Sec. 2-5) or a line number. You must use parentheses around line-number ***expressions***, as in the case of **go to** statements.

The data pointer always automatically resets to the first **data** expression when any program line is modified (as during debugging).

Examples of User-defined Functions

(such functions can be collected in library files for reuse)

```
FUNCTION  tan(xx) = sin(xx)/cos(xx)
FUNCTION  abs2Dl(uu, vv) = sqrt(uu * uu + vv * vv)
FUNCTION  min(aa, bb) = aa - lim(aa - bb)
FUNCTION  max(aa, bb) = aa + lim(bb - aa)
FUNCTION  asat(xx, aa) = aa * sat(xx/aa)          (aa > 0)
FUNCTION  BOUND(xx, aa, bb) = aa + lim(xx - aa) - lim(xx - bb)
```

2-11. User-defined Functions

Experiment-protocol scripts can create ***user-defined functions*** with **FUNCTION** declarations like

```
FUNCTION  abs2Dl(u$, $v) = sqrt(u$^2 + v$^2)
```

The function definition must fit one program line - but that can be a long line "wrapping around" on the display. Function definitions can include constant parameters and also other vasriables. We marked the ***dummy arguments*** **x$**, **y$**, **z$** with dollar signs so they can be recognized easily, but this is not necessary. Dummy arguments must not be subscripted. Dummy-argument names are "protected" to prevent "side effects". This

means that any attempt to use their names after the function definition produces an error message.

Once declared, your new function can be ***invoked*** in the experiment protocol or in a DYNAMIC program segment, say with

```
RR = abs2D(x, y)
```

which would be equivalent to **RR = sqrt(x^2 + y^2)**. DYNAMIC-segment code is generated by the Desire compiler, so there is no runtime function-call/return overhead.

The invocation arguments replacing the dummy arguments can be any expressions legal in the invocation context; they can thus be subscripted variables. In experiment-protocol scripts, invocation arguments can be previously declared complex numbers or integers as well as real numbers. In DYNAMIC program segments, invocation arguments may even be vectors.

FUNCTION definitions may be ***nested***, i.e., they can contain previously defined functions. But ***recursive definitions*** like

```
FUNCTION f1(x) = f1(x) + 1
```

or

```
FUNCTION  f1(x) = f2(1) + x     |     FUNCTION  f2(y) = f1(y+1)
```

and also ***recursive function calls***, as in

```
FUNCTION  incr(x) = x + 1  |     q = incr(incr(incr(y)))
```

are ***illegal.***.

2-12. Script Procedures

User-defined, named ***procedures*** to be used in Desire interpreter programs are declared in the form

```
PROCEDURE procname(x, y, ...; X, Y, ...)
  ... procedure code involving x, y,...,X,Y, ...,
            and other ("global") variables
  end
```

Desire listings automatically indent the procedure body. The semicolon separates parameters **x, y, ...** which will be called ***by value*** (typically expressions serving as procedure inputs) and parameters **X, Y, ...** which will be called ***by name*** (like **VAR** parameters in PASCAL, e.g. procedure outputs). Here is an example:

```
PROCEDURE  xform(r, theta; X, Y)
   X = r * cos(theta)
   Y = r * sin(theta)
   end
```

An **exit** statement anywhere within a procedure causes an immediate return. User-defined procedures can have only value parameters, only **VAR** parameters, or no arguments:

```
PROCEDURE foo(p, q)
```

```
PROCEDURE fee(; g, h, i)
PROCEDURE fum()
```

Redundant semicolons and parentheses must not be omitted.

Once defined, a procedure can be called later in the interpreter program (but not in a DYNAMIC program segment) with

```
call procname(a, b, ...; A, B, ...)
```

You can substitute any expression for the value parameters **a, b, ...;** and named variables or arrays, but not subscripted array elements, for the **VAR** parameters **A, B, ...** An error is returned when arguments and/or argument types in procedure definition and call do not match. The procedure **xform** in our example could be called with

```
call xform(r1, theta1; x1, 1)
```

or

```
call xform(RADIUS - 2.12, 4 * PI/180; xx, y3)
```

Procedure arguments can be REAL, **INTEGER**, or **COMPLEX** (Sec. 2-13). Procedures must not contain **label**s. Desire "protects" dummy arguments in procedure definitions just like the function arguments in Sec. 2-11.

Procedure calls can be nested; but a procedure must not call itself directly or indirectly. **PROCEDURE** definitions can contain procedure and function *calls*, but *no declarations*. Recursive **PROCEDURE** definitions, like recursive function definitions, are illegal.

2-13. INTEGER *and* COMPLEX *Variables*

Interpreter programs admit named variables and arrays declared as **INTEGER** and **COMPLEX** before they are used, e.g.

```
INTEGER a, b, c, dd[2, 3]
COMPLEX q[100], z, w
```

Mixed expressions are legal, and data conversion is automatic. Real expressions assigned to **INTEGER** variables are rounded to fixed-point format. **INTEGER** function arguments are converted to real numbers. Desire rarely uses **INTEGER**s. The truncation function **trnc(x)** and the rounding function **round(x)** produce integer-valued real numbers, ***not*** **INTEGERs**.

The first* COMPLEX *declaration automatically defines the imaginary number

```
j = sqrt(-1)
```

Thus, as soon as any named **COMPLEX** quantity is declared, **COMPLEX** numbers can be assigned values **a + j * b**, where **a** and **b** are REAL expressions. You can use the symbol **j** for other purposes (say as an array index) if (and only if) your program contains no **COMPLEX** declarations. Interpreter expressions, including those in **data/read** and **input** operations (Sec. 2-8), and also user-defined functions and procedures, admit and recognize **COMPLEX** quantities both in the form **a + j * b** and in the form **[a, b],** where **a** and **b** are REAL expressions.

EXAMPLE:

```
INTEGER  k, knum, NUMERUS
 COMPLEX z, cc, W
---------
k = 15  |  knum = k + 4  |   NUMERUS = - 3
realv = 21.2 * k - NUMERUS
W = 53.121 - j * G   |   cc = 1000
cc = cc + 1 + j * 2
z = a * pp1 + W - realv + [2, realv-1]
```

Note that **cc = 1000** in the example is equivalent to **cc = [1000, 0].**

input, data/read, and **write** statements, and appropriate user-defined interpreter functions and procedures (Secs. $$) all admit and recognize **INTEGER** and **COMPLEX** quantities. Data types in function or procedure definitions and calls must match. DYNAMIC program segments, and also interpreted **Vector**, **MATRIX**, and **DOT** operatios (Sec. $$), admit only real quantities.

Library functions other than **exp(x)**, and also exponentiation (**x^y**), **input**, and **read** operations do ***not*** work for **COMPLEX x** but return an error message.

The real functions

Re(z) Im(z) Cabs(z) Arg(z)

(***real part***, ***imaginary part***, ***absolute value***, and ***argument*** of **z**, and the **COMPLEX** functions

Conjug(z) and **exp(z)**

(***complex conjugate*** and ***complex exponential***) admit **COMPLEX** arguments. The **COMPLEX** function

expj(x) = exp(j * x) [equivalent to **cos(x) + j * sin(x)]**

is normally useful only for real **x**. For **COMPLEX z, expj(z)** simply returns **expj(RE(z)).**

Complex-number plotting permits complex frequency-response plots and even conformal mapping (Chap. 8, ***Interactive Dynamic-system Simulation***). The interpreted graphics statement

plot *expression1, expression2*

plots a complex-number point **(x,y)** provided that ***expression1*** contains any previously declared **COMPLEX** quantity. **round(*expression2*)** is a color code. Assignments to the system variable **scale** scale **x** and **y** together. A **display A** statement (Table 2-2) must precede plotting and produces a Graph Window; **display C n** sets coordinate-net colors.

2-14. Octal and Hexadecimal Integers

Literal ***octal and hexadecimal integers*** are, respectively, designated with prefixes **&** and **&&** and can be used in expressions like

Noctal = &17370 Mhex = &&FC2

Currently such octal and hexadecimal integers can be as large as 65535 (decimal) and may encode negative numbers. Desire rejects unacceptable codes and disregards the leading digit(s) if you enter too many digits.

Conversely,

write %12918 and **write %%12918**

respectively display the decimal integer 12918 in octal and hexadecimal form.

2-15. Interpreted Vector/Matrix Operations

The interpreted experiment protocol program also admits operations on one-dimensional (vector) and two-dimensional (rectangular matrix) arrays. This is discussed in Secs. 5-22 to 5-24.

Table 2-1. Desire Library Functions

(a) Both interpreted experiment-protocol scripts and compiled DYNAMIC program segments both admit the library functions

round(x) **sqrt(x)** **sin(x)** **cos(x)** **tan(x)** **exp(x)**
asin(x) **acos(x)** **atan(x)** **sinh(x)** **cosh(x)** **tanh(x)**
ln(x) (base *e*) **log(x)** (base 10) **sinc(x)** (≡ sin(x)/x)

atan2(y,x) (arctan(*y/x*) between **-PI** and **+PI**)

sigmoid(x) ≡ 1/(1 + exp(-x))

SIGMOID(x) ≡ 0, x^2/(1 + x^2) (**x ≤ 0, > 0)**

and the piecewise-linear functions

abs(x) **lim(x)**) **sat(x)** **SAT(x)** **deadz(x)** **tri(x) (≡ 1 - abs(x))**
swtch(x) **sgn(x)** **deadc(x)** **rect(x)**

defined in Fig. 2-1.

Function arguments can be expressions, which may involve literal numbers and subscripted variables.

(b) In DYNAMIC program segments, **Vector** and **Vectr d/dt** assignments admit library functions of vector expressions. But DYNAMIC-section function arguments for **MATRIX** and **DELTA** (**LEARN)** assignments must be, respectively, simple matrix factors (e.g. **A**, **A * B**), or "primitive sums" in simple parentheses,

(A + *scalar*) or **(A + *simple matrix*)**

not general **MATRIX** expressions (Chaps. 5 and 8). Here **A** a matrix factor in **MATRIX** or **DELTA**. The scalar must be a named scalar (not an expression or literal number). As an example,

MATRIX y = tanh(W * V + b)

is legal whether **b** is a scalar or a matrix.

The library function **recip(x)** works ***only for matrices*** and produces the reciprocal of each matrix element.

(c) Interpreted experiment-protocol programs, but ***not DYNAMIC segments***, admit the additional REAL functions **trnc(x)** (truncation) and also

Re(z) **Im(z)** **Cabs(z)** **Arg(z)**

where **z** can be **COMPLEX**. There are also two **COMPLEX** library functions

Conjug(z) **exp(z)**

The **COMPLEX** function

expj(x) = exp(j * x) [equivalent to **cos(x) + j * sin(x)**]

is of interest only for REAL **x**. For **COMPLEX** arguments **z**, **expj(z)** simply returns **expj(RE(z)).**

(Table 2-1 continued)

(d) The REAL library function **ran()** produces uncorrelated REAL pseudorandom-noise samples uniformly distributed between -1 and 1 in both experiment-protocol scripts and DYNAMIC program segments. Desire also accepts the older notation **ran(0)** but disregards the argument **0**. In new 64-bit Linux versions[1] of Desire the REAL library function **gauss(0)** [***not*** **gauss()**] similarly produces uncorrelated samples of a standardized normal variable (mean 0, variance 1),[2] using the Box-Mueller formula

x = s * sqrt(- 2 * ln(abs(ran()))) * cos(2 * PI * abs(ran()))

The programmed or command-mode statement

SEED n

where **n** is a positive integer, can be used as a command or anywhere in the interpreter program to start **ran()** or **gauss(0)** with a new seed. If **SEED n** is not used the seed defaults to a preset value.

(e) In the interpreted experiment-protocol program, the REAL function **tim(0)** returns the time in seconds. This can be used to time programs or program segments:

```
aa = tim(0)
… …  experiment-control-script lines ……..
bb = tim(0)  |  write bb - aa
```

(f) The function

$$\text{comp(x, minus, plus)} = \begin{cases} \text{plus} & (x > 0) \\ \text{minus} & (x \le 0) \end{cases}$$

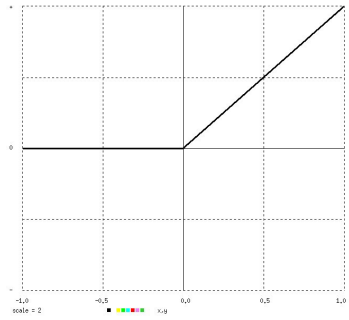
represents a ***comparator (relay)*** or ***function switch***. **x, plus**, and **minus** can be REAL expressions. **minus + (plus - minus) * swtch(x)** would accomplish the same purpose.

(g) The ***track/hold pair*** or ***unit-delay operator*** **trkhld y, y0 = ctrl, x is** still recognized for backward compatibility; ***it is obsolete*** because it cannot be vectorized. Use a difference equation instead:

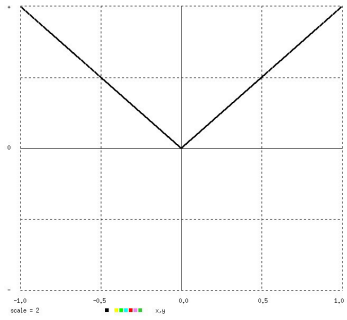
```
step
y = y + swtch(ctrl) * (x - y)
```

.

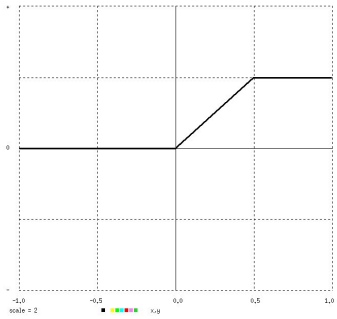
[1] Under Windows you must use the Box-Mueller formula explicitly.
[2] Note the **gauss()** will return an error.



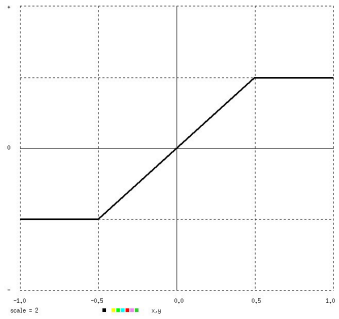
$$\text{lim}(x) = \begin{cases} 0 & (x \le 0) \\ x & (x > 0) \end{cases}$$



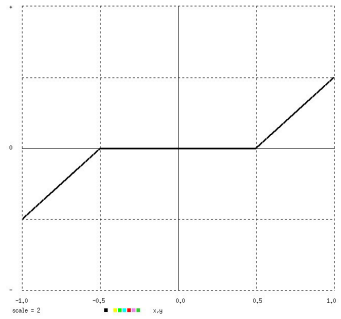
$$\text{abs}(x) = |a|$$



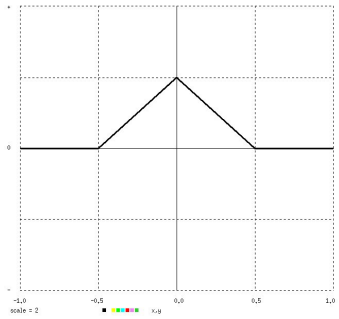
$$\text{SAT}(x) = \begin{cases} 0 & (x \le 0) \\ 1 & (x > 0) \end{cases}$$



$$\text{sat}(x) = \begin{cases} -1 & (x \le -1) \\ x & (-1 \le x \le 1) \\ 1 & (x > 1) \end{cases}$$

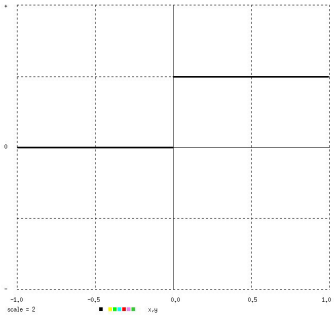


$$\text{deadz}(x) = \begin{cases} x + 1 & (x \le -1) \\ 0 & (-1 < x \le 1) \\ x - 1 & (x > 1) \end{cases}$$

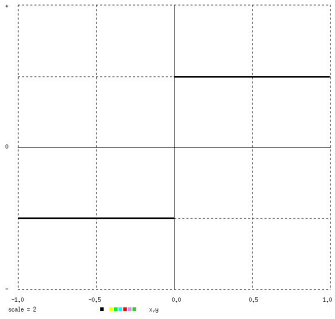


$$\text{lim(tri(x))} = \begin{cases} 0 & (|x| > 1) \\ 1 - |x| & (|x| < 1) \end{cases}$$

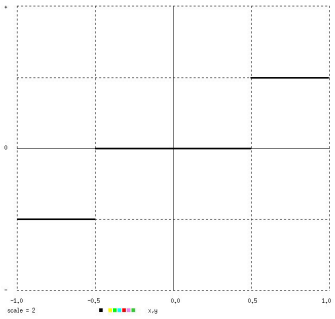
Figure 2-1a. Limiter functions.



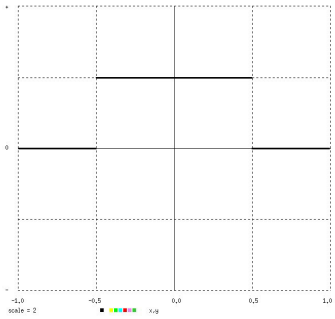
$$\text{swtch}(x) = \begin{cases} 0 & (x \le 0) \\ 1 & (x > 0) \end{cases}$$



$$\text{sgn}(x) = \begin{cases} -1 & (x < 0) \\ 0 & (x = 0) \\ 1 & (x > 0) \end{cases}$$



$$\text{deadc}(x) = \begin{cases} -1 & (x \le -1) \\ 0 & (-1 < x \le 1) \\ 1 & (x > 1) \end{cases}$$



$$\text{rect}(x) = \begin{cases} 0 & (|x| > 1) \\ 1 & (|x| < 1) \end{cases}$$

Figure 2-1b. Switching functions

Chapter 3. Experiment Protocol, DYNAMIC Program Segments, and Simulation Runs

DESIRE SIMULATION PROGRAMS

3-1. Interpreted Experiment-protocol Scripts and Compiled Simulation Runs. **drun** *Statements*

(a) Every Desire program begins with an interpreted ***experiment-protocol script***. This is an interpreter program (like Basic) that sets model and experiment parameters and state-variable initial values, fills arrays, and selects display colors and scales as needed.

A programmed or command-mode interpreter statement

drun or **drun** ***label***

calls a ***simulation run***. The simulation-run code is a procedure that normally "exercises" a dynamic-system model defined by differential equations and/or difference equations in a ***DYNAMIC program segment***. **drun** causes automatic, very fast compilation of the DYNAMIC-segment code. The compiled code, combined with library routines for integration, function generation, and display, then executes immediately through successive time steps or trial steps to produce ***time histories*** of model variables and arrays. You can display and/or list such results in vasrious ways (Chap. 6).

A simulation run normally ***terminates*** after a specified number of time steps (see below) or when a programmed termination condition is satisfied (**term** statement). You can ***continue a terminated simulation run*** with a programmed or command-mode **drun** statement. By default, all variables then restart with the values they had at run termination; you can also change variables or parameter values before continuing. Time-history graphs or listings continue with correct labels.

(b) To ***pause*** a simulation run under most Linux systems (e.g. Fedora) type **ctrl-c**. Under Windows, you can pause a run only when a Graph Window is open; click twicc in the Graph Window.

To ***continue*** a paused run, type **Enter** in the Command Window. Type **space** followed by **Enter** to ***terminate*** the run.

(c) DYNAMIC program segments are separated from the interpreter program by the **DYNAMIC** statement, which must be the only statement on its line. A Desire program has at most one **DYNAMIC** statement, but it can have more than one DYNAMIC program segment.

(d) In a ***multirun simulation study***, the experiment protocol may reset initial conditions after a simulation run, modify parameters, initial values, and/or arrays, and then call additional simulation runs. New simulation runs can utilize results from earlier runs. Different simulation runs can optionally use ***multiple, labeled DYNAMIC program segments*** to exercise different models, or simply to obtain different types of output.

3-2. Simulation Time and Other Simulation Parameters

The experiment-protocol script sets the ***total simulation-run time*** **TMAX**, the ***integration step*** **DT** (if any), and the desired number **NN** of communication points, and the initial ***display scale*** **scale**. The progress of a simulation run is measured by the ***simulation time*** **t**, which starts with an initial value **t = t0** and increases until **t** = **t0 + TMAX**. The experiment protocol can set **t0** with

t = ***expression***

(***not*** **t0** = ***expression***) before a simulation run. **t0** can be negative as well as positive and defaults to 0 if the program contains differential equations; otherwise **t0** defaults to 1.

Desire produces time-history output (Chap. 6) at (or close to) the sampling points (***communication points***)

t0, t0 + COMINT, **t0 + 2 COMINT, ..., t0 + TMAX**

where **COMINT** is the ***communication interval***

COMINT = TMAX/(NN –1)

Time-history sampling within integration steps makes no sense, so Desire returns an error message when the integration step **DT** exceeds **COMINT**.

With the -step integration rules 5 to 16, the last integration step before each communication point automatically ends at that communication point. To obtain the same desirable result with fixed-step integration rules you can ***select*** **DT** ***and*** **NN** ***and then set*** **TMAX** ***to*** **(NN-1)*DT**.

Time-history listings and displays are automatically labeled with the selected numerical values of **t0** and **TMAX**, and with the current value of **scale**, as needed. To obtain output only at every **MM**th communication point, set the system parameter **MM** to an integer value larger than its default value 1. This lets you use the communication points for sampling e.g. pseudorandom noise at a rate faster than you might want simulation output.

3-3. Integration Rules

(a) If a DYNAMIC program segment contains ***differential equations*** (**d/dt** and/or **Vectr d/dt** statements, Chaps. 4 and 5) you can specify an ***integration rule*** with the programmed or command-mode statement

irule ***n***

Each integration routine increments **t** through successive integration steps **DT** and accumulates state-variable increments obtained from successive passes through the DYNAMIC-segment code (Chap. 4).

The choice of integration rules is an art based on experience rather than a science. A step size **DT** small enough for accurate integration can often be established by decreasing **DT** until the solution no longer changes perceptibly.

(b) Table 3-1 lists the available integration rules. Euler and fixed- and variable-step Runge-Kutta rules admit up to 40,000 first-order ordinary differential equations. Variable-order/ vari-

able-step Adams and Gear rules admit up to 1000 ordinary differential equations. **irule 1** (2nd-order fixed-step Runge-Kutta-Heun) is the default rule; **reload**, **new, NEW,** and **old** (but not **run)** restore **irule 1**. You can select different integration rules for successive simulation runs. The command **irule** displays the number ***n*** of the integration rule currently in force.

NOTE: Adams and Gear rules should not be used in more than one DYNAMIC program segment.

(c) **irule 10** and **irule 14** require a user-furnished **n**-by-**n** ***Jacobian matrix*** for **n** state variables. The Jacobian may have any legal name. ***The* ARRAY *statement declaring the Jacobian, say***

ARRAY J[5, 5]

must precede all other data declarations or assignments in the interpreter program; failure to do this returns an integration-deadlock error message. A subsegment starting with the statement

JACOBIAN

at the end of the DYNAMIC segment must specify all non-zero elements **J[i, k]** of the Jacobian matrix as functions of other problem variables (see also Chap. 4 and Examples **fuelx.lst, gearx.lst, rule15.lst**, and **stiffx.lst**).

3-4. Default Values

(a) Desire automatically defines the constant **PI** = 3.141.... In addition, **reload, new, NEW,** and **old** set the display scale **scale** to its initial default value 1. ***All other variables and array elements are initialized to* 0 *by default***, except that **t0** defaults to 1 if there are no differential equations. In particular, differential-equation state variables whose initial values equal 0 need not be initialized.

Desire's variable-step integration routines automatically force the last integration step in each communication interval to end on one of the communication points specified by Eq. ($$); you are warned if the initial **DT**-value exceeds **COMINT**. Fixed-step integration routines, on the other hand, increase each sampling time by a fraction of **DT** so that it coincides with the end of an integration step. That causes no errors when you plot or list variables versus **t**. But you will want to ***set* DT *to a small integral fraction of* COMINT** to ensure accurate periodic sampling.

If a reserved variable is not specified by the program or by a command then the first **drun** (or **drunr**, Sec. 3-4) of a Desire program sets the ***initial default values*** listed below.

If you should need to access these parameters earlier in the experiment protocol script you must set their values yourself. Note also that ***subsequent* drun*s* *do not reset default values if the program has changed them***. For these reasons, you might want to program such values explicitly.

Here is a list of default values:

1. **scale** defaults to 1.
2. **NN** defaults to 251 or **TMAX/DT + 1**, whichever is smaller. **MM** defaults to 1.

If the first DYNAMIC segment to run explicitly involves *differential equations* **(d/dt** and/or

Vectr d/dt statements), then

3. the integration rule defaults to **irule 1** (2nd-order fixed-step Runge-Kutta-Heun integration).
4. the initial value **t0** of **t** is set to its default value 0 by the first **drun** or **drunr**.
5. ***an error message warns you*** if the total run time **TMAX** is not specified.
6. **DT** defaults to **COMINT/2 = ½TMAX/(NN –1)**.

If the first DYNAMIC program segment to run does *not* involve explicit differential equations (no **d/dt** and/or **Vectr d/dt**, e.g. only difference equations or graph plotting), then

7. the integration rule defaults to **irule 0** (no integration), and
8. the first **drun** or **drunr** sets the default initial value **t0** of **t** to **1**, and the default value of **TMAX** to **NN - 1**. Thus **t** (which labels time-history graphs and listings) conveniently steps through successive ***trial numbers*** **t = 1, 2, ... , NN**.

Note that 5 and 7 hold ***only*** when the DYNAMIC program segment in question is the first DYNAMIC program segment to run. Otherwise you must set **irule, t,** and **TMAX** yourself.

(c) The integration parameters **DTMIN, DTMAX, ERMIN, ERMAX,** and **CHECKN** used with variable-step integration (Table 3-1) are reserved variables; they are normally set by the experiment protocol. If their values have not been set by the program, the first **drun** or **drunr** sets these parameters to harmless but possibly not useful default values.

ERRMAX is just used to read out the error in deadlocked variable-step Runge-Kutta routines (Table 3-1).

3-5. Resetting Initial Conditions: **reset** *and* **drunr**

(a) For DYNAMIC program segments with explicit differential equations, the programmed or command-mode statement **reset** resets all scalar and/or vector differential-equation state variables, and also **t** and **DT**, to the values they had at the start of the last simulation run.

(b) Note that **reset** does ***not*** reset

- state variables associated with ***difference equations***, such as recursive assignments to scalars or arrays, and "sample-hold" state variables generated by **step** statements.
- state variables associated with **Vectr delta** and **DELTA** difference equations (Chap. 8), even when they emulate Euler integration of differential equations.

But you can, if you wish, define and **call** a suitable interpreter **PROCEDURE** that resets scalar variables, including array elements, to their initial values. To reset entire arrays to equal corresponding arrays of initial values after **M** trials, program a **SAMPLE M** statement (Sec. 3-8), or use a separate DYNAMIC program segment.

(c) For extra convenience, the programmed or command-mode statement **drunr** is equivalent to

drun | reset

MORE ADVANCED TECHNIQUES

3-6. Suppressing Runtime Displays and Disk Storage

To save time in simulation studies with repetitive simulation runs (e.g. for optimization, statistics),

1. programmed or command-mode **display 0** and **display 1** statements let you turn the runtime display off and on (Chap. 6).

2. two modified **drun** statements, **drun*** and **drunr*,** act like **drun** and **drunr** but suppress disk storage (**stash** storage, Chap. 7) of simulation-run time histories.

3-7. Programs with Multiple DYNAMIC Segments

(a) A Desire program can compile, and run multiple DYNAMIC program segments in any desired order. One DYNAMIC segment may be unlabelled and will then compile and execute on **drun, drunr, drun*,** or **drunr***. Additional DYNAMIC segments, if any, begin with a user-named **label**

label ***labelname***

Only comments may follow a **label** on the same line. A DYNAMIC segment associated with the **label** ***labelname*** compiles and executes on one of the interpreter statements

drun ***labelname***	**drunr** ***labelname***
drun* ***labelname***	**drunr*** ***labelname***

Segment compilation automatically ends at the end of the program, on the next **label**, or on a **STOP** statement.

Different DYNAMIC segments often simply process, display, or print data before or after a simulation run. They can also represent different models, or different versions of the same model; they run successively, not in parallel.

(b) A scalar state variable **x** may appear in **d/dt x = ...** statements in more than one DYNAMIC segment; in this case, **x** *must* be declared with a **STATE** declaration. **x** need not be declared if **d/dt x** occurs only in one DYNAMIC segment and not in a **SUBMODEL** invocation.

(c) An unlabelled DYNAMIC segment can run repeatedly without recompilation until a labelled segment runs; but labelled DYNAMIC segments are compiled each time they run. For this reason, you cannot "continue" simulation runs of *labelled* DYNAMIC subsegments with a **drun** statement.

In principle, a Desire program could have only labelled DYNAMIC segments. In that case it is legal to omit the **DYNAMIC** statement.

Table 3-1. Desire Integration Rules

(a) EULER AND RUNGE-KUTTA RULES
(up to 40,000 state variables)

k1 = G(x, t) * DT

RULE 1 ***(fixed-step 2nd-order R-K Heun rule*** *- this is the* ***default rule)***

k2 = G(x + k1, t + DT) * DT
x = x + (k1 + k2)/2

RULE 2 ***(fixed-step explicit Euler rule, 1st order)***
Users may change **DT** in the course of a simulation run.

x = x + G(x, t) * DT = x + k1

RULE 3 ***(fixed-step 4th-order Runge-Kutta rule)***
Users may change **DT** in the course of a simulation run.

k2 = G(x + k1/2, t + DT/2) * DT **k4 = G(x + k3, t + DT) * DT**
k3 = G(x + k2/2, t + DT/2) * DT
x = x + (k1 + 2 * k2 + 2 * k3 + k4)/6

Variable-step Runge-Kutta rules compare two Runge-Kutta formulas of different order. The step size doubles when the absolute difference is less than **ERMIN**, until **DT** reaches **DTMAX**. If the system variable **CHECKN** is a positive integer *n*, then the step size **DT** is halved if the ***absolute*** difference of the two expressions for the *n*th state variable exceeds **ERMAX**. If **CHECKN** = 0, then **DT** is halved when the ***relative*** difference exceeds **ERMAX** for ***any*** state variable. A variable-step deadlock error results if **DT** attempts to go below **DTMIN**; the deadlocked absolute difference can then be read in **ERRMAX**.

RULE 4 ***(variable-step Runge-Kutta 4/2)*** compares the 4th-order Runge-Kutta result with

x = x + k2

RULE 5 (***2nd-order R-K-Heun***, like **RULE 1** but users may change **DT** during a run)

RULE 6 (spare, not currently implemented)

RULE 7 ***(variable-step Runge-Kutta 2/1)*** compares

k2 = G(x + k1, t + DT)
x = x + (k1 + k2)/2 **with** **x = x + k1**

RULE 8 ***(variable-step Runge-Kutta-Niesse)*** compares

k2 = G(x + k1/2, t + DT/2) * DT
k3 = G(x - k1 + 2 * k2, t + DT) * DT
x = x + (k1 + 4 * k2 + k3)/6 **with x = x + (k1 + k3)/2**

(b) ADAMS-TYPE VARIABLE-ORDER/VARABLE-STEP RULES

(up to 1000 state variables)

RULE 9	functional iteration
RULE 10	chord/user-furnished Jacobian
RULE 11	chord/differenced Jacobian
RULE 12	chord/diagonal Jacobian approximation

(c) GEAR-TYPE VARIABLE-ORDER/VARABLE-STEP RULES

(for stiff systems, up to 1000 state variables)

RULE 13	functional iteration
RULE 14	chord/user-furnished Jacobian
RULE 15	chord/differenced Jacobian
RULE 16	chord/diagonal Jacobian approximation

For Integration Rules 9 to 16 the experiment protocol must specify a maximum ***relative*** error **ERMAX** for all state variables; values set to 0 are automatically replaced by 1 (examples **orbitx.src**, **to22x.src, rule15.src**). The initial value of **DT** must be low enough to prevent integration-step lockup.

For Integration Rules 10 and 14 the experiment-protocol script must declare an **n**-by-**n** ***Jacobian matrix*** **J[I, k]** whose elements are defined by assignments in a compiled **JACOBIAN** ***program segment*** (examples **orbitx.src**, **fuel.src**, **fuel3.src**).

Original References for the Adams and Gear Rules:

Gear, C.W.: DIFSUB, Algorithm 407, *Comm. ACM*, 14, No. 3, 3/71.
Hindmarsh, A.C.: LSODE and LSODI, *ACM/SIGNUM Newsletter*, 15, No. 4, 1980.

Chapter 4. DYNAMIC Program Segments

DYNAMIC-SEGMENT CODE

4-1. DYNAMIC Program Segments

Compiled DYNAMIC program segments define model behavior and signal processing during simulation runs. A DYNAMIC program segment is really the body of a program loop: the DYNAMIC-segment code repeats in successive time steps to generate time histories of model variables.

DYNAMIC program segments handle only REAL variables and REAL arrays. Array operations (Chap. 5) involve only one-dimensional arrays ("vectors") and two-dimensional arrays (rectangular matrices), but Desire also admits subscripted variables with more than two dimensions. Subscripted-variable indices can be REAL expressions, which will be truncated. Subscripted-variable indices used in a DYNAMIC program segment are computed ***at compile time*** and thus will not change during a simulation run.

4-2. Defined Variables and Scalar Differential Equations

The most important scalar DYNAMIC-segment statements are

1. ***scalar assignments to defined variables***

   y = ***scalar expression*** (1)

2. ***scalar differential equations*** in first-order state variable form

   d/dt x = ***scalar expression*** (2)

x and **y** are REAL variables. Importantly, one can also program vector differential equations (Chap. 5).

Each ***scalar expression*** can freely use REAL literals, subscripted variables, parentheses, and the library functions listed in Table 2-1, e.g.

```
2.44 * cos(t - a) + (p - b * x) * (sin(z) + AA3[3,  4])
```

New versions of Desire admit subscripted state variables declared in experiment-protocol script **STATE** ***declarations (Chap. 5) as well as unsuscripted state variables***. If the interpreter program does not assign a state variable **x** an ***initial value*** before calling **drun** or **drunr** then **x** starts at the default initial value 0, so that ***zero initial values need not be programmed***. The programmed or command-mode interpreter statements **reset** and **drunr** ***reset all differential-equation state variables (and also*** **t** ***and*** **DT***) to their initial values at the start of the last simulation run*** (Sec. 3-5).

Unsubscripted scalar differential-equation state variables do not need **STATE** declarations unless their differential equations appear either in multiple DYNAMIC segments or in a submodel invocation (Secs. 3-7 and 4-12).

4-3. DYNAMIC-segment Execution

(a) DYNAMIC-segment execution depends on the integration rule in force (Sec. 3-3). If the segment contains no differential equations (no **d/dt** and/or **Vectr d/dt** statements), then integration defaults to **irule 0**, and the DYNAMIC-segment code simply repeats **NN** times. **t** increments by **COMINT = TMAX/(NN - 1)** at each step (see also Sec. 3-4). If **TMAX = NN - 1** and **t0 = 1** (default values, Sec. 3-4) then **t** counts time steps. Such DYNAMIC-segment updating is used with difference-equation models, or just as an automatic loop for general-purpose signal processing .

Integration rules for solving differential equations normally call the DYNAMIC-segment code ("derivative routine") and increment **t** more than once during each successive ***integration step*** **DT**. Integration steps are normally smaller than (and at most equal to) the communication interval **COMINT**.

(b) DYNAMIC program segments can deal with difference equations as well as with conventional differential-equation models defined by differential equations and “continuous” defined-variable assignments. Desire models can also combine differential equations and difference equations.

In differential-equation problems, assignments preceded by a **step** statement execute only at **t = t0** and at the end of integration steps. Such program sections are needed for switched or hard-limited variables and pseudorandom-noise generators that do not admit conventional numerical integration. Program sections preceded by an **OUT** or **SAMPLE m** statement execute only at periodic sampling times define sampled-data operations (difference equations and/or defined-variable assignments).

Variables fed from **step**, **OUT**, or **SAMPLE m** sections to differential equations and/or “continuous” defined-variable assignments represent sample-hold outputs computed at an earlier time. Such a quantity is a type of difference-equation state variable and requires initialization (see Chapter 2 of Advanced Dynamic-system Simulation).

(c) Initial Values. The experiment-protocol script must establish initial values of all state variables at the starting time t = t0 before the first simulation run, i.e. before drun or drunr is called. State variables include

1. ***differential-equation state variables*** (which conveniently default to 0 if you do not set them). Their initial values are automatically reset by **reset** and **drunr** statements.
2. ***difference-equation or sampled-data state variables***. These include difference-equation state variables in explicit difference equations, but also "sample-hold” state variables fed to a differential-equation program from DYNAMIC-segment sections preceded by **step**, **OUT**, or **SAMPLE** m statements.

Desire warns you if you do not set initial values of difference-equation state variables, except for subscripted variables, which default to 0.

Note that initial values of difference-equation state variables are ***NOT*** automatically reset by reset and drunr statements but must be reset explicitly by the experiment protocol. Initial values of all subscripted variables default to 0.

(d) Order of Statements in DYNAMIC program segments. Defined-variable assignments must be in the correct procedural order to compute current values of defined variables from current values of **t** and current values of the state variables. Desire does not sort defined-variable assignments. The program returns an error when it finds an undefined unsubscripted scalar; there is no such warning for subscripted and arrays.

Desire makes an extra pass through a DYNAMIC program segment at **t = t0**, so that all defined variables always have correct current values before the actual simulation run starts with **t = t0**. For this reason, differential-equation statements like **d/dt x = y**, **Vectr d/dt v = u** always read the correct values of the defined variables y or v initialized or computed in the preceding pass, ***no matter where the differential equations are placed in relation to the "continuous" defined-variable assignments.***

4-4. User-defined Functions

DYNAMIC segment-expressions can freely use REAL functions defined with **FUNCTION** declarations in the experiment-protocol script (Sec. 2-11).

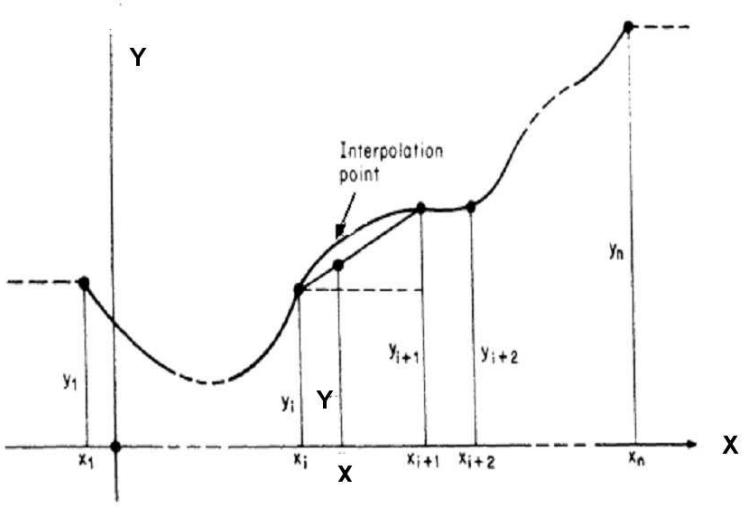


Figure 4-1. Table lookup and linear interpolation.

4-5. Table-lookup/interpolation Function Generators

(a) Functions of One Variable. The DYNAMIC-segment function generator called with

y**= func1(x; F)**

produces a wide variety of functions **y = y(x)** of the input **x** by table-lookup and linear interpolation in a one-dimensional breakpoint-table array **F** set up the experiment-protocol program. The semicolon can be replaced with a comma.

The assignment **y = func1(x; F)** implements

$$y = Y[1] \quad (x < X[1])$$

$$y = Y[i] + \frac{Y[i+1] - Y[i]}{X[i+1] - X[i]} \quad (x - X[i]) \quad (X[i] \le x \le X[i+1])$$

$$y = Y[N] \quad (x > X[N])$$

(Figure 4-1).

A ***function-table array*** **F** for **func1** is declared with

ARRAY X[N] + Y[N] = F

(Sec. 5-2), where **X[1]**, …, **X[2] …, X[N]** are ***breakpoint abscissas***, and **Y[1]**, **Y[2], …, Y[N]** are ***breakpoint ordinates*** for the desired function. Function-table values can be read from a file or **data** list, but the program can also compute them.

A single function array **F** can serve to generate the same function for different inputs, e.g.

y = func1(x; F) | u = func1(p; F)

(b) Functions of Two Variables. The assignment

f = func2(x, y; F, X, Y)

computes the output **f** by bilinear interpolation in a two-dimensional function table of **NM** values

F[i, k] = f([i],[k]) (i = 1, 2, . . . , N; k= 1, 2, . . . , M)

The semicolon can be replaced with a comma. The three arrays **F**, **X**, **Y** must be declared in an **ARRAY** declaration like

ARRAY X[N], Y[M], F[N, M], . . .

before they are used. Desire returns an error if the array dimensions do not match.

A single set of arrays **X**, **Y**, **F** can serve to generate the same function for different inputs **x , y**.

Examples: **newfunc1.src, newfunc2.src**. Fuzzy logic easily produces multilinear or piecewise linear interpolation in function tables (rule tables) of *any* dimension (neural-network text), but the **func2** operation is much faster (Korn, G.A., Simplified Function Generators based on Fuzzy-logic Interpolation, *SIMPRA*, 2000).

func1(x; F) and **func2(u, v; G, X, Y)** can be used in any scalar DYNAMIC-segment expression, and the same breakpoint and function-value arrays can serve different arguments **u1**, **u2, …** and **v1, v2, …**.

The new functions can appear in any DYNAMIC-segment expression, ***including vector expressions***; **x** and/or **Y** can be vectors (Chap.6). They are more powerful and faster than the obsolete Desire/2000 operations

func y = F(x and **func2 f = F(x, y, X, Y**

which refer to similar arrays. These older operations still work in new versions of Desire (Examples **futest1.src**, **fuzztst1,src**, **fuzztst2.src**), but **func1(x; F)** ***and*** **func2(x, y; F, X, Y)** ***do not work in the obsolete Desire/2000 program.***

(c) Vector Functions. Since the same table-lookup function **func1** can be applied to all components of a vector, **func1** is ***vectorizable***, i.e. **x** and **func1(x; F)** can be vectors (all of equal dimension **n**) in any vector assignment or vector differential equation. The same is true of **x**, **y**, and **func2(x, y; G, X, Y).**

4-6. Order of Expression Assignments

Desire is a procedural language. Variables or parameters must be defined before they are used in a right-hand-side expression. Variables used in a DYNAMIC segment may be defined earlier in the DYNAMIC segment, or in the interpreter program before the DYNAMIC segment is called. Both Desire compiler and interpreter return an error if they encounters an undefined variable (see also Sec. 4-7).

Differential equations can appear in any desired order, since their state variables are defined as initial values (even if they just default to 0). Defined-value assignments needed as intermediate results in differential equations must precede these differential equations. Other defined-value assignments may be needed only for output and can be placed at the end of the DYNAMIC segment, possibly after an **OUT** statement (see also Sec. 4-8).

Arguments of input/output requests like **type, dispt, stash, ...** must also be defined before the request is called (Chap. 6).

MORE ADVANCED TECHNIQUES

Note: Some of the programming techniques outlined in Secs. 4-7 and 4-8 are new (2007) and require a detailed discussion of the theory with examples of actual programs. This is the subject of Chapter 2 of Ref. 3.

4-7. Sampled-data Operations and Difference Equations. DYNAMIC Program Segments Containing both Differential Equations and Difference Equations.

(a) Output Sampling. ***Input/output requests*** (**dispt, dispxy, DISPXY, type, stash, recover**, Chap. 6) normally execute at the communication points. When you want a larger output-sampling interval than the communication interval needed for sampled-data computations, you can set the system variable **MM** to produce output only at every **MM**th communication point (Chap. 2 of Ref. 3).

(b) Sampled-data Operations. DYNAMIC program segment may contain differential equations (representing, say, a conventional dynamic system) and/or difference equations (simulating, say, a digital controller or an averaging operation). In this case, ***sampled-data or difference-equation variables. (which include all pseudorandom noise inputs) must be updated only at communication points.*** For this reason, all assignments to sampled-data variables ***must*** follow an **OUT** or **SAMPLE n** statement (see below) ***at the end*** of all assignments to "continuous" variables in a DYNAMIC program segment. Then

1. DYNAMIC-segment code following an **OUT** statement executes ***only at the communication points.***

2. with the exception of input/output requests (**dispt, type**, etc.) DYNAMIC-segment code following a **SAMPLE m** statement executes only ***once per*** **m** ***communication points***. **m** can be any positive REAL expression, which will be truncated to an integer. Desire admits only one **SAMPLE m** statement per DYNAMIC program segment.

This ensures that all sampled-data code executes at uniformly spaced sampling times, while the differential equations still integrate correctly between communication points. This is true even with variable-step integration rules, for they are designed so that integration steps always "hit" the next communication point properly. Note that pseudorandom noise samples are sampled-data variables.

(c) Recursive Assignments and Difference Equations (see also Sec. 2-2). "Implicit" scalar recursive assignments like

y = sin(x) | x = a * y + b

(***algebraic loops***) return an "undefined variable" error unless you have previously defined **x**, say by assigning it an initial value. In that case the recursion defines a difference equation. Such difference equations alternately update **x** and **y**, and ***you must watch carefully which one comes first.***

In a DYNAMIC segment without differential equations, a ***recursive assignment*** like

x = cos(x) + a (3)

is not a simple defined-variable assignment but a state equation (***difference equation***), which expresses the ***new*** value of a state variable **x** as a function of an ***earlier*** **x**-value on the right. **x** then needs an initial value; if no initial value has been assigned in the experiment protocol, the DYNAMIC-segment assignment (3) ***defines*** **x** and gives it the default initial value 0. Such recursions have many very useful applications (track-hold operations, hysteresis, Schmitt trigger – see Chap. 2 of Ref. 3).

(d) Sampled-data Initialization. Just like differential-equation state variables, difference-equation state variables must be given initial values before the DYNAMIC program segment is executed for the first time. Nonzero initial values of all state variables are assigned by the experiment-control script.

> ***NOTE.*** For convenience, the initial values of array components, and also state variables defined by simple recursive difference equations like **x = cos(x) + a**, default to 0. Desire warns you with an error message if other difference-equation state variables are not initialized.

Note that **reset** and **drunr** statements reset only differential-equation state variables; difference-equation state variables must be reset explicitly.

4-8. Using Limiter, Switching, and Random-noise Functions with the Desire **step** *Operator*

For correct numerical integration, limiter and switching functions (Fig. 2-1) of "continuous" state variables or defined variables, and also the random-noise generator **ran()**, must follow a **step** ***statement to ensure that they execute only at the end of integration steps and at*** **t = t0**. Correct programming techniques for using the **step** operator, and in particular for integration through discontinuities, are discussed in Chap. 2 of Ref. 3.

CONDITIONAL RUN TERMINATION AND if STATEMENTS

4-9. The Run-termination Operator

A ***run-termination statement***

term ***expression***

terminates the simulation run as soon as the value of ***expression*** exceeds 0. A DYNAMIC segment may contain more than one termination statement; the run terminates when the first termination expression exceeds 0.

4-10. **if** *Statements in DYNAMIC Program Segments*

All code except for input/output requests like **type, dispt,...** which follows a DYNAMIC-segment **if** statement

if ***scalar expression***

executes only when the expression is positive. A DYNAMIC program segment may contain multiple and/or nested **if** statements. Note that such compiled **if** statements are quite different from the interpreted **if ... then ... else ... proceed** statements of Sec. 2-5.

SUBMODELS

4-11. Submodel Definition, Invocation, and Nesting (Chap. 3, of Ref. 3)

Submodels in DYNAMIC program segments permit invocation of a complete differential-equation system, neural-network component, or data-processing operation in a single statement. More than a convenient shorthand notation, submodels are useful modeling abstractions.

A submodel must be declared with a **SUBMODEL** statement and definition in the interpreter program before it is used in a DYNAMIC program segment, e.g.

```
SUBMODEL normalize(x)
  DOT xnormsq = x * x | x$ = 1/sqrt(xnormsq)
  Vector x=x$ * x
  end
```

Desire listings automatically indent the **SUBMODEL** body, which may contain any legal DYNAMIC-segment code. Once declared, submodels can then be ***invoked*** anywhere in a DYNAMIC program segment, e.g.

invoke normalize(input1)

with appropriate REAL or **ARRAY** variables substituted for each dummy argument. Our invocation would generate compiled in-line code for

DOT xnormsq = Input1 *I nput1 | x$ = 1/sqrt(xnormsq)
Vector input1 = x$ * Input1

SUBMODEL declarations can use REAL scalar and array variables as dummy arguments and may also contain additional variables common to all invocations. Desire returns an error if declaration and invocation arguments do not match. Invocation arguments are called by name and must be named variables or arrays, not literals, subscripted variables, or expressions. But it is legal to set, say

GAIN=alpha * at(q/alpha)

either in the interpreter program (constant **GAIN**) or in the DYNAMIC program segment (variable **GAIN**) before invoking a submodel with the argument **GAIN**.

As always, **ARRAY** variables must be declared before they are invoked. Note that arrays used as ***dummy arguments*** in **SUBMODEL** declarations must be separately declared. But since such dummy ARRAYs are never "filled" with actual values, ***you can save memory by setting all dummy-array dimensions to unity***. Specifically,

ARRAY x[1], nput1[1200], Input2[17]

would create the three vector arrays needed for our simple example. Note that you can use ***different dimensions*** for the different invocations of the dummy array **x**.

Submodel definitions and invocations can contain user-defined functions and may invoke other submodels (***nested*** submodels). Nested and recursive **SUBMODEL** definitions, and recursive submodel invocations, are illegal. Dummy arguments defined in a submodel declaration can no longer be used elsewhere; they are "protected" to prevent side effects.

The keyword **SUBMODEL** replaces the older keyword **MACRO**. For compatibility with older versions of Desire, **MACRO** is still recognized; **keep**, **keep+**, **list**, and **list+** file operations automatically convert program text to the new notation. Desire **SUBMODEL**s are ***not*** text macros, nor do they generate time-consuming compiled procedure calls. **SUBMODEL** statements produce ***compiler procedures***. When the compiler encounters a submodel invocation, the corresponding compiler procedure is called and compiles efficient *inline code* for the submodel. No extra compiler pass is required.

4-12. Submodels with Differential Equations

Submodel definitions can contain differential equations (**d/dt** or **Vectr d/dt** statements) ***provided that all differential-equation-type state variables have been declared in* STATE *declarations in the interpreter program*.** This is necessary even for scalar state variables, not just for state-variable arrays. But the *dummy* state variables do ***not*** need **STATE** declarations.

DYNAMIC SEGMENTS WITH USER-DEFINED JACOBIANS

4-13. **JACOBIAN** *Subsegments*

irule 10 and **irule 14** (see also Sec. 3-3) require a user-furnished *n*-by-*n* ***Jacobian matrix*** for *n* state variables. The Jacobian may have any legal name. *The* **ARRAY** ***statement declaring the Jacobian, say***

ARRAY J[5, 5]

must precede all other data declarations or assignments in the interpreter program; failure to do this returns an integration-deadlock error message. A subsegment starting with the statement

JACOBIAN

at the end of the DYNAMIC segment must specify all non-zero elements **J[i, k]** of the Jacobian matrix as functions of other problem variables, e.g.

J[2, 5] = 3 * x - sin(t)

Note that only nonzero elements **J[i, k]** need be entered (Examples **fuel*x*.src**, **gear*x*.lst**, **rule15.src**, and **stiffx.src**). A missing or duplicated **JACOBIAN** statement returns an error message.

References: 1 Gear, C.W.: DIFSUB, Algorithm 407, *Comm. ACM*, 14, No. 3, 3/71; and

2. Hindmarsh, A.C.: LSODE and LSODI, *ACM/SIGNUM Newsletter*, 15, No. 4, 1980.

3. Korn, G.A.: *Advanced Dynamic-system Simulation: Model-replication Techniques and Monte Carlo Simulation*, Wiley, New York, 2007.

Chapter 5. Vector/Matrix Operations and Fast Fourier Transforms

ARRAY DECLARATIONS

5-1. Desire Arrays

Both experiment-protocol programs and DYNAMIC program segments use ***arrays*** of real (double-precision) numbers. Experiment-protocol programs also admit arrays of **INTEGER**s. Arrays can have up to 10 dimensions. Most Desire programs, though, involve only one-dimensional real arrrays (***vectors***) and two-dimensional real arrays (rectangular ***matrices***). All arrays must be declared in the experiment-protocol script with **ARRAY** statements like

ARRAY x[3], bias[N2], Qmatrix[12,3], q[m+2]

before they are used.[1] An **ARRAY** statement can declare multiple arrays, and there can be multiple **ARRAY** statements. Array dimensions can be entered as positive floating-point expressions which are truncated to integer values. ***All array elements are initially set to 0.***

Once declared, a real array **x** and its elements **x[1], x[2], …** (***subscripted variables***) can be accessed both in the interpreted experiment-protocol program and in compiled DYNAMIC program segments. Examples of one-dimensional arrays (vectors) are arrays of successive time-history or frequency-domain sample values, geometrical vector components, one-dimensional patterns, neuron-layer activations, and also variables in replicated models. There are also one-dimensional **STATE** arrays of subscripted differential-equation state variables (Sec. 5-4). Examples of two-dimensional arrays (rectangular matrices) are vector-transformation matrices, such as state-transition, feedback, and neural-network connection-weight matrices; pattern arrays whose rows are individual pattern vectors (Sec. 5-8); and two-dimensional patterns.

Declaring an array name more than once returns and error message. **clear** erases all array declarations (as well as variable, function, etc. definitions)..

5-2. Subvector Declarations

A modified **ARRAY** declaration partitions a one-dimensional array (vector) **x** into ***subvectors***, say **x1**, **x2**, . . .:

ARRAY x1[n1] + x2[n2] + . . . = x, . . .

This declares vectors **x1, x2, . . .** together with a vector **x** of dimension **n1 + n2 + …**. whose elements overlay the subvectors **x1, x2, . . .** , starting with **x1**. You can then access, say, **x2[3]** also as **x[n1+3]**. Subvectors let you perform a transformation **A * x** of the entire vector **x** as well as individual transformations **like B * x1** of subvectors. This is particularly useful for creating compact neural-network models.

Subvectors can be very useful. They produce simpler and more compact neural-network programs - and save much memory - in many important applications. In particular, subvector declarations make it easier to combine neural-network layers. Subvectors let you

[1] To accommodate user programs prepared with earlier versions, Desire also accepts the keyword **dimension** as equivalent to **ARRAY**.

1. conveniently specify function-breakpoint tables
2. replace large, sparse connection-weight matrices with multiple smaller ones.
3. add a unit-bias input to a network layer, so that output-neuron bias values are conveniently replaced by extra connection weights.
4. combine Monte Carlo samples (e.g. for probability-density estimation).

5-3. **STATE**-*array Declarations*

(a) A **STATE** declaration like

STATE x[14], svar[3], u, q

in the experiment protocol must precede any use of vectors **x**, **svar**, **u**, **q,** … on the left-hand side of vector differential equations (**Vectr d/dt x = …** , Sec. 5-16), or of their array elements **x[i], svar[i], …** as subscripted scalar state variables in DYNAMIC program segments..

(b) Unsubscripted ***scalar state variables do not need* STATE *declarations***, unless their **d/dt** statements appear

1. in more than one DYNAMIC program segment, or
2. in a **SUBMODEL** invocation.

Inexperienced users can, therefore, formulate simple scalar differential-equation problems in a straightforward "natural" mathematical notation, without programming **STATE** declarations.

5-4. Equivalent One- and Two-dimensional Arrays

The experiment-protocol equivalence declarations

ARRAY VV[m, n] = v **STATE VV[m, n] = v**

let you ***access a two-dimensional array or state array and its elements both as a two-dimensional array* VV *and as a one-dimensional array*** (vector) **v** of dimension **nm**. This lets you extend the powerful **Vector**, **Vectr d/dt**, and **Vectr delta** operations (Sec. 5-5) to "vectorize" two-dimensional arrays and permit powerful operations on matrices reformulated as vectors. Applications include matrix differential equations, image processing, fuzzy-controller membership functions and rule tables, and solution of partial differential equations (refer to the *Advanced Dynamic-system Simulation* text),

COMPILED VECTOR OPERATIONS IN DYNAMIC PROGRAM SEGMENTS

5-5. **Vector**, **Vectr d/dt**, *and* **Vectr delta** *Operations*[2]

In DYNAMIC program segments

[2] The **VECTOR**, **MAT d/dt**, and **delta** (**UPDATE**) operations used in early versions of Desire were replaced with the much more powerful (and also faster) **Vector**, **Vectr d/dt**, and **Vectr delta** operations. The old operations still work.

Vector y = f(t; x,z, …,alpha, ...) compiles into **y[i] = f(t;x[i],z[i],…,alpha, ...) (i=1, 2,…, n)**
Vectr d/dt x = f(t; x,y,alpha, …) compiles into **d/dt x[i] = f(t; x[i],y[i], …) (i =1,2, …, n)**
Vectr delta x = f(t; x,y,alpha, …) compiles into **x[i] = x[I] + f(t; x[i],y[i], …) (i =1,2, …, n)**

where **x**, **y**, **z**, … are previously declared **n**-dimensional arrays (vectors), **alpha** is a scalar, and **f(…)** stands for a Desire expression. ***Such Desire vector expressions can contain literal numbers, nested parentheses, and functions.*** For example

Vector y = (1 – v) * (cos(alpha * z * t) + u)
compiles into **y[i] = (1 - v[i]) * (cos(alpha * z[i] * t) + u[i]) (i = 1, 2, ..., n)**

Desire returns an error message when you try to combine nonconformable matrices.

5-6. Matrix-vector Products in Vector, Vectr d/dt, *and* Vectr delta *Assignments. Avoiding Illegal Recursion.*

(a) **n**-dimensional vectors in a **Vector**, **Vectr d/dt**, and **Vectr delta** operation can be a suitably dimensioned matrix-vector product **A * x** , which then contributes vector-component terms

$$v[i] = \sum_{k=1}^{n} A[i, k] * x[k] \qquad (i = 1, 2, \ldots)$$

For matrix-vector products written as **A% * x**, Desire ***transposes*** the matrix **A**; this is useful in important applications (e.g. neural networks). Nonconformable matrices cause error messages.

(b) The vector **x** in a matrix-vector product **A * x** or **A% * x** must be a ***simple vector*** (not an expression). But it is perfectly legal to define **x** with another **Vector** assignment.

(c) No vector* x *used in a matrix-vector product on the right-hand side of a* Vector *or* Vectr delta *assignment must be identical with the assignment target. Thus,

Vector x = A * x or **Vectr delta x = exp(- W * x)**

would be ***illegal recursive assignments*** and return an error message. To implement such recursions correctly, you must, as in most computer languages, declare an extra vector, say **xx**, having the same dimension as **x** and then program, as in

Vector xx = x | Vector x = A * xx

But note that

Vectr d/dt x = A * x

is legal, since integration routines implicitly provides a separate variable for the derivative.

(d) Matrix-vector products involving ***subvector*s** can produce a more subtle type of illegal recursion. ***If* x1 *is a subvector of* x*, then assignments like***

Vector x = A * x1 and **Vector x1 = B * x**

are *illegal recursive assignments*, just like **Vector x = A * x**. But ***they do not produce error messages warning you about the illegal recursion. You must watch this yourself***. In either example, you can declare an extra vector **xx1** with the same dimension as **x1** and program

Vector xx1 = x1 | Vector x = A * xx1

or

Vector xx1 = B * x | Vector x1 = xx1

5-7. Vector Index-shift Operations and Convolutions

(a) A vector term or factor **x**{*scalar expression***}** in a **Vector**, **Vectr d/dt**, or **Vectr delta** expression, as in

Vector v = x{*scalar expression*}

contributes ***index-shifted*** vector elements **x[i + round*(scalar expression)***. Elements **v[i]** of **v** elements that correspond to index values **i** shifted beyond its index range are set to 0. The rounded scalar expression for the index shift is ***a constant expression computed at compile time*** and can contain literals and/or subscripted variables; it will not change in the course of a simulation run.

An index-shifted vector* x *appearing in a* Vector or Vectr delta *expression must not be identical with the assignment target* v *when the index shift is positive. This would cause an illegal recursion and returns an error message, since the system fills vector arrays starting with high index values. There is no such restriction for **Vectr d/dt** operations.

(b) Vector shift operations let you shift vector-element values back and forth along an array or neuron layer (Examples **shift*.src** *and* **predict.src**). In particular, one can shift successive samples of a scalar function **s(t)** of the simulation time **t** out of and/or into of a vector array **x** of dimension **n**.

(c) **Vector** assignments like

Vector v = a[1] * x{K} + a[2] * x{K - 1} + a[3] * x{K - 2} + . . .

efficiently produce one-dimensional ***convolutions***, i.e.

$$v[i] = \sum_{k=0}^{KK} a[k + 1]\, x[i + K - k]$$

Two-dimensional convolutions are possible (Chap. 9 of Ref. $$). The weights **a[k + 1]** are components of a vector **a** that can be manipulated with **Vector** and **Vectr delta** operations.

(d) Index shifting has many useful applications (Ref, $$). In particular, you can implement delay lines and serial/parallel conversions that can be used to load arrays and to display array outputs (Chap. 9 of Ref. $$). A One interesting applications implements the ***method-of-lines solution of partial differential equations*** [$$] and create very compact models of analog and digital filters [$$].

5-8. Pattern-matrix Rows Supply Vectors for Successive Trial Steps

A **N**-by-**n** ***pattern-row matrix* P** stores **N n**-dimensional pattern vectors in successive rows and lets you ***select*** one of the pattern vectors. After you specify the value of the system variable **iRow**

> 0, vector assignments like

Vector u = P# + A * x2 | Vector v = (x - alpha) * P# + bias

substitute ***the vector in the*** **iRow**$^{\mathbf{th}}$ ***row*** of **P** for **P#**. The system variable **iRow** determines the index **i** of the selected row as the remainder **iRow mod N** in the quotient **iRow/N**. Desire returns an error message if **iRow < 1.**

If **t** equals an integer trial number, then the program line

iRow = t

causes the selected row to ***cycle*** through successive rows of **P** in successive trials. If the **t = 0** for your first trial you can use **iRow = t + 1**. Other useful pattern sequences are obtained with

iRow = k * abs(ran())	pseudorandom "scrambling" of successive patterns
iRow = t/m	repeats each row-vector for **m** trials before proceeding to the next row

A DYNAMIC program segment can reassign **iRow** as often as needed to produce different patterns or pattern sequences. You can use as many pattern arrays as you wish. They need not have the same dimensions, although they often have the same number **N** of rows. The same **iRow** assignment can serve two or more pattern matrices, or new **iRow** values may be assigned for the same or different pattern matrices, e.g.

iRow = t | Vector x1 = INPUT#
iRow = t + 2 | Vector x2 = INPUT# | Vector x3 = x1 + x2

If **X** is an **N**-by-**n** pattern-row matrix then **X#** can be used as a vector in an **n**-dimensional vector expression, but that **A#** cannot be index-shifted or used in a vector-matrix product.

5-9. Selection of Maximum Vector Components and Vector Masking

(a) The modified **Vector** assignment

Vector x^ = ***Vector expression***

sets all but the largest component of the vector **x** to 0. The given vector components **x[k]** can be positive, negative, or 0.

(b) **Vector**, **Vectr d/dt**, and **Vectr delta** operations for **n**-dimensional vectors can be ***masked*** with an **n**-dimensional ***mask vector*** **vv**, as in

Vector x = [vv] ***vector expression***
Vectr d/dt x = [vv] ***vector expression***
Vectr delta x = [vv] ***vector expression***

A masked vector expression is set to 0 for all values of the vector index **i** such that **vv[i]** $\neq$ **0.** Mask vectors **vv** must be set by the experiment-protocol program and do not change in the course of a simulation run. Vector masking can suppress unnecessary computations in method-of-lines solution of partial differential equations, and is also used to “prune” neuron layers in neural-network simulations (see also Sec. 8-3 for matrix masking).

DOT PRODUCTS, AVERAGES, VECTOR NORMS, ERROR MEASURES, AND SUMS IN DYNAMIC PROGRAM SEGMENTS

5-12. **DOT** *Products and Averages*

(a) The DYNAMIC-segment statement

DOT alpha = u * v

assigns the ***inner product*** of two vectors **u**, **v** of equal dimension **n** to the scalar **alpha**, i.e.

$$\text{alpha} = \sum_{k=1}^{n} u[k] * v[k]$$

Nonconformable matrices are rejected with an error message. Compiled **DOT** operations are fast - they incur no loop overhead.

---------- <****

NOT CURRENTLY IMPLEMENTED!

(b) In DYNAMIC program segments, the second vector **v** of the inner product can be the suitably conformable matrix product **A * b** or **A% * b** of a rectangular matrix **A or** matrix transpose **A%** and a suitably conformable vector **b**, as in

DOT beta = y * W * x | DOT beta = x * W% * y

(c) **DOT** assignments no longer admit ***sums of inner products*** such as

DOT gamma = u1 * v1 + u2 * A * v2 + u3 * v3 + . . .

(b) Averages. It is easy to comput averages by dividing the result of **DOT** operations by the vector dimension **n.**

The following operation is still experimental and may not always work.

Given two vectors **u**, **v** of equal dimension **n**, the DYNAMIC-segment statement

AVG alpha = u * v

assigns

$$\text{alpha} = (1/n) \sum_{k=1}^{n} u[k] * v[k]$$

Nonconformable matrices are rejected with an error message. Compiled **AVG** operations incur no loop overhead.

5-13. Euclidean Norms

DOT assignments efficiently compute squared vector norms such as error and energy measures for optimization studies. In particular,

```
DOT xnormsq = x * x
```

produces the square **xnormsq** of the ***Euclidean norm*** **||x||** of a vector **x**. The ***Euclidean distance*** between two vectors **x, y** is the norm **||x – y||** of their difference. Thus

```
Vector  error = x - y      |    DOT enormsq = error * error
```

produces

$$\text{enormsqr} = \sum_{k=1}^{n} (x[k] - y[k])^2$$

5-14. Simple Sum and Averages. Taxicab and Hamming Norms

(a) Given a vector **x**, the special **DOT** assignment

```
DOT xsum = x * 1
```

efficiently produces the ***sum*** **xsum = x[1] + x[2] + . . .** ***of all the elements of*** **x**. The notation designates the **DOT** product of **x** and a vector of unity elements, but the multiplications by 1 are not actually performed. Compiled **DOT** operations are fast - they incur no loop overhead.

(b) You can neatly produce sums like

S = exp(x[1]) + exp(x[2]) + . . .

with

```
Vector y = exp(x)   |    DOT S = y * 1
```

(you can replace **y** by **x** to save memory if there is no danger of recursion).

(c) In particular,

```
Vector y = abs(x)     |    DOT anorm = y * 1
```

produces the ***taxicab norm*** (city-block norm) **anorm = |(x[1])| + |(x[2])| + . . .** of a vector **x**. The taxicab norm of a vector difference (***taxicab distance***, as in a city with rectangular blocks) is another useful error measure.

(d) If all elements of a vector **x** equal 0 or 1, then the taxicab norm reduces to the ***Hamming norm***, which simply counts all elements equal to 1. The ***Hamming distance*** **||x - y||** between two such vectors is the count of corresponding element pairs which differ

(e) ***Average****s.* Given an **n-**dimensional vector **x**, the assignment

```
AVG  xAvg = x * 1
```

produces the ***sample average*** **n**

$$\mathbf{xAvg = (1/n)} \sum_{\mathbf{k=1}} \mathbf{x[k]}$$

for statistical computations. Compiled AVG operations incur no loop overhead.

5-15. Vector Normalization

To ***normalize*** each of a set of non-zero vectors **x** to unit Euclidean, taxicab, or Hamming norm, divide each vector by its norm. In particular,

DOT xnormsq = x * x | xxx = 1/sqrt(xnormsq) | Vector X = xxx * x

produces ***Euclidean normalization***. If the original vector **x** is no longer needed, you can replace **X** with **x** to save memory. The normalized vectors lie on a unit-radius hypersphere; they do not constitute a vector space, since there is no null vector.

When you normalize a set of vectors which might include a zero vector (e.g. random vectors), you can prevent division-by-zero errors by adding a very small positive quantity to each **||x||**; the zero vector is, of course, not normalized.

VECTOR/MATRIX DIFFERENTIAL EQUATIONS AND DIFFERENCE EQUATIONS

5-16. Vector Differential Equations

Given a one-dimensional state-variable array (state vector) **x** previously declared with a **STATE** declaration (Sec. 5-4),

Vectr d/dt x = ***vector expression***

implements **n** first-order ordinary differential equations, where **n** is the dimension of **x**.

NOTE: **Vectr d/dt x = A * x** does ***not*** cause illegal recursion (Sec. 5-6), since integration routines implicitly provide internal intermediate variables.

DYNAMIC segments can contain both scalar and vector differential equations. **reset** and **drunr** correctly reset vector as well as scalar differential-equation state variables to their values at the start of the last simulation run. All integration routines work; you can have up to 40,000 state variables with **irule 1** to **8** (Euler and fixed- or variable-step Runge-Kutta integration), and up to 600 with **irule 9** to **16** (variable-order/variable-step Adams and Gear integration).

5-17. Vector Difference Equations

Recursive **Vector** assignments can implement ***vector difference equations***, as in

Vector x = x + A * y + . . . **Vectr delta x = A * y + . . .**

or in

Vector x = alpha * y + beta | Vector y = A * x

(see also Sec. 2-13). This is used e.g. in simulations of sampled-data control systems. Note that (1) the components **x[i]** of the vector **x** are state variables and require initial values and (2) they are ***not*** automatically reset by **reset** or **drunr**. A comprehensive treatment of difference equations

in Desire applications is presented in Ref. $$.

TIME-HISTORY FUNCTION STORAGE IN ARRAYS

***5-18. Function Storage and Recovery with* store *and* get** (Chap. 4 of *Interactive Dynamic-system Simulation*)

(a) The DYNAMIC-segment statements **store** and **get** store and read time-history points into and from a one-dimensional array. e.g. for recovery in subsequent simulation runs, for Fourier transformations, or as neuron-layer input patterns. This capability of manipulating *functions* - not merely function *values* - gives the Desire system exceptional power.

The DYNAMIC-segment statement

store X = x

stores values of a scalar variable **x** read at successive communication points **t** = **t0, t0** + **COMINT**, . . . as elements **X[1]**, **X[2], . . .** of a previously declared one-dimensional array **X**. **COMINT** is the communication interval defined in Sec. 3-2. **x** is passed *by name* (no expressions are admitted).

(b) Similarly, the DYNAMIC-segment statement

get x = X

assigns successive elements of an array **X** to the variable **x** between communication points:

x = X[k] for **(k - 1) COMINT ≤ t - t0 < k COMINT (k = 1, 2, . . .)**

(c) The dimension of the array **X** normally equals the number **NN** of communication points; recall that Desire array indices range between 1 (not 0) and **NN**. In any case, **store** and **get** simply stop their operation when the either the array or the simulation run is done. **reset** or **drunr** cause the array index to restart at **k** = 1. **get** then reproduces the array values from the last simulation run, unless new array values were supplied between runs. If **drun** "continues" a simulation run without **reset**, both **store** and **get** continue properly if the array dimension is large enough.

5-19. Fixed and Variable Time Delays

The DYNAMIC-segment ***time-delay operations***

delay y = DD(x, tau and **tdelay y = DD(x, tau**

produce delayed output

y(t) = x(t - tau)

by storing and recovering samples of the input **x(t)** in a buffer array **DD** dimensioned to 1000 (only) by the experiment-protocol program with

ARRAY DD[1000]

delay samples its input once per communication interval, so that the delay **tau** must be less than 1000 **COMINT**. **tdelay** stores samples at successive **DT** steps. That is preferable with large values of **COMINT** and requires ***fixed-step*** integration and **tau** < 1000 **DT**. Desire delay operators work

correctly through simulation runs "continued" with **drun** without **reset**.

delay and **tdelay** arguments are passed ***by name***: they must be variable names, not expressions. The time delay **tau** must be nonnegative but can be variable. There is no interpolation between samples. One can assign initial values to delayed variables by preloading their delay buffers. Such initial values are not automatically reset by **reset** or **drunr**.

INTERPRETED VECTOR AND MATRIX OPERATIONS, FAST FOURIER TRANSFORMS, AND CONVOLUTIONS

Interpreted vector/matrix and FFT operations in the experiment-protocol program can access all previously declared arrays. Thus FFTs can directly handle one-dimensional pattern vectors as well as frequency-response calculations for time histories obtained by solving differential equations.

5-20. **Vector**, **DOT,** *and* **AVG** *Operations.*

(a) **Vector** ***Assignments.*** Interpreted experiment-protocol programs admit simple **Vector** assignments[3] of the form

Vector v = alfa * x + A * y + . . .

where **v**, **x**, and **y** are vectors, **alfa** is a scalar, and **A** is a rectangular matrix. An error message is returned if the matrices are not conformable, or if a term like **A * v** would cause an illegal recursive assignment (see also Sec. 5-6). Scalars like **alfa** can be literals as well as symbolic variables, but not expressions.

Unlike DYNAMIC program segments (Sec. 5-5), interpreted experiment-protocol scripts do ***NOT*** accept more general vector expressions with vector functions, parentheses, and pattern-matrix rows.

(b) Interpreted **DOT** and **AVG** operations like

DOT beta = a * b **DOT sum = x * 1** **DOT gamma = x * A * b**

AVG xAvg = x * 1 **AVG f - x * x**

work exactly as shown in Secs. 5-12 and 5-14 for DYNAMIC program segments.

5-21. Interpreted **MATRIX** *Assignments*

Interpreted experiment-protocol programs implement a number of matrix operations that are ***not*** admissible in compiled DYNAMIC program segments.

(a) Let **A, B, C, . . .** be previously declared ***square*** REAL two-dimensional arrays all having equal dimensions. The interpreter statement

[3] The older keyword **VECTOR** is still recognized.

MATRIX D = alpha * A * B * C * . . .

implements fast ***multiplication of square matrices***. The optional first term **alpha** is a REAL scalar multiplier, which can be a literal number or a simple variable (but ***not*** an expression). Such assignments are, in particular, useful for implementing similarity transformations.

(b) Once you declare a square matrix **C** with **ARRAY C[n,n],** the interpreted **MATRIX** assignments

MATRIX C = 0 **MATRIX C = 1**

respectively create a square ***null matrix*** (all elements = 0) and a ***unit matrix*** (1s along principal diagonal).

(c) The interpreted **MATRIX** assignment

MATRIX B = $In(A)

where **A** and **B** are previously declared ***square*** matrices of equal dimensions, produces the ***matrix inverse*** **B** of **A,** or an error message if the inverse does not exist.

(d) The interpreted **MATRIX** assignment

MATRIX B = A%

produces the ***transpose*** **B = A%** of any rectangular matrix **A**. **A** and **B** must have been declared and must be suitably conformable

5-22. Fast Fourier Transforms and Convolutions

(a) The programmed or command-mode experiment-protocol statements

FFT F, n, x, y and **FFT I, n, x, y**

operate on two REAL one-dimensional arrays **x** and **y** previously declared and dimensioned to dimensions at least equal to **n**. **n** must be one of the literal numbers 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, or 16384. If you have, say, only 900 data points data points, you can use 1024-point arrays, which will be automatically padded with 0s. If one of the array sizes *exceeds* **n**, then the array is effectively truncated to that size.

The **FFT F** and **FFT I** operations produce, respectively, an in-place ***forward or inverse discrete Fourier transformation*** of the complex array **x + jy**. Note, though, that these FFT operations do ***not*** use a COMPLEX array, but two separate ***real*** arrays **x** and **y**. Table 5-1 lists the exact mathematical relationships.

(b) The programmed or command-mode statement

FFT R, n, x, y, u, v

simultaneously transforms ***two real input arrays*** **x** and **y** into four real output arrays **x, y, u, v**, all previously dimensioned to a size at least equal to **n**, which must again equal a power of 2 up to 16384. The complex arrays **x + ju** and **y + jv** are the discrete Fourier transforms of the two REAL input arrays **x** and **y**, which are overwritten.

(c) The programmed or command-mode statement

FFT C, n, x, y

accepts two REAL arrays **x, y** dimensioned as before. The operation transforms **x** and **y** and then computes the inverse transform **x** of their product. **x** is then the ***cyclical convolution*** of the REAL input arrays **x** and **y**: each element **x[i]** of the array **x** is replaced with

$$\mathbf{xnew[i]} = \sum_{\mathbf{k=1}}^{\mathbf{n}} \mathbf{x[k] * y[i - k]} \quad \text{with } \mathbf{y[i - n] = y[i]} \qquad \mathbf{(i} = 1, 2, \ldots, \mathbf{n)}$$

The array **y** represents the imaginary part of the (real) convolution, so that all **y[i]** must be approximately equal to 0.

Table 5-1. Desire Fast Fourier Transforms

1. **FFT F, NN, x, y** implements the ***discrete Fourier transform***

$$x[i] + j\,y[i] \leftarrow \sum_{k=1}^{NN} (x[k] + j\,y[k])\,\exp(-2\pi j\,ik/NN) \qquad (i = 1, 2, \ldots, NN)$$

FFT I, NN, x, y implements the ***discrete inverse Fourier transform***

$$x[k] + j\,y[k] \leftarrow (1/NN) \sum_{i=1}^{NN} (x[i] + j\,y[i])\,\exp(2\pi j\,ik/NN) \qquad (k = 1, 2, \ldots, NN)$$

2. When the **x[k], y[k]** represent **NN *time-history samples*** taken at the sampling times

t = 0, COMINT, 2 COMINT, … , TMAX with **COMINT = TMAX/(NN - 1)**

then the time-domain period associated with the discrete Fourier transform equals

T = NN * COMINT = NN * TMAX/(NN - 1)

(***not*** **TMAX**). Approximate frequency-domain sample values of the ordinary ***integral*** Fourier transform are represented by **x[i] * T/NN, y[i] * T/NN**. Windowing may be needed and can be implemented in either the time or the frequency domain.

3. If the **x[i], y[i]** represent **NN *frequency-domain samples*** taken at the sample frequencies

f = 0, COMINT, 2 COMINT, … , TMAX with **COMINT = TMAX/(NN - 1)**

then

t	represents	**f** (frequency)
COMINT	represents	**1/T** (frequency-domain sampling interval)
TMAX	represents	**(NN - 1)/T**
NN * TMAX/(NN-1)	represents	**NN/T** (frequency-domain “period”)

Chapter 6. Time-history Output, Storage, and Recovery. SHOW Displays

TIME-HISTORY OUTPUT IN DYNAMIC PROGRAM SEGMENTS

6-1. Time-history Output

(a) The DYNAMIC-segment output request

type x1, x2,...

lists **t** and up to 5 variables on the console display during a simulation run. Under Windows, you can ***stop and resume*** such listing by stopping/resuming the simulation run with the **Return** key. Under UNIX, you can stop the run by typing **ctrl-c** and resume it with **drun**.

Time-history graphs between **t** = **t0** and **t** = **t0** + **TMAX** are produced by the DYNAMIC-segment display requests

dispt x1, x2,...	(up to 12 variables vs. **t**)
dispxy x, y1, y2,...	(up to 11 variables vs. the first)
DISPXY x1, y1, x2, y2,...	(up to 6 pairs of xy plots)

Each DYNAMIC program segment may have only one **dispt**, **dispxy**, **DISPXY**, *or* **type** list. The variables in such a list must be defined earlier in the program. Output variables can be subscripted variables (this was illegal in older versions of DESIRE).

(b) Simulation-run time-history output is produced at the **NN** ***communication points*** at or near

t = t0, t0 + COMINT, t0 + 2 COMINT, ... , t0 + TMAX

where the ***communication interval*** **COMINT** equals **TMAX/(NN-1)** or **DT**, whichever is larger.

If **NN** and **DT** are not specified, **NN** automatically defaults to 251, and **DT** to **0.5 TMAX/(NN-1)**. But at the time of the first simulation run, **NN** reduces to **TMAX/DT + 1** if it would otherwise exceed that value.

To save evaluating complicated expressions needed only for output (and not as intermediate values for integration), you can use the DYNAMIC-segment statement

OUT

Then all subsequent defined-variable assignments, and also **term** and **if** statements, will execute only at the communication points.

6-2. Display Coordinate Axes and Colors

(a) **drun** or **drunr** automatically creates coordinate axes and labels for time-history graphs. For **dispt** (plots vs. **t**), the time scale (abscissa) is automatically labeled and scaled between the time-variable values **t0** (which defaults to 0) and **t0 + TMAX**.

Color-key tablets at the bottom of the graph relate the colors of up to 8 curves to the list of variables in the graph label. The command-mode or programmed experiment-protocol statement

display N *n* or **display N*n***

sets the color of the first curve can be set to Color ***n***. To get thicker curves, use **display R**, or increase the number of display points **NN**.

The command-mode or programmed statement

display C m or more conveniently **display Cm**

sets the coordinate-line colors to Color **m**.

(b) Under ***Linux or Unix***, **display C17** produces black coordinate lines and a white background. **display N15** causes all curves to be white, and **display N16** makes all curves black. **display C16** removes the coordinate net (e.g. for 3-dimensional plots).

Under ***Windows***, **display Nn** with **n** < 0 produces a white background. **display N18** and **display N -18** cause all curves to have the same color. Under Windows, interpreter plots always have a black background.

To remove the coordinate lines, use **display C16** (black) for ***n*** > 0 (black background) or **display C15** (white) for ***n*** < 0 (white background).

6-3. Display Scaling and Stripchart-type Displays

The system variable **scale** (defaults to 1) equals the half-scale (0 to maximum excursion) for the ordinate on all graph displays, and ***also for the abscissa*** on xy displays obtained with **dispxy** and **DISPXY** statements in DYNAMIC program segments, or with **plot** statements in experiment protocol scripts. We frequently ***offset*** different curves vertically to separate them in the manner of stripchart records. As an example, one might display

P = 0.5 * (p + scale) **Q = 0.5 * (q - scale)**

instead of **p(t)** and **q(t)**.

The system does not return an error on display-scale overloads but simply clips the display. The DESIRE text has an example showing ***automatic graph scaling*** (program example **scalef.lst**).

6-4. Display Control

Displays are controlled with the following programmed or command-mode experiment-protocol program statements:

display 0	turns the simulation-run display OFF
display 1	turns display ON (default)
display 2	don't erase the display between runs(- combines graphs from multiple runs and/or multiple DYNAMIC program

	segments); do not use in command mode
display C*n*	sets color ***n*** for coordinate net
display N*n*	sets color ***n*** for first curve
display F	erases Command/Graphics windows
display Q	graph using small dots (default)
display R	graph using large dots (takes more time)
display Y	graph using extra- large dots (takes more time)
display W *x,y*	moves graph-window origin to (***xiy***)
display A	produces *xy*-axes for interpreter plots
display B	produces *xt*-axes for interpreter plots

6-5. Printing and File of Desire Programs and Results.

(a) Hard-copy Text and File Storage. Under Windows and Linux or Unix, you can print user files with the editor-window's print-menu commands.

For ***direct file or device output*** of simulation time histories, open a file or device (printer, communication line) with **connect as output *n*** before the simulation run. Then the modified **type** statement

type #*n*, x1, x2,...

lists **t** and up to 5 variables via Channel ***n***, much as with a **write** statement (Sec. 2-3b). The modified type statement **type ##*n*, x1, x2, ...** suppresses the file header to produce a ***computer-readable*** text file. The channel can accept data from more than one simulation run and must be **disconnect**ed when it is no longer needed.

(b) Saving and Printing Desire Graphics. To capture Desire graphics in the Linux or Windows clipboard for file storage or printing use a Linux snapshot tool or the Windows **print-screen** key. You can capture selected windows or the entire desktop, as shown in your operatng-system manual.

FAST BINARY-FILE TIME-HISTORY STORAGE AND RECOVERY

6-6. Using **stash** *and* **recover** *in Simulation-run Time Histories*

DYNAMIC-segment **store** and **get** statements can save and recover simulation-run time histories for use in other simulation runs or other DYNAMIC program segments (Chap. 3). Here we discuss more permanent time-history storage in binary files (**.tim** files), which can be written and read much more quickly than ASCII time-history listings written with **type** requests.

A DYNAMIC-segment statement

stash x, y, . . .

causes disk storage of the time histories of **t** and up to 20 previously defined variables **x, y, ...** at the **NN** communication points **t = t0, t = t0 + COMINT, . . ., t0 + TMAX** in the binary *time-history*

file ***PIC*****.TIM** in the same directory as the source file. Here ***PIC*** is the current program identification code (Sec. 1-13). Disk storage is ***suppressed*** if you call the simulation run with **drun*** or **drunr*** instead of **drun** or **drunr**.

Normally ***PIC*****.TIM** is ***overwritten*** when the run is repeated. But the programmed or command-mode statement **note** adds a version number to ***PIC*** and thus changes its name, thus preserving the current time histories. **note**, moreover, creates an ASCII source-program file ***PIC*****.rcv** (***recovery file***). This preserves the **stash** list and the **t0**, **TMAX**, and **NN** values associated with the saved time histories in the form of a ***recovery-program stub***,

```
connect 'PIC.TIM' as input - 9
t  =  . . . | TMAX = . . .
NN =  . . .
DYNAMIC
recover xx, . . .
/--
```

The negative channel number marks a binary file. You can load ***PIC*****.RCV** with

```
reload 'PIC.RCV'
```

and then edit it to add lines which display or print time-history data; or to use stored time histories as inputs in a new simulation program. For example, simply adding

dispt y or **type y**

at the end of the above recovery program produces a display or listing of the stored time history **y(t)**.

Each DYNAMIC program segment can have at most one **stash** list and one **recover** list, but different DYNAMIC program segments can have different **stash** lists. In that case, each **.TIM** file must be saved with a new **note** statement.

EXAMPLE: Try using **note** after running **airpln.lst** on the distribution disk.

INTERPRETER GRAPHICS AND COMPLEX-NUMBER PLOTTING

6-7. Simple Graph Plotting and Interpreter Graphics (see also Chap. 7 of *Interactive Dynamic-system Simulation*)

(a) DESIRE can be used for graph plotting. Graphs, even in pure interpreter programs, are normally best prepared with a small compiled DYNAMIC segment, since this automatically produces labeled coordinate axes and is also faster.(**sincos.lst** is an example).

(b) Command-mode or programmed *interpreter-graphics statements* are useful (1) for *drawing or plotting individual points* (possibly adding individual points to an existing graph) and (2) for *plotting loci of complex numbers* **x + j * y**, typically to produce complex frequency-response

plots, root-locus plots, or conformal maps.

The programmed or command-mode statement

plot *expression1, expression2, expression3*

where the expressions are real, plots a point (***expression1, expression2***) in the color ***expression3***.

(c) Similarly,

plot *expression1, expression3*

plots a complex ***expression1*** **x + j * y** as a complex-number point **(x, y).** The scale of the plot is set by the current value of **scale**. **plot** statements must be preceded by **display A** or **display B** to start the DESIRE graph window.

SHOW DISPLAYS OF REAL ARRAY DATA

NOTE: compiled **SHOW, SHOW*** implemented in DESIRE/2000 and OPEN DESIRE for Linux, but ***not*** in OPEN DESIRE for Windows.

6-8. SHOW *Displays*

(a) **SHOW** displays the elements of a one- and two-dimensional REAL array graphically as small rectangles of different colors. A typical application is the display of a two-dimensional image pattern represented by a two-dimensional array. **SHOW** images can be superimposed on graphs.

Given a one- or two-dimensional REAL array **A**, the interpreter or DYNAMIC-segment line

SHOW | SHOW A, *ncolummn*

represents the elements of the array **A** in row-major order by successive small rectangles with color values suitably scaled between 0 and 15.

There will be ***ncolumn*** rectangles per screen line, so that ***ncolumn*** is usually set to equal the number of columns if **A** is a two-dimensional array. If ***ncolumn*** rectangles do not fit one window line, or if ***ncolumn*** is omitted, the rectangles wrap to the next line.

(b) One ***initial* SHOW *statement without arguments*** is needed to start the graph window (if it is not already started by a graph) and to position the first **SHOW** array near the top left-hand corner of the window. Two or more successive **SHOW** displays with arguments will then be spaced downward in the graph window while they fit, until another **SHOW** without arguments restarts the displays at the top of the window.

SHOW statements can be used both in experiment-protocol programs and in DYNAMIC segments, where they display once per communication point. The *Neural-networks* text contains a number of examples illustrating applications of **SHOW** displays, and the proper scaling of

SHOW colors.

NOTE: PRINTING PROGRAMS AND LISTINGS ON OLD UNIX SYSTEMS

If an old UNIX or LINUX setup makes it inconvenient to call printer jobs directly, simply **list, list+, write,** or **type** to a temporary text file and print this file with your usual UNIX print command prefaced by **sh**, e.g.

sh lp ***(options) filename***

Chapter 7. Special Facilities for Interactive Experiments

INTERACTIVE PARAMETER CHANGES AND PROGRAM MODIFICATIONS

7-1. User-written Editing Screens

(a) One often needs to modify a few program lines, say those setting parameter values like

```
1200  NN = 1000
1220  gain=2.5
1222  damping=5
```

repeatedly after successive computer runs. Instead of scrolling through a long program with a full screen editor, it is then convenient to display just specific lines and to edit them with **edit** or **ed** (Sec.7-1b).

(b) The experiment protocol program can include a labeled section like

```
2000        label eee  |  --      parameter-editing screen
2100        ed 1200, 1220-1222
```

so that **go to eee** displays an ***editing screen***.

(c) Note that such user-written editing screens can also quickly control program lines other than parameter assignments, e.g. display requests or **FUNCTION** definitions. An editing screen can, moreover, include comment lines (lines starting with **--**) containing explanations or instructions for users.

All screen editing only modifies the user program currently ***in memory***; this program is then automatically recompiled when you type **run**. If desired, the modified program can then be ***saved to disk*** with **keep** or **keep** '***filespec***'.

7-2. Command-mode Parameter Changes

To try new parameter values temporarily ***without*** changing the program currently in memory you can use parameter assignments in ***command mode***, say

```
gain = 7.33
NN = 2000 | damping = -1.2
```

drun then resumes the simulation with the new parameter values.

This is especially useful for continuing already-started optimization runs with new parameters. To use command-mode parameter changes for a ***new*** simulation run you must reset initial conditions (typically with **reset**) and then call **drun**. This is true because **run** reruns all programmed assignments and therefore overwrites your new command-mode assignments. The best technique is to use **drunr** when you want command-mode parameter changes, for this automatically resets initial conditions ***before*** you change parameters.

EXAMPLE: The commands

```
R=5.0E-05
reset | xdot=2.55 | drunr
```

will, if **xdot** is a differential-equation state variable,

1. set **R = 5.0E-05**
2. reset all state variables to their last initial values
3. set the state variable **xdot** to the new initial value **xdot= 2.55**
3. cause an immediate new simulation run
4. reset the state variables again (**xdot = 2.55**)

Only temporary ***data values*** are affected. The ***program*** currently in memory is unchanged, and the DYNAMIC segment is not recompiled.

Note that every *program* change or line deletion, even in the interpreter program, erases the compiled program (since you might have zapped needed data). But you will probably never notice the fast automatic recompilation.

INTERACTIVE DEBUGGING

7-3. Debugging with Programmed **STOP** *Statements*

Programmed **STOP** statements inserted between program lines act as breakpoints that stop program execution anywhere in the interpreter program. You can then examine variables with command-mode or programmed **write** or **dump** statements (see below) and restart execution by typing **go**.

7-4. **trace** *and* **dump** *Facilities*

Desire's interpreter debugging facilities let you single-step the experiment-protocol program, trace execution of program lines, functions, and procedures, and trace and dump numerical values of variables. Specifically,

1. **trace** ***toggles single-stepping*** of the interpreter program and displays ***executed program lines***, ***function/procedure***/**go to** ***execution***, and ***assignments to variables***. Use **go** to continue after each line. Type **trace** again to restore normal operation.
2. **dump** dumps all variables and arrays currently in the symbol table to the monitor screen. Under Linux, you can pause/resume long listings with **ctrl-c/(*any key*)**. UnderWindows, type any key to pause and resume listings; type **z** to terminate a paused listing. **dump *#n*** dumps to a file or device previously **connect**ed to Channel **n**.

You can write, modify, or dump variables while single-stepping. Programs will not continue correctly on **go** after you add or delete a program line.

NOTE: **old**, **reload**, and **new** (but not **run**) automatically restore **trace** and disable stepwise operation.

MISCELLANEOUS FACILITIES

7-5. The Automatic Notebook File

Desire automatically opens a ***notebook file (journal file, log file)*** **notes.jrn** in your directory. This notebook file ***automatically records***

1. the ***current date and time*** when the program is first loaded, and whenever a programmed or command-mode **time** statement executes ("timestamp").

2. the ***problem identification code*** (PIC, Sec. 1-13) of the current program whenever a programmed or command-mode **PIC, note**, **new**, **old**, **reload**, **save**, **keep**, or **chain** statement executes.

3. ***comments*** (typically about parameter values and results of interactive runs) when you use programmed or command-mode statements like

```
3000    note 't=';t,'mass=';m,'    increased m'
        note 'A4=3.5 causes too much overshoot!'
```

   The syntax of such **note** statements is just like that of **write**.

4. ***selected lines of the current program***, when you use programmed or command-mode statements of the form

```
note # 12,20-60,3
```

   (like **list** statements, Sec. 1-4). Such program lines might specify parameter values, comments, or mathematical relations.

Notebook files conveniently preserve time-stamped records of interactive sessions. To avoid losing information, **notes.jrn** is closed and saved on disk only when Desire exits. You must, therefore, quit Desire before you can read or modify the notebook file. To save disk space, **notes.jrn** is ***overwritten*** when you start a new Desire session. To preserve an old notebook file, you must therefore ***rename*** it, say to **notes.*xxx***. This renaming could, if you wish, be done

automatically by a shell script calling the Desire session.

To write and edit more extensive notes on a group of programs, ***you can also open extra Editor Windows to write special text files***. Last but not least, you can add more extensive comments to your *program files*. In addition to ordinary comments started with --, you can terminate the last DYNAMIC program segment with **STOP**. Then ***all text following* the STOP *line is treated as a comment.***

7-6. The time *Statement*

The programmed or command-mode interpreter statement **time** displays ***the current date and time*** on the console terminal and also enters it in the notebook file as a time stamp.

7-7. Access to Operating-system Commands

Desire executes operating-system commands preceded by **sh.** Such commands let you consult directories, copy, print, or rename files, format diskettes, and run shell scripts. What is more, operating-system commands can be Desire program lines, so that you can execute external programs as part of a Desire simulation study. Such programs can be pre- or postprocessors for simulation data or text, e.g. statistics or graphics programs written in C, Fortran, or any other language. They can communicate with your simulation program by reading and writing ASCII files or pipes.

7-8. The Desire Help Facility

Desire help files are ordinary text files, so that users can add heir own help screens at any time. Help files can be collected in a special folder or folders.

To display any text file you can simply drag its icon into a Desire Editor Window.

Text files located in the **\mydesire** folder (Windows) or in the **/desire** folder (Linux) are also displayed in the Command Window when you type **help** ***filespec*** in the Command Window.Hit the space bar to continue long files.

Chapter 8. Matrix Operations in DYNAMIC Program Segments

COMPILED MATRIX ASSIGNMENTS

8-1. Matrix Expressions: Sums of Product Terms

(a) **MATRIX** assignments in DYNAMIC program segments manipulate entire rectangular matrices, such as state-transition matrices and neural-network connection-weight matrices. Rectangular **n**-by-**m** matrices **W** properly declared in the experiment protocol script can be related to scalars, vectors, and other matrices with DYNAMIC-segment ***matrix assignments***

MATRIX W = A1 * A2 * … + B1 * B2 * … + … (1a)

where **A1, A2, …**, **B1, B2, …**, … are constant or variable **n**-by-**m** matrices. Note carefully that **A1 * A2 * …** ***does NOT represent a matrix product***; the assignment ($$) compiles into **n * m** scalar assignments

W[i, k] = A1[i, k] * A2[i, k] * … + B1[i, k] * B2[i, k] * … + …
(i = 1, 2, …, n; k = 1, 2, … , m) (1b)

Minus signs may replace plus signs, and a leading minus sign is admitted. Nonconformable matrices are automatically rejected with an error message.

Such matrix expressions are less general than the vector expressions in Sec. $$. Still, each of the factors **A1, A2, …,**, **B1, B2, …** in the matrix expression ($$) can be ***one of the 6 following types***:

1. a ***simple matrix*** **A1** with elements **A1[i, k].**
2. a ***scalar factor***, say **alpha**. This produces **A1[i, k]** = **alpha** for all **i**, **k**. Such scalars must be ***named***, unsubscripted or subscripted variables or parameters, ***not*** expressions or literal numbers.
3. a ***pseudorandom factor*** **ran()**. This produces successive pseudorandom matrix elements **A1[i, k]** uniformly distributed between -1 and 1. The pseudorandom seed can be initialized as shown in Table 2-1. A **MATRIX** assignment containing a pseudorandom factor is necessarily a sampled-data operation.
4. a ***direct product of two vectors*** **x * y**, which contributes **A1[i, k] = x[i] * y[k]**. The vectors **x** and **y** must have appropriate dimensions.
5. A ***logical product of two vectors*** **x & y**, which contributes **A1[i, k] = min(x[i], y[k])**. This is useful for implementing fuzzy-set logic.
6. a ***primitive sum*** **(*matrix* ± *scalar*)** or **(*matrix* ± *matrix)*** enclosed in parentheses, where ***matrix*** a matrix of Type 1 to 5, and ***scalar*** is a named unsubscripted or subscripted scalar variable or constant (not an expression or literal number).

7. a ***library function*** **f(X)** of a matrix of types 1 to 6.

As an example of such a matrix expression, consider

MATRIX W = alpha * X1 + beta * X2 * (X3 - gamma) + u * v + cos(S)

where **X1, X2, X3,** and **S** are matrices with the same dimensions as **W**, and **u, v** are appropriately declared and dimensioned vectors. **alpha** and **beta** are scalars represented by matrices with all-equal elements and the same dimensions as **W**. **gamma** is scalar represented by a matrix with the same dimensions as **X3**.

The special library function **recip(X)** produces the reciprocal **1/X[i, k]** of each array element **X[i, k].**

8-2. Output-limited Matrix Expressions

The assignment

MATRIX W = ***MATRIX expression*** **; min, max**

keeps the matrix expression within specified bounds. **min** and **max** can be scalar variables. Different **MATRIX** statements can have different bounds.

NOTE: In some versions of DESIRE, **min** and **max** must be symbolic variables rather than expressions or literal numbers.

8-3. Masking Matrix Expressions. The modified **MATRIX** assignment

MATRIX W = [WMASK] ***matrix expression***

skips computation of all elements **W[i, k]** such that **WMASK[i, k]** = 0, where **WMASK** is a ***mask matrix*** with the same dimensions as **W**. The use of an appropriately defined mask matrix can save time if the matrix **W** is sparse. Bounds **min, max** may be specified as in Sec. 5-13.

DIFFERENCE EQUATIONS

8-4. **Vectr delta** *Operations* (**UPDATE, delta**, or **Delta** in older versions of DESIRE)

(a) A DYNAMIC-segment statement

Vectr delta x = ***vector expression***

is equivalent to the recursive **Vector** assignment (vector difference equation, Sec. 5-19)

Vector x = x + ***vector expression***

which computes and then accumulates samples of a vector expression (Secs. 5-5 to 5-10) in successive time steps. The vector components **x[i]** are state variables and require inital values, which default to 0.

Such difference equations can, in particular, implement ***Euler integration*** of vector differential equations for much larger arrays than we can handle with **Vectr d/dt** (neural networks, partial differential equations). Note, however, that initial values of **Vectr delta** state variables are ***not*** reset by **reset** and **drunr** (see also Sec. 5-19).

8-5. Matrix Difference Equations: Recursive Assignments. Much as in Secs. 2-13 and 5-19, recursive **MATRIX** assignments can implement ***vector difference equations***, as in

MATRIX W = alpha * W + x * y + . . .

Note that (1) the matrix elements **x[i, k]** are state variables and require initial values and (2) they are not automatically reset by **reset** or **drunr**.

8-6. Recursive Matrix Updating:* DELTA *Operations (**LEARN** in earlier versions**)**

(a) A DYNAMIC-segment statement

DELTA W = ***matrix expression***

is equivalent to the recursive **MATRIX** assignment (matrix difference equation)

MATRIX W = W + ***matrix expression***

which computes and then accumulates samples of a **MATRIX** expression (Secs. 8-1 to 8-3) in successive time steps. The matrix elements **W[i, k]** are state variables and require initial values, which default to 0.

DELTA difference equations can, in particular, implement Euler integration of matrix differential equations. As in the case of **Delta**, initial values of **DELTA** state variables are ***not*** reset by **reset** and **drunr** (see also Sec. 5-19).

(b) The assignment

DELTA W = ***matrix expression*****; min, max**

limits all elements **W[i, k]** of the output **W** to a specific range and stops accumulation at the limits.

(c) Much as in Sec. 8-3

DELTA W = [WMASK] ***matrix expression***

suppresses the updating operations for matrix elements **W[i,k]** such that **WMASK[i,k]** = 0.

CLEARN, INTP, and PLEARN OPERATIONS

8-7.* CLEARN *Facilities

(a) The DYNAMIC-segment statement

CLEARN v = W(x) lrate, crit

implements a variety of competitive-learning operations. **x** is an **m**-dimensional input ***pattern vector***, **v** is the **n**-dimensional output ***category-code vector***, and **W** is a ***template matrix*** with **n** rows and **m** columns. The operation depends on the values of the named scalars **lrate** and **crit**, which must not be literal numbers or expressions. **lrate** is a ***learning rate***, often a decreasing function of the time or trial number **t**.

1. for **crit < 0**, **CLEARN** implements ***simple template-matching competitive learning***. At each trial step, the algorithm first clears all **v[i]** to 0 and then finds the **v**-index **i = I** which minimizes the squared template-matching error

$$E(x, i) = \sum_{k=1}^{m} (W[i, k] - x[k])^2 \qquad (2)$$

The winning **v[I]** is set to 1, and the template weights **W[I, k]** are updated to improve template matching:

W[I, k] = W[I, k] + lrate * (x[k] - W[I, k]) (k = 1, 2, …)

2. for **crit = 0**, **lrate > 0**, **CLEARN** implements ***frequency-sensitive competitive learning*** (FSCL, due to Ahalt *et al.*), a "conscience-type" algorithm. You ***must*** declare a selection-counting array **h** of the same dimension as the category array **v** ***immediately after*** **v**,

ARRAY … , v[n], h[n],. . . or **ARRAY …, v[n] + h[n] = dummy, …**

Let the **h[i]** default to 0 (or initialize them to a very small positive value, say 0.00001 if you want to use logarithms to compute entropy), and program

```
CLEARN v = W(x)lrate, crit
Vectr delta h = v
```

CLEARN then multiplies each squared template-matching error (2) by **h[i]** before comparing them. This tends to make the category counts **h[i]** more nearly equal.

3. for **crit = 0, lrate < 0**, **CLEARN** implements Oja's ***Rival-penalized Competitive Learning (RPCL)***, which acts like FSCL but also "penalizes" the ***second-best*** template row, say with **i = I2**, by updating it so that

W[I2, k] = W[I2, k] - 0.05 * lrate * (x[k] - W[I2, k]) (k = 1, 2, …)

This tends to drive all but one template vector away from each cluster center. While this does not always work, RPCL may be preferable to FSCL.

4. for **crit** > 0, **CLEARN** efficiently emulates ***adaptive-resonance*** behavior. ***This new version is a substantial improvement over the*** **CLEARN** ***operation described in our MIT Press text***. **x** ***need no longer be normalized*** (you can still use normalization if that should prove desirable). Also, ***the initial random template components no longer need to be initialized with small values***.

CLEARN now tags templates (rows of **W**) as ***committed*** when they first meet the vigilance-matching criterion **E(x, I) < crit**. **CLEARN** then eliminates committed templates from the competition if **E(x, I) > crit** .

5. ***Fast-learn Mode***. If **crit** is positive and is ***marked with a pound sign #***, then **CLEARN** similarly imitates adaptive resonance but immediately sets each selected template to **W[I, k] = x[k]**. Try the example programs **alpha.lst** and **cpgood.lst** with and without the pound sign on **crit**.

The new version of **CLEARN** ***requires you to declare a committed-tag array, say*** **h,** ***of dimension*** **n** ***immediately after the categorizer-code array*** **v,** thus

ARRAY … , v[n], h[n], … or **ARRAY …, v[n] + h[n] = dummy, …**

All user programs written for the old **CLEARN** work with the new version if you add the committed-tag-array declaration; you may also have to readjust the values of **crit** and/or **lrate**. DESIRE also still accepts the older syntax **CLEARN v = W * x; lrate, crit**.

The template-matrix elements **W[i, k]** are difference-equation state variables. If you do not specify their initial values they will start at 0. Such initial values are not reset by **reset** or **drunr**.

(b) For normal learning, **lrate** > 0. **lrate** = 0 ***suppresses the updating operation entirely*** to produce fast recall.

(c) Programming

Vector z = W% * v

after **CLEARN** produces ***the currently selected template vector*** (template-matrix row) **z**, where **z[k]** = **W[I, k]** (**k = 1, 2, . . ., m**).

8-8. **INTP** *Operation*

The DYNAMIC-segment statement

INTP v = W * x; thresh

where **x** and **v** are vectors, and **W** is a matrix, sets **v[i]** = 1 if

$$\mathbf{E(x, i)} = \sum_{\mathbf{k=1}}^{\mathbf{m}} \mathbf{(W[i, k] * x[k] - v[i])}^2 < \mathbf{thresh}$$

thresh is a named positive scalar (not a scalar expression). The matrix elements **W[i, k]** again require initial values, which are not reset by **reset** or **drunr**.

8-9. **PLEARN** *Principal-axes Operations*

For random input pattern vectors **x** of dimension **nx**, with nonsingular correlation matrix

$$\mathbf{C = E\{ x^T x \}}$$

(where $\mathbf{x^T x}$ is an outer product, not an inner product) the DYNAMIC-segment statement

PLEARN y = W * x

recursively trains the transformation matrix **W** to produce the ***principal-axes transformation*** of the correlation matrix **C** if the dimension **ny** of the vector **y** equals **nx**. For **ny** < **nx**, the **y** space is spanned by the first **ny** eigenvectors of **C**. **PLEARN** implements the algorithm described by T. D. Sanger in *Neural Networks*, **2**, 499, 1989.

FAST COMPILATION AND MODEL GENERALITY

8-10. Use of Parentheses vs. Compiler Speed

To permit the extraordinarily fast compilation needed for truly interactive experiments, **MATRIX** expressions are kept simple:

1. ***the only parentheses permitted*** in **MATRIX** expressions enclose "primitive sums" defined in Secs. and 8-1.

2. **MATRIX** ***function arguments*** can be scalars, simple matrices **W,** "primitive sums" of matrices, or product matrices **x * y**.

3. **MATRIX** expressions admit named, unsubscripted or subscripted scalars, but ***not*** expressions or literal numbers.

8-11. Writing More General **MATRIX** *Expressions*

(a) "Primitive sums" (Sec. 5-5c) can be used as product terms and as function arguments. Thus new versions of DESIRE accept **MATRIX** assignments like

```
MATRIX X = alpha * (x * y  + Z)
MATRIX y = cos(W + alpha)
```

(b) You can produce more completely general matrix expressions by simply programming extra assignments as needed. For example,

```
MATRIX Q = tan(2.5 * x * (A * x + B * y + alpha))
```

is illegal, but

```
Vector p = 2.5 * x   |   Vector y = A * x + B * y + alpha
MATRIX Q = tan(p * y)
```

does the same job very well.

Table 8-1. Examples of MATRIX Assignments

(**a, b, alpha** are scalars; **u, v** are vectors; **W, A, B** are rectangular matrices)

MATRIX W = a * A + b * B	**W[i, k] = a A[i,k] + b B[i, k]**
MATRIX W = u * v	**W[i, k] = u[i] v[k]**
MATRIX W = recip(A)	**W[i, k] = 1/A[i, k]**
MATRIX W = sin(A)	**W[i, k] = sin(A[i, k])**
MATRIX W = (A - alpha)	**W[i, k] = A[i,k] - alpha**
MATRIX W = u & v	**W[i, k] = min(u[i], v [k])**

LIST OF DESIRE AND DESIRE/NEUNET ERROR MESSAGES

ERROR 1 : undefined statement ;
ERROR 2 : array subscript out of bounds ;
ERROR 3 : arithmetic overflow ;
ERROR 4 : **RETURN** without **CALL** ;
ERROR 5 : attempt to divide by zero ;
ERROR 6 : floating-point format error ;
ERROR 7 : missing or misplaced parenthesis ;
ERROR 8 : more than one display/type list, stash list, recover list, or **SAMPLE** statement ;
ERROR 9 : square root argument < 0, logarithm argument <=0, or power base <=0 ;

ERROR 10 : illegal conditional operator, or missing THEN ;
ERROR 11 : cannot **CONNECT** %s ;
ERROR 12 : input/output line too long ;
ERROR 13 : bad line number, or out of range ;
ERROR 14 : illegal or missing variable, or missing parenthesis ;
ERROR 15 : identifier too long ;
ERROR 16 : only one stash list is allowed ;
ERROR 17 : display/type or stash/recover list too long ;
ERROR 18 : cannot read ;
ERROR 19 : array %s was not declared ;

ERROR 20 : INTEGER overflow ;
ERROR 22 : this statement is illegal in command mode ;
ERROR 23 : missing **TO**, or missing **STEP** value in **FOR** statement ;
ERROR 24 : missing **NEXT**, **END WHILE**, or **UNTIL**; or incorrect nested loops ;
ERROR 25 : missing bracket, comma, or semicolon in array or variable list ;
ERROR 26 : unmatched quotes in statement ;
ERROR 27 : **NEXT**, **END WHILE**, or **UNTIL** without corresponding **FOR**, **WHILE**, or **REPEAT**; or incorrecty nested loops or **IF/ELSE** clauses ;

ERROR 28 : multiple declaration of variable or array %d" ;
ERROR 29 : missing equal sign, or unexpected char. before = ;

ERROR 30 : channel or file not **CONNECTed** ;
ERROR 31 : operation restricted to REAL and INTEGER quantities ;
ERROR 32 : dimension of array %d is <=0 or too large\n" ;
ERROR 33 : illegal termination of a statement or variable, ; or illegal variable type ;
ERROR 34 : supply device and/or file name, in single quotes ;
ERROR 35 : undefined statement - missing blank or tab ?;
ERROR 36 : illegal argument; check for multiple declaration ;
ERROR 37 : command mode or illegal syntax in **FUNCTION** or **PROCEDURE** ;
ERROR 38 : file %s not found on the specified device ;
ERROR 39 : user-area overflow - program is too large ;

ERROR 40 : out of storage for REAL or COMPLEX values ;
ERROR 41 : out of storage for INTEGER values ;
ERROR 42 : operator stack overflow ;
ERROR 43 : **GO** without **STOP**, or program change before **GO** ;
ERROR 44 : illegal argument delimiter, or missing argument ;
ERROR 45 : illegal file name or file extension; - too long? ;
ERROR 46 : missing = sign! ;
ERROR 47 : argument of **SQRT** < 0 ;

ERROR 48 : loop control variable must be unsubscripted REAL ;
ERROR 49 : bad CONNECT statement ;

ERROR 50 : illegal channel number ;
ERROR 51 : out of data! ;
ERROR 52 : illegal recursion in a function or procedure ;
ERROR 53 : wrong number of dimensions in array element, or missing bracket ;
ERROR 54 : + or - expected after E in E-format ;
ERROR 55 : function or procedure is already defined ;
ERROR 56 : arguments in function/procedure call do not match ;
ERROR 57 : illegal octal or hexadecimal number ;
ERROR 58 : PROCEDURE or SUBMODEL without end, or incorrectly nested loops or IF/ELSE clauses ;
ERROR 59 : function %s not defined in DYNAMIC segment ;

ERROR 60 : missing operator, or illegal variable ;
ERROR 61 : FFT array is not properly defined ;
ERROR 62 : only comments may follow LABEL or RETURN on the same line ;
ERROR 63 : Channel %d already CONNECTed ;
ERROR 64 : Channel %d not CONNECTed" ;
ERROR 65 : Channel 0 or 1 is DISCONNECTed! ;
ERROR 66 : no such device! ;
ERROR 67 : Channel %d not CONNECTed on DISCONNECT" ;
ERROR 68 : non-existent target in GO TO or RESTORE ;
ERROR 69 : too many dimensions ;

ERROR 70 : undefined symbol %s in vector/matrix expression ;
ERROR 71 : dummy variable %s appears outside its FUNCTION, PROCEDURE, or SUBMODEL definition ;
ERROR 72 : n for FFT must be a power of 2 between 32 and 16384 ;
ERROR 73 : same symbol %s used for variable or procedure and array ;
ERROR 74 : attempt to use a non-conformable matrix ;
ERROR 75 : illegal syntax in vector/matrix or complex expression ;
ERROR 76 : illegal operator in DYNAMIC segment ;
ERROR 77 : illegal value of line number increment ;
ERROR 78 : cannot close channel ;
ERROR 79 : LABEL must be at start of line ;

ERROR 80 : dimension <1, out of range, or too many dimensions ;
ERROR 81 : only comments, ELSE, or PROCEED may follow GO TO, EXIT,
RESTORE, OLD, CHAIN, drun LABEL statements on the same line ;

ERROR 82 : illegal or duplicated LABEL ;
ERROR 83 : we can print arrays, but not matrix expressions! ;
ERROR 84 : attempt to use an illegal type of variable, %s ;
ERROR 85 : illegal recursive matrix operation ;
ERROR 86 : no such integration rule! ;
ERROR 87 : specify TMAX and check DT ;
ERROR 88 : nested function or procedure definition ;
ERROR 89 : GO TO was switched OFF!! ;

ERROR 90 : user-area overflow in DYNAMIC program segment ;
ERROR 91 : statement illegal in command mode ;

ERROR 92 : no stash list, or drun(r)* - rerun? ;
ERROR 93 : CHECKN is larger than order of system ;

ERROR 94 : variable-integration-step deadlock ;
ERROR 95 : cannot DISCONNECT ;
ERROR 96 : explicit array elements illegal in stash list ;
ERROR 97 : maximum PIC version number is 999 ;
ERROR 98 : illegal EXIT operation ;
ERROR 99 : no program to save ;

ERROR 100 : too many state equations ;
ERROR 101 : multiple definition of state variable %s ;
ERROR 102 : derivative code in interpreter segment, ; or jump into DYNAMIC program segment ;
ERROR 103 : can't DISCONNECT channel %d ;
ERROR 104 : attempted multiple or illegal use of DYNAMIC ;
ERROR 105 : delay is negative or too large ;
ERROR 106 : cannot write ;
ERROR 108 : no subscripted state variables in this Version ;
ERROR 109 : DOS hardware error: check device ;

ERROR 110 : illegal sampling rate ;
ERROR 111 : IF without ELSE, ELSE without PROCEED, or incorrect IF/ELSE/PROCEED nesting ;
ERROR 112 : missing or duplicate SHOW without argument ;
ERROR 113 : ABORT: illegal memory reference or C error ;
ERROR 121 : cannot load new shell ;
ERROR 122 : DT must be CONSTANT for tdelay ;
ERROR 123 : matrix is singular or ill-conditioned ;
ERROR 124 : iRow must be greater than 0 ;
ERROR 125 : illegal erun or load: need a NEW editor window! ;
ERROR 196 : illegal dummy argument in FUNCTION, PROCEDURE, or SUBMODEL;
ERROR 197 : Escape from INPUT - variable unchanged, or 0 if new ;
ERROR 198 : line number out of range ;
ERROR 199 : this statement may be used ONLY in command mode ;

ERROR 200 : variable or function %s is not defined ;
ERROR 201 : procedure or submodel not defined ;
ERROR 202 : unimplemented data type! ;
ERROR 206 : channel number needed in CONNECT statement ;
ERROR 207 : no compilation in command mode - use run command ;
ERROR 208 : display 2 must have a running display ;
ERROR 209 : subscripted ARRAY element is illegal here ;
ERROR 210 : XWindows error ;
ERROR 215 : no DYNAMIC statement or DYNAMIC program segment ;
ERROR 218 : illegal non-REAL or subscripted variable in DYNAMIC segment ;
ERROR 219 : statement illegal in DYNAMIC program segment ;

ERROR 220 : ERROR 222 : ERROR 223 : ERROR 228 : variable-integration-step deadlock ;
ERROR 224 : integrator is at lower DT limit ;
ERROR 225 : requested ERMAX is too small ;
ERROR 226 : integrator corrector did not converge ;
ERROR 227 : implicit integrator cannot solve equations ;

ERROR 229 : irule 10 or 14 needs a JACOBIAN segment ;
ERROR 230 : only one JACOBIAN per segment ;
ERROR 231 : click to exit graphics ;

ERROR 232 : we need more templates! ;
ERROR 300 : statement illegal in interpreter segment ;

Table 3-$$. Default Values of Desire Simulation Variables

1. The display scale **scale** defaults to **1**.

2. The number **NN** of sampling points defaults to **251** or, if **TMAX** and **DT** are specified, to **TMAX/DT + 1** if that is smaller.

3. ***If the first DYNAMIC segment to run contains differential equations*** (**d/dt** and/or **Vectr d/dt** statements), then the integration rule defaults to **irule 1** (2nd-order fixed-step Runge-Kutta-Heun integration), and the first **drun** or **drunr** sets the initial time **t0** to **0**. **DT** defaults to **COMINT/2 = TMAX/2(NN - 1)**, and an error message warns you if the total run time **TMAX** is not specified.

4. ***If the first DYNAMIC program segment does not contain differential equations*** (e.g. if the DYNAMIC segment is used only for difference equations or graph plotting) then the integration rule defaults to **irule 0** (no integration). The first **drun** or **drunr** then sets **t0** to **1**, and **TMAX** to **NN - 1**. Thus **t** (which labels time-history graphs and listings) conveniently steps through successive ***trial numbers*** **1, 2, . . ., NN**.

5. Note that these defaults apply only to the first DYNAMIC program segment to run. The experiment protocol must explicitly set values for the other DYNAMIC segments.

The integration parameters **DTMIN, DTMAX, ERMIN, ERMAX,** and **CHECKN** used with variable-step integration (Table $$) are normally set by the experiment protocol. Otherwise the first **drun** or **drunr** sets these parameters to harmless but possibly not useful default values.

DESIRE/2000 INDEX

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z