PV-WAVE *Advantage*™
PV-WAVE Command Language™

User's Guide

Visual Numerics™
IMSL and Precision Visuals are now one.

Version 4.2 • • • •
2862Version
4.2
2861
June 1993Ver-
sion 4.2

Visual Numerics, Inc.

Corporate Headquarters	Boulder, Colorado	France
Suite 400, 9990 Richmond Avenue Houston, Texas 77042 United States of America 713/784-3131 FAX: 713/781-9260	6230 Lookout Road Boulder, Colorado 80301 United States of America 303/530-9000 FAX: 303/530-9329	33-1-42-94-19-65 FAX: 33-1-42-94-94-22 33-1-34-51-26-26 FAX: 33-1-34-51-97-69
Germany	**Japan**	**United Kingdom**
49-211-367-7122 FAX: 49-211-367-7100	81-3-5689-7550 FAX: 81-3-5689-7553	44-0753-790-600 FAX: 44-0753-790-601

Contents Summary

Table of Contents

Preface

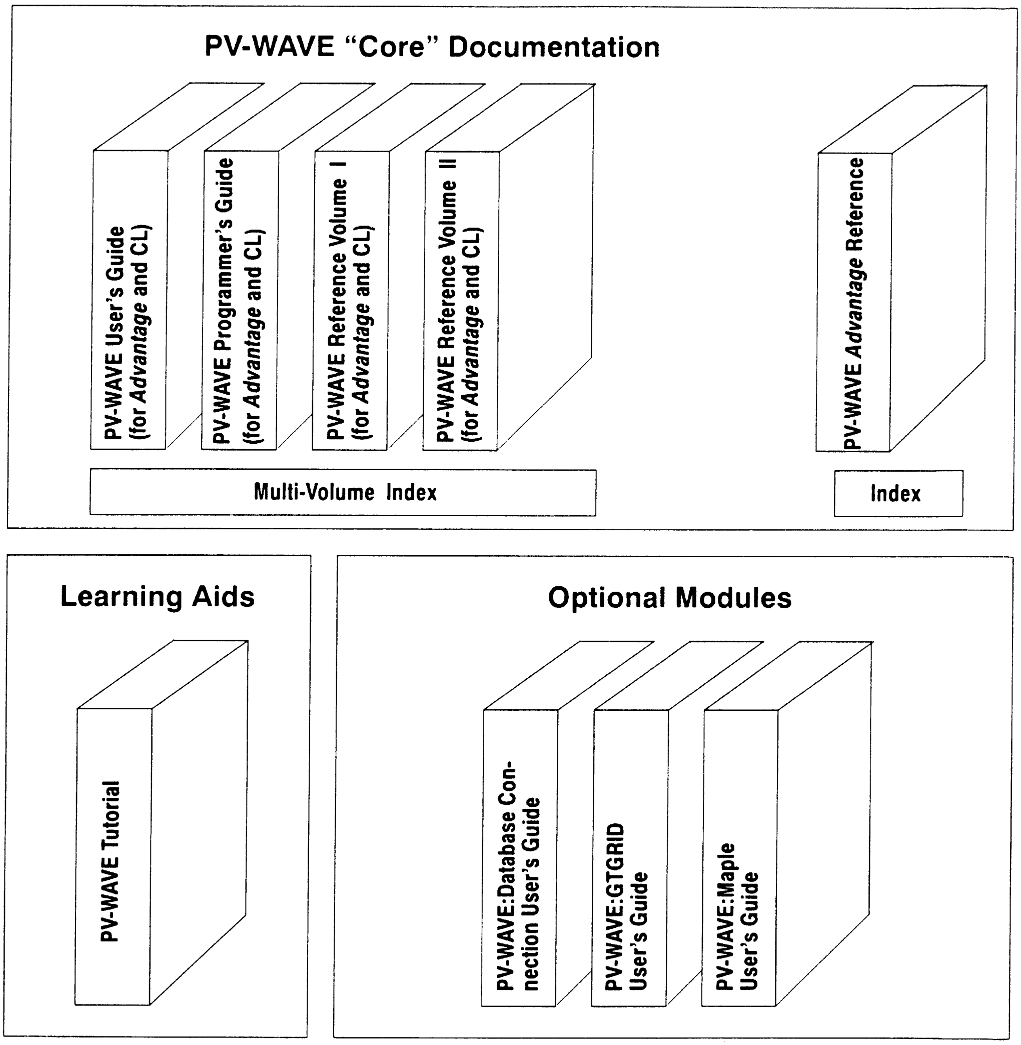
This user's guide is part of a larger set of documentation; the entire set is shown in Figure I. Start with the *PV-WAVE Tutorial*, and then refer to this user's guide for other fundamental information about how to use the numerous features of PV-WAVE Command Language. Advanced users will also want to refer to the *PV-WAVE Programmer's Guide for Advantage and CL* and the *PV-WAVE Reference for Advantage and CL*, *Volumes I* and *II* for detailed information.

For your convenience, all the documents shown in Figure I are available online, as well as being available in a hardcopy format. Additional copies of the hardcopy documentation can also be ordered from Visual Numerics, Inc., by calling 800/447-7147.

Contents of this User's Guide

This user's guide contains the following chapters:

- **Preface** – Describes the contents of this guide, lists the typographical conventions used, explains how to use the PV-WAVE documentation set, and explains how to obtain customer support.



PV-WAVE "Core" Documentation
PV-WAVE User's Guide (for *Advantage* and CL)
PV-WAVE Programmer's Guide (for *Advantage* and CL)
PV-WAVE Reference Volume I (for *Advantage* and CL)
PV-WAVE Reference Volume II (for *Advantage* and CL)
PV-WAVE *Advantage* Reference
Multi-Volume Index
Index
Learning Aids
PV-WAVE Tutorial
Optional Modules
PV-WAVE:Database Connection User's Guide
PV-WAVE:GTGRID User's Guide
PV-WAVE:Maple User's Guide

Figure I PV-WAVE documentation set; for more information about any one book you see shown here, refer to its preface, where the contents of each chapter are explained briefly. All documents are available both online and in a hardcopy format. Additional copies of the hardcopy documentation can be ordered by calling Visual Numerics, Inc., at 303/447-7147.

- **Chapter 1: Overview** – Provides an overview of the topics discussed in this manual.
- **Chapter 2: Getting Started** – Discusses some of PV-WAVE's basic operations, such as starting and stopping the software, using the online documentation system, running the software, special characters, journaling, saving and restoring sessions, and modifying your environment.
- **Chapter 3: Displaying 2D Data** – Covers the basics of X versus Y plotting.
- **Chapter 4: Displaying 3D Data** – Describes the basics of contour and surface plotting.
- **Chapter 5: Displaying Images** – Describes routines used for displaying images and image processing.
- **Chapter 6: Guide to Advanced Rendering** – Describes the routines used to render volumes using ray tracing techniques.
- **Chapter 7: Working with Date/Time Data** – Explains how to create plots with a Date/Time axis.
- **Chapter 8: Creating and Querying Tables** – Discusses how to create and subset tables of data using SQL-like functions.
- **Chapter 9: Software Fonts** – Discusses how to use and format PV-WAVE's software, or vector-drawn, fonts. This chapter also discusses the difference between software and hardware fonts and how to choose between them.
- **Chapter 10: Using Color in Graphics Windows** – Discusses color systems and introduces the routines that control color tables and plot colors.
- **Appendix A: Output Devices and Window Systems** – Explains how to use the standard graphic output devices and window systems supported by PV-WAVE.
- **Appendix B: Picture Index** – A graphical index of the illustrations that appear in the PV-WAVE documentation set.
- **Index** – A multivolume index that includes references to the *PV-WAVE User's Guide*, *PV-WAVE Programmer's Guide*, as well as both volumes of the *Reference*.

Typographical Conventions

The following typographical conventions are used in this guide:

- PV-WAVE code examples appear `in this typeface.` For example:

  ```
  PLOT, temp, s02, Title='Air Quality'
  ```

- Code comments are shown in this typeface, below the commands they describe. For example:

  ```
  PLOT, temp, s02, Title='Air Quality'
  ```
  This command plots air temperature data vs. sulphur dioxide concentration.

  Comments are used often in this reference to explain code fragments and examples. Note that in actual PV-WAVE code, all comment lines must be preceded by a semicolon (;).

- PV-WAVE commands are not case sensitive. In this reference, variables are shown in lowercase italics (*myvar*), function and procedure names are shown in uppercase (XYOUTS), keywords are shown in mixed case italic (*XTitle*), and system variables are shown in regular mixed case type (!Version). For better readability, all widget routines are shown in mixed case (WwMainMenu).

- A $ at the end of a PV-WAVE line indicates that the current statement is continued on the following line. By convention, use of the continuation character ($) in this document reflects its syntactically correct use in PV-WAVE. This means, for instance, that *strings* are never split onto two lines without the addition of the string concatenation operator (+). For example, the following lines would produce an error if entered literally in PV-WAVE:

  ```
  WAVE> PLOT, x, y, Title = 'Average $
     Air Temperatures by Two-Hour Periods'
  ```
  Note that the string is split onto two lines; an error message is displayed if you enter a string this way.

  The correct way to enter these lines is:

```
WAVE> PLOT, x, y , Title ='Average '+$
   'Air Temperatures by Two-Hour Periods'
```

This is the correct way to split a string onto two command lines.

- The | symbol means "or" when used in a usage line. It is not to be typed. For example, in the following command:

  result = QUERY_TABLE(*table*,
  ' [Distinct] * | col_i [*alias*] [, ..., col_n [*alias*]] ...

  the | means use either * or col_i [*alias*] [, ..., col_n [*alias*]], but not both.

- Reserved words, such as FOR, IF, CASE, are always shown in uppercase.

Customer Support

If you have problems unlocking your software or running the license manager, you can talk to a Visual Numerics Customer Support Engineer. The Customer Support group researches and answers your questions about all Visual Numerics products.

Please be prepared to provide Customer Support with the following information when you call:

- The name and version number of the product. For example, PV-WAVE 4.2 or PV-WAVE P&C 2.0.
- Your license number, or reference number if you are an Evaluation site.
- The type of system on which the software is being run. For example, Sun-4, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, SunOS 4.1.3.
- A detailed description of the problem.

The phone number for the Customer Support group is 303/530-5200.

Trademark Information

1

CHAPTER

Overview

This manual is a user's guide for PV-WAVE Command Language, a software system that lets you explore and understand your scientific, engineering, or commercial data visually and quickly. Integrated graphics, data analysis, image processing, data animation, tabular manipulation of record-based data, and custom application development make PV-WAVE a productive Visual Data Analysis (VDA) environment.

You can enter and execute PV-WAVE commands interactively at the keyboard, or combine them into powerful applications. As a high-level, interpretive programming language, PV-WAVE is ideal for VDA tasks ranging from ad-hoc programming projects to the development of complete, organization-wide solutions.

High-Level PV-WAVE Features

This section provides a brief discussion of PV-WAVE's high-level graphics, data manipulation, and mathematical features. The topics include:

- ✔ graphics routines
- ✔ image processing

- ✔ handling date/time data
- ✔ manipulating tables of data
- ✔ advanced rendering
- ✔ data manipulation functions
- ✔ mathematical functions
- ✔ producing output

PV-WAVE *Advantage* users, see the *PV-WAVE Advantage Reference* for detailed information on all of the *Advantage* functions and procedures. PV-WAVE *Advantage* gives you access to all of the functionality of PV-WAVE Command Language, plus many more functions for mathematical, statistical, and scientific computing.

Graphics Routines

The next two subsections describe basic and advanced graphics routines available in PV-WAVE.

Displaying Basic Graphics

PV-WAVE allows data to be displayed in a variety of ways. The most common ways are listed below. Each method, by itself, allows you to get a quick look at your data by displaying a simple, "stripped-down" plot. The procedure provides intelligent defaults for the axis range and tick mark values. Thus, if you just want to take a quick look at the data, the basic procedure is all you need.

Regardless of the way you are displaying your data, a clear understanding of the principles for creating and modifying 2D plots is beneficial. Many of the principles, keywords, and system variables are shared by all of the data display procedures.

The basic plotting types include:

- **2D Line/Scatter Plots** – PLOT, OPLOT
- **Polar Plots** – PLOT with *Polar* keyword
- **Logarithmic Plots** – PLOT_IO, PLOT_OI, PLOT_OO

- **Contour Plots** – CONTOUR
- **Wire Mesh Plots** – SURFACE
- **Calendar Axis Plots** – Any plot showing date/time data along the X axis.

For more information, see Chapter 3, *Displaying 2D Data*, and Chapter 4, *Displaying 3D Data*. Also refer to Chapter 2 *Procedure and Function Reference*, in the *PV-WAVE Reference*. For information on date/time data, see Chapter 7, *Working with Date/Time Data*.

Modifying Basic Plots

To finalize your graphics display, you will want to customize it using keywords. All keywords can be abbreviated and can be listed in any order. Keywords that are either enabled or disabled may be written with a forward slash in front of the keyword to set it equal to 1, which is equivalent to enabling that option. For example, `/Row` and `/Column` are equivalent to `Row=1` and `Column=1`. Many keywords have an X, Y, or Z in front of the name to specify the axis they apply to. Keywords may also correspond to system variables.

All the basic plot types can be modified by using keywords or additional routines to fit your needs. Some of the more commonly used modifications include:

- Multiple plots per page.
- Using color, line styles or symbols.
- Combining different display techniques.

For a description of the keywords that help you accomplish these goals, refer to Chapter 3 *Graphics and Plotting Keywords*, in the *PV-WAVE Reference*.

Creating Advanced Graphics

Along with the basic display techniques, you can opt for more complex graphics. Following is a list of some of the more common functions and procedures used to produce special effects:

- **Multiple Axes on One Plot** – AXIS
- **Interactive Graphics** – CURSOR, TVCRS, DEFROI, and PROFILES, WgIsoSurfaceTool, and others.
- **Shading Surfaces** – POLYSHADE, SHADE_SURF
- **Animation** – MOVIE, WgMovieTool, WgAnimateTool
- **Advanced Rendering** – RENDER, CONE, CYLNDER, SPHERE, MESH, VOLUME

Check the Users' Library for even more functions and procedures. For more information about libraries, see Chapter 10, *Programming with PV-WAVE*, in the *PV-WAVE Programmer's Guide*. For more information about data display, see Chapter 4, *Displaying 3D Data*, and Chapter 5, *Displaying Images*.

Image Processing

If you want to do image processing, you'll appreciate the many routines available for displaying images. You can also convert conventional plots into images or manipulate and enhance images.

Displaying Images

PV-WAVE offers many options for displaying images. You can simply display the data, or you can easily enhance your image display with many built-in image display tools. A brief list includes:

- **Creating or Modifying Color Tables** – ADJCT, C_EDIT, COLOR_EDIT, LOADCT, PALETTE, WgCeditTool, WgCBarTool
- **Histogram Equalization** – HIST_EQUAL_CT, HIST_EQUAL
- **Displaying Images on Screen** – TV, WgSimageTool

- **Scaling and Displaying Images** – TVSCL
- **Morphologic Building Blocks** – DILATE, ERODE

For more information on displaying and manipulating image data, see Chapter 5, *Displaying Images.*

Processing Images

In addition to traditional image processing applications, many applications (such as the display of time-series data) that usually have been displayed as XY plots can be easily converted into 2D arrays that can be used by the image processing routines. An example of this can be seen in the "Time Series" example in the PV-WAVE Demonstration Gallery. (You start this demonstration by typing `demo` at the `WAVE>` prompt.)

A partial list of routines that are useful when working with images and 2D arrays includes:

- **Shrinking or Expanding an Image** – CONGRID or REBIN
- **Histogram Equalization of the Image** – HIST_EQUAL
- **Convolving Two Vectors or Arrays** – CONVOL
- **Creating Filters** – DIGITAL_FILTER, HANNING, LEEFILT, MEDIAN
- **Warping of Images** – POLYWARP
- **Edge Enhancement** – ROBERTS, SOBEL

For more information, see Chapter 5, *Displaying Images.*

Handling Date/Time Data

Converting Your Data into Date/Time Data

If you are importing date/time data into PV-WAVE, four functions simplify converting this data into PV-WAVE date/time data. These functions are:

- **STR_TO_DT** – Converts string data or variables containing string data into PV-WAVE date/time variables.

- **VAR_TO_DT** – Converts numeric variables containing date/time information into PV-WAVE date/time variables.
- **SEC_TO_DT** – Converts seconds into PV-WAVE date/time variables.
- **JUL_TO_DT** – Converts the Julian day into a PV-WAVE date/time variable.

Generating PV-WAVE Date/Time Data

You can generate PV-WAVE date/time data for data files that do not have date and time stamps. There are two steps:

- ❑ Create an initial PV-WAVE date/time structure using one of four conversion functions: STR_TO_DT, VAR_TO_DT, SEC_TO_DT, or JUL_TO_DT.
- ❑ Use the DTGEN function to create a variable that contains an array of PV-WAVE date/time structures.

Manipulating Date/Time Data

PV-WAVE provides several functions for manipulating PV-WAVE date/time variables. These functions are:

- DT_ADD
- DT_SUBTRACT
- DT_DURATION
- CREATE_WEEKENDS
- CREATE_HOLIDAYS
- LOAD_HOLIDAYS
- LOAD WEEKENDS
- DT_COMPRESS

For details on all of the date/time functions, see Chapter 7, *Working with Date/Time Data.*

Manipulating Tables of Data

A table is a natural and easily understood way of organizing data into columns and rows. Many computer systems use the table model to organize large amounts of data. For example, a relational database stores all of its data in a tabular format.

The PV-WAVE table functions let you create tables and subset them in various ways. These functions are both powerful and easy to use. Tables, which you create with the BUILD_TABLE function, can be subsetted and manipulated with the QUERY_TABLE function. QUERY_TABLE, which closely resembles the Structured Query Language (SQL) SELECT command, is an easy to learn and conceptually natural way to access data in tables.

The PV-WAVE table functions include:

- **BUILD_TABLE** – Creates a new table from PV-WAVE numeric or string vectors (1D arrays) of equal length.
- **QUERY_TABLE** – Lets you subset, rearrange, group, and sort table data. This function returns a new table containing the query results.
- **UNIQUE** – Removes duplicate elements from any vector (1D array).

For detailed information on the table functions, see Chapter 8, *Creating and Querying Tables.*

Advanced Rendering

You can render 3D geometric and volumetric data using the advanced rendering capabilities of PV-WAVE. Most of these functions are part of the standard library. The RENDER function is a system routine that performs rendering using the ray tracing technique.

In addition, the standard library contains several utility functions for gridding (2D, 3D, 4D, and spherical) and for conversion of rectangular, polar, cylindrical, and spherical coordinates.

For detailed information on the advanced rendering routines, see Chapter 6, *Advanced Rendering Techniques.*

Data Manipulation Functions

PV-WAVE contains a large number of routines that allow you to search, evaluate, sort, and modify your data. In addition to the operators described earlier in this chapter, there are basic statistical routines, array analysis and manipulation routines, plus search and sort routines. Some of these routines include:

- **Basic Statistics** – AVG, CORRELATE, SIGMA, STDEV
- **Array Information** – DETERM, HISTOGRAM, MAX, MIN, TOTAL
- **Array Manipulation** – REBIN, REFORM, REVERSE, ROTATE, SHIFT, SMOOTH, TRANSPOSE
- **Search Utility** – WHERE
- **Sort Utility** – SORT
- **Table Functions** – BUILD_TABLE, QUERY_TABLE, and UNIQUE

For a complete list of array routines, see Chapter 1 *Functional Summary of Routines*, in the *PV-WAVE Reference.*

Note PV-WAVE Advantage offers many specialized routines for mathemetical, analytical, and scientific computing. PV-WAVE Advantage users, see the *PV-WAVE Advantage Reference* for information on these routines.

Mathematical Functions

There are many mathematical procedures and functions included in PV-WAVE to assist you in analyzing data. Furthermore, you may also create your own procedures and functions with PV-WAVE.

Here is a partial listing of the more commonly used mathematical functions:

- **Absolute Value** – ABS
- **Cross Product** – CROSSP
- **Derivatives** – DERIV
- **Determinant** – DETERM
- **Eigenvalues and Eigenvectors** – TRED2
- **Fast Fourier Transforms** – FFT
- **Linear Equation Solvers** – LUBKSB, LUDCMP, MPROVE, SVBKSB, TRIDAG
- **Curve-Fitting Algorithms** – CURVEFIT, GAUSSFIT, POLYFIT, POLYFITW

For a listing of mathematical functions, see Chapter 1 *Functional Summary of Routines*, in the *PV‑WAVE Reference.*

Note In addition, PV‑WAVE Advantage users have access to a much more varied and technically sophisticated library of mathematical routines. See the *PV‑WAVE Advantage Reference* for details.

Producing Final Output

PV‑WAVE supports hardcopy output to various plotters and printers. The SET_PLOT and the DEVICE procedures are used to select output devices and modify output characteristics. For descriptions of the SET_PLOT and DEVICE procedures, see Appendix A, *Output Devices and Window Systems.* This appendix also contains general information about how to produce hardcopy output and specific information on producing output for various devices and window systems.

Your Next Step

If you have never used PV-WAVE, the *PV-WAVE Tutorial* offers additional information on the basics of PV-WAVE and provides several examples for you to try.

With PV-WAVE, you can apply sophisticated graphics routines to a wide range of applications. Successful users have already realized how much more productive they can be when they use PV-WAVE for analyzing their data. PV-WAVE offers the features and performance that your analysis situation demands.

2

CHAPTER

Getting Started

This chapter discusses some of PV-WAVE's basic operations. If you are a novice PV-WAVE user, see the previous *Overview* chapter, and see the *PV-WAVE Tutorial* for basic concepts, lessons, and exercises. This chapter contains the following topics:

- *Starting PV-WAVE* on page 12 — Explains how to start
- PV-WAVE using the `wave` command.
- *Stopping PV-WAVE* on page 13 — Describes various methods for exiting, suspending, interrupting, and aborting PV-WAVE.
- *Using the Online Documentation System* on page 16 — Explains how to use the online documentation system.
- *Getting Information about the Current Session* on page 16 — Introduces the INFO procedure, which displays information about the current PV-WAVE session.
- *Running PV-WAVE* on page 17 — Explains how to run PV-WAVE interactively and with command files, procedures, functions, and main programs. The section also explains how to use executive commands and command recall.
- *Special Characters* on page 33 — Describes various special characters, such as the exclamation point (!), ampersand (&), and dollar sign ($).

- *Journaling* on page 36 – Tells you how to save a PV-WAVE interactive session using the JOURNAL procedure, and explains the two different uses of this procedure.
- *Saving and Restoring Sessions* on page 39 – Explains how to save variables and restore them for use in future sessions. This section discusses the syntax and keywords for the SAVE procedure and the RESTORE procedure.
- *Using PV-WAVE in Runtime Mode* on page 41 – Discusses how to run applications that have been previously compiled and saved with the COMPILE procedure.
- *Modifying Your PV-WAVE Environment* on page 44 – When you start PV-WAVE, an environment is automatically set up for you with the `wvsetup` file. This section describes various ways to change that environment. This section also explains how to change your PV-WAVE prompt, define keyboard accelerators, and use PV-WAVE with X Windows.

Starting PV-WAVE

Before running PV-WAVE, the `wvsetup` file (UNIX) or `WVSETUP.COM` file (VMS) must be executed. When this file, which is described in detail in your installation guide, has been executed, you are ready to start PV-WAVE.

Starting PV-WAVE Interactively

You initiate PV-WAVE from your operating system prompt. At the operating system prompt, type `wave` and press <Return>. The PV-WAVE prompt appears:

```
WAVE>
```

This places you in a mode where you can interactively enter commands at the `WAVE>` prompt. If you see an error and PV-WAVE does not start, see your installation guide for trouble-shooting information.

Executing a Command File at Startup

A command file is a file that contains PV-WAVE commands. When a command file is executed, each command in the file is executed. When the end of the file is reached, control reverts to the interactive mode, that is, the `WAVE>` prompt is displayed, and you can type commands from the keyboard. Also you may call the EXIT procedure from within the command file to exit PV-WAVE and return to the operating system prompt.

You can execute a command file directly at startup by entering the following at the operating system prompt:

`wave` *filename*

Note The filename must be a correctly constructed command file. It cannot be a PV-WAVE procedure file. Command files are explained in more detail in *Running a Command File* on page 19.

You can also set the environment variable (or VMS logical) `WAVE_STARTUP` to execute a command file when you enter the command that starts PV-WAVE. See *WAVE_STARTUP: Using a Startup Command File* on page 48.

Stopping PV-WAVE

The simplest way to stop PV-WAVE is to type `EXIT` or `QUIT` at the `WAVE>` prompt. Other more complicated methods of stopping include aborting, suspending, and interrupting. All these methods are explained in this section.

Exiting PV-WAVE

When you exit PV-WAVE, you are returned to the operating system prompt. Variable assignments are lost, but data that is buffered for open files is saved to these files before exiting is complete.

Exiting on a UNIX System

If you type `EXIT` or `QUIT` at the `WAVE>` prompt, you exit back to the operating system. Entering a <Control>-D as the first character on the command line performs the same function. If the <Control>-D is not the first character on the command line, it simply ends the input line as if a <Return> had been entered.

Exiting on a VMS System

If you type `EXIT` or `QUIT` at the `WAVE>` prompt, you exit back to the operating system. Entering a <Control>-Z as the first character on the command line performs the same function. If the <Control>-Z is not the first character on the command line, it ends the input line as if a <Return> had been entered. The input line is executed, and then PV-WAVE exits.

Suspending PV-WAVE

When you suspend PV-WAVE, you are returned to the operating system prompt; however, PV-WAVE is still running as a background process. All variables and their values are saved.

Suspending PV-WAVE on a UNIX System

<Control>-Z is the normal UNIX suspend character. Temporarily, it stops a process and places it in the background. Typing the suspend character suspends PV-WAVE and returns you to the shell process where you can enter one or more commands, for example, to run a text editor. After completing the commands, type `fg` to return PV-WAVE to the foreground.

Suspending PV-WAVE on a VMS System

There is no method for suspending PV-WAVE on VMS systems.

Interrupting the Current PV-WAVE Command

<Control>-C is the interrupt character. Typing the interrupt character generates a PV-WAVE *keyboard interrupt.* Under VMS, <Control>-C is always the interrupt character. However, under UNIX, the interrupt character can be changed by you outside of PV-WAVE. This is rarely done, so for the purposes of this manual, we assume the default convention.

When you type <Control>-C at the `WAVE>` prompt, the following message is displayed:

```
% Interrupt encountered.
```

When the interpreter regains control, you are returned to the `WAVE>` prompt. You can continue after interrupting PV-WAVE with the .CON command.

Aborting PV-WAVE

When you abort PV-WAVE, a message appears, such as `quit (core dumped)` and you are returned to the operating system prompt. Remove the core file before re-entering PV-WAVE.

Aborting on a UNIX System

As with any UNIX process, PV-WAVE may be aborted by typing <Control>-\. This is a very abrupt exit – all variables are lost, and the state of open files will be uncertain. Thus, although it can be used to get out of PV-WAVE in an emergency, its use should be avoided.

Note After aborting PV-WAVE in a UNIX environment, you may find that your terminal is left in the wrong state. You can restore your terminal to the correct state by issuing the UNIX command:

```
% reset
```

or

```
% stty echo -cbreak
```

Aborting on a VMS System

As with any VMS program, PV-WAVE may be aborted by typing <Control>-Y. Aborting PV-WAVE with <Control>-Y should only be used as an emergency measure since all the variables are lost and some output may disappear. It is possible to resume PV-WAVE by typing the DCL command.

```
$ CONTINUE
```

However, if any DCL command that causes VMS to run a new program is issued prior to the `CONTINUE` command, the PV-WAVE session is totally and irreversibly lost.

Using the Online Documentation System

To access PV-WAVE's online documentation, enter:

```
HELP
```

at the PV-WAVE command line. HELP opens the main window for the online documentation system. To see detailed information on using the online documentation, click the button About Online Help in this main window.

You can also run the online documentation system from the UNIX operating system level, by entering the following commands at the operating system prompt:

```
% source $WAVE_DIR/bin/wvsetup

% wavedoc
```

Getting Information about the Current Session

The INFO procedure provides information about the PV-WAVE session in progress.

Calling INFO with no parameters displays an overview of the session, including the current definitions of all of your variables. You can obtain more specific information about the session by providing keywords with the INFO command.

For example, `INFO, /Device` provides information about the current graphics device being used by PV-WAVE. `INFO, /Memory` reports the amount of dynamic memory in use and the number of times it has been allocated and deallocated. For more information about the INFO procedure, see Chapter 14, *Getting Session Information*, in the *PV-WAVE Programmer's Guide.*

Running PV-WAVE

The following subsections describe how to use PV-WAVE interactively and with program and command files. When you enter commands or create and run programs, procedures, and functions from the `WAVE>` prompt, you are using PV-WAVE interactively. You may also use PV-WAVE indirectly, that is, create and run programs, procedures, and functions that are contained in files. This section discusses:

- Entering commands at the command line
- Using a text editor to create and run programs
- Creating and running programs interactively
- Using executive commands
- Using command recall

Entering Commands at the Command Line

When the `WAVE>` prompt is visible, you are located at the PV-WAVE command line. This is also called the main program. The command line gives you immediate access to all the data analysis and graphics display commands and procedures that are part of PV-WAVE. As you enter commands at the keyboard, they are compiled and executed immediately. You see the data transformations and results on your computer screen instantly.

When using the command line, data analysis is quick and simple. Read in the data and, within seconds, you can begin manipulating it, discovering what trends and patterns it holds. Here are some examples of some simple command line entries:

```
WAVE> x = 7*8
```

Assigns the value of 7 times 8 to the variable x.

```
WAVE> PRINT, 'x = ', x
x = 56
```

Prints the string "x = " and the value of x which is 56.

```
WAVE> SET_PLOT, 'PS'
```

This command tells PV-WAVE to use the PostScript driver to produce graphics output for a PostScript printer or plotter.

```
WAVE> .RUN testfile
```

Compiles and runs the file named testfile. If this file is not found in the current directory, the directory search path is examined.

```
WAVE> FOR I = 1,3 DO PRINT, I, I^2
   1 1
   2 4
   3 9
```

You can also enter statements at the prompt. This statement calculates the square of the numbers 1 through 3.

```
WAVE> INFO, /Device
Available graphics_devices: 4510 CGM HPL
NULL PCL PS QMS REGIS SIXEL TEK X

Current graphics device: PS
File: <none>
Mode: Portrait, Non-Encapsulated, Color
   Disabled
 Offset (X,Y): (1.905,12.7) cm., (0.75,5) in.
 Size (X,Y): (17.78,12.7) cm., (7,5) in.
 Scale Factor: 1
 Font Size: 12
```

```
  Font: Helvetica
  # bits per image pixel: 4
```
Displays available graphics output devices, the current graphics device, and the default values for the current graphics device.

```
WAVE> PLOT, mydata
```
Plots a two-dimensional graph of mydata.

Using a Text Editor to Create and Run Programs

You can create program files using a text editor from the operating system prompt and then execute these programs within PV-WAVE. This method is usually how programs are created because these programs can be saved in files for future use. The types of files you can create include:

- ✔ command files
- ✔ procedures
- ✔ functions
- ✔ main programs

Running a Command File

A command file is simply a file that contains PV-WAVE executive commands and statements. Command files are useful for executing commands and procedures that are commonly used. The commands and statements in the command file are executed as if they were entered from the keyboard at the `WAVE>` prompt.

There are three ways that you can run a command file:

- You can enter the command file mode (run a command file) by entering the following at the `WAVE>` prompt:

  `WAVE>` `@`*filename*

- From the UNIX or VMS prompt, you can enter the filename in conjunction with the `wave` command:

  `wave` *filename*

- If you have created a startup file that has been defined with the environment variable, `WAVE_STARTUP`, then you can enter the `wave` command at the UNIX or VMS prompt to run the command file. See *WAVE_STARTUP: Using a Startup Command File* on page 48 for details.

PV-WAVE reads commands from the specified file until the end is reached. You can nest command files by prefacing the name of the new command file with the @ character. The current directory and then all directories in the !Path system variable are searched for the file. See *WAVE_PATH: Setting Up a Search Path* on page 46.

Note A semicolon (`;`) after the `@` character can be interpreted as a VMS filename in a VMS environment. Surround the semicolon within blank spaces or tabs to create a comment after the @ sign.

Command file execution may be terminated before the end of the file with control returning to the interactive mode by calling the STOP procedure from within the command file. Calling the EXIT procedure from the command file has the usual effect of terminating PV-WAVE.

Command File Execution – Sample Usage

An example of a PV-WAVE command line that initiates command file execution is:

```
WAVE> @myfile
```

Use myfile for statement and command input. If not in the current directory, use the search path !Path.

Possible contents of `myfile` are shown below:

```
.RUN PROGA
```

Run program A.

```
.RUN PROGB
```

Run program B.

```
PRINT, avalue, bvalue
```

Print results.

```
CLOSE,3
```

Close file on logical unit 3.

The command file should not contain complete program units such as procedures or functions. However, complete program units can be compiled and run by using the .RUN and .RNEW commands in the command files, as shown in the previous example.

Guidelines for Creating a Command File

Each line of the command file is interpreted exactly as if it was entered from the `WAVE>` prompt. In the command file mode, PV-WAVE compiles and executes each statement before reading the next statement. This is different than the interpretation of programs, procedures, and functions compiled using `.RNEW` or `.RUN`, in which all statements in a program are compiled as a single unit and then executed. Labels, as described in *Statement Labels* on page 52 of the *PV-WAVE Programmer's Guide*, are not allowed in the command file mode because each statement is compiled independently.

Note Multi-line statements must be continued on the next line using the `$` continuation character, because in interactive mode PV-WAVE terminates every statement not ending with `$` by an END statement. A common mistake is to include a block of commands in a FOR loop inside a command file:

```
FOR I = 1,10 DO BEGIN
   PRINT, I, ' square root = ', SQRT(I)
   PRINT, I, ' square = ', I^2
ENDFOR
```

In command file mode (this is not the case for functions and procedures), PV-WAVE compiles and executes each line separately, causing syntax errors in the example above because no matching `ENDFOR` is found on the same line as the `BEGIN` when the line is compiled. The above example can be made to work by inserting an ampersand between each statement in the block of statements and by terminating each line (except the last) with a $:

```
FOR I = 1,10 DO BEGIN & $
   PRINT, I, ' square root = ', SQRT(I) & $
   PRINT, I, ' square = ', I^2 & $
ENDFOR
```

Note Note that the combination of the ampersand and the dollar sign is required. For example, with just an ampersand at the end of each line, the sample program does not run properly because each line is compiled as a separate entity. Hence, a syntax error results when the ENDFOR statement is compiled because it is seen as a single statement that is not connected to a FOR statement. With the dollar sign at the end of each line, no compilation occurs until the ENDFOR statement. For more information on the dollar sign and ampersand characters, see *Special Characters* on page 33.

Creating and Running a Function or Procedure

Using an ordinary text editor, you can create files that define procedures and functions. For example, here's the program listing for a file named `square.pro` that defines a function to square a number:

```
FUNCTION SQUARE, NUMBER
   RETURN, NUMBER^2
END
```

The file automatically compiles and executes when being called at the WAVE> prompt:

```
WAVE> x = SQUARE(24) & PRINT, x
% Compiled module: SQUARE.
   576
```

The file automatically compiles and executes only under the following circumstances:

- if the file is in the !Path or current directory

and

- the filename is the same as the function or procedure name and has a `.pro` extension.

If the file is not the same name as the function, then you must use the .RUN command to compile it. See the section in this chapter, *WAVE_PATH: Setting Up a Search Path* on page 46 for details about search paths.

Note You can also create a function or procedure from the `WAVE>` prompt by using the .RUN command. See *Creating and Running Programs Interactively* on page 25.

Running Existing Functions and Procedures

Besides being able to create and run your own functions and procedures, you can also use a variety of functions and procedures supplied by PV-WAVE. These are known as Standard Library routines. The source code for these can be found in the `wave/lib/std` subdirectory. An important point to remember is that functions and procedures, whether created or supplied, can be used inside of program files as well as at the `WAVE>` prompt.

An example of one useful PV-WAVE function is REBIN. This function resizes a specified array or vector to new dimensions:

```
array2 = REBIN(array1, 512, 512, /Sample)
```

An example of a commonly used PV-WAVE procedure is PLOT:

```
PLOT, X, Y, Color=128
```

Function calls are in the format:

result = *function_name(parameters*)

Procedure calls are in the format:

procedure_name [, *param*$_1$, *param*$_2$]

Keywords

Notice that the examples of the REBIN function and PLOT procedure use the keywords, *Sample* and *Color*. A keyword is normally followed by an equal sign and a value (for example, `Color =128`). Some keywords may be also be specified with the syntax */Keyword*, which is the same as setting the keyword parameter to 1 (for example, `Sample=1`). Many functions and procedures employ keywords. For more information about keywords, see *Procedure and Function Parameters* on page 234 of the *PV-WAVE Programmer's Guide.*

Relationship Between Keywords and System Variables

For some keywords, the default values are derived from PV-WAVE's *system variables.* System variables are a special class of predefined variables available to all applications. All system variables are characterized by an initial exclamation point (!). For example, the keyword *Color* in the example PLOT procedure is based on the system variable !P.Color. The keyword *Color* is used to choose the color. The system variable, !P.Color, contains the default setting for the keyword *Color.* You can change the value of the default:

```
tek_color
```

This command loads 32 colors into the color table.

```
!P.Color = 6
```

Changes the color to purple which is identified by the number 6.

For more information about system variables, see *System Variables* on page 26 of the *PV-WAVE Programmer's Guide.*

Creating and Running Main Programs

A main program is a series of statements that are not preceded by a procedure or function heading (PRO or FUNCTION) and is compiled as a unit. Main programs can also be created interactively as indicated in the next section. Since there is no heading, it cannot be called from other routines, and cannot be passed arguments. When PV-WAVE encounters the END statement in a main program as the result of a .RUN executive command, it compiles it into the special program named `$MAIN$` and immediately executes it as a unit. Afterwards, it can be executed again with the .GO executive command. For example, for a main program file named `testfile` that contains the following statements:

```
FOR I = 3,5 DO BEGIN
 PRINT, 'Square of ', I, ' = ', I^2
 PRINT, 'Square root of ', I, ' = ', SQRT(I)

ENDFOR
END
```

To compile and run this main program file, enter the following at the WAVE> prompt:

```
WAVE> .RUN testfile
```

The results are:

```
Square of 3 = 9
Square root of 3 = 1.73205
Square of 4 = 16
Square root of 4 = 2.00000
Square of 5 = 25
Square root of 5 = 2.23607
```

Note The differences between a main program and a command file are that main programs must have an END statement, must be executed with the .RUN command, and are executed as a unit. Command files do not have an END statement, are executed by typing *@filename*, and are executed one line at a time.

Creating and Running Programs Interactively

At the WAVE> prompt, you can also create and execute programs interactively. However, a program created with the interactive method cannot be saved for use in later PV-WAVE sessions unless you turn on journaling. See *Journaling* on page 36.

To create a program interactively, you use the .RUN command. Here's an example using the .RUN command:

```
WAVE> .RUN
- FOR I = 0,3 DO BEGIN
- PRINT, 'SQRT of ', I
- PRINT, ' = ', SQRT(I)
- ENDFOR
- END
% Compiled module: $MAIN$
SQRT OF 0
= 0.00000
```

```
SQRT OF 1
= 1.00000
SQRT OF 2
= 1.41421
SQRT OF 3
= 1.73205
```

The example program calculates and prints out the square root for the numbers 0 through 3.

After typing `.RUN` and pressing <Return>, a dash (–) prompt is displayed indicating that you are in program mode. When you have completed the program, you must enter `END` as the last line and press <Return>. The message `%Compiled module: $MAIN$` that displays indicates that this is a main program.

Two other types of programs, procedures and functions, can also be created from the `WAVE>` prompt using the .RUN command. Here's an example of how to create a function that squares a number:

```
WAVE> .RUN
- FUNCTION SQUARE, NUMBER
- RETURN, NUMBER^2
- END
%Compiled module: SQUARE
```

After you type `END` and press <Return>, the message, `%Compiled module: SQUARE`, displays. Now, you can use the SQUARE function to calculate the square of a number. Normally, functions and procedures are created in files so that they can be used in future PV-WAVE sessions. However, occasionally you may need to create a short program or function that you do not want to save. The .RUN command provides this option.

Using Executive Commands

PV-WAVE executive commands compile programs, continue stopped programs, and start previously compiled programs. All these commands begin with a period. Under UNIX, file names are case sensitive, while under VMS, either case may be used. Executive commands can be executed from files or from the `WAVE>` prompt.

Executive commands are summarized in the following table:

Table 2-1: Executive Commands

Command	Action
.RUN	Compiles and possibly executes text from files or from the WAVE> prompt.
.RNEW	Erases main program variables and then executes .RUN.
.CON	Continues execution of a stopped program.
.GO	Executes previously compiled main program from the beginning of the program.
.STEP	Executes a single statement. This command may be abbreviated as .S.
.SKIP	Skips over the next statement and then single steps.
.SIZE	Resizes the code area and the data area used to compile programs in terms of bytes.
.LOCALS	Resizes the data area in terms of local variables and common block symbols.

Using .RUN

The `.RUN` command compiles procedures, functions and main programs. The .RUN command also executes main programs. The command may be followed by a list of files to be compiled. Separate the filenames with blanks or commas:

.RUN *file*$_1$, ..., *file*$_n$

If no files are specified with the .RUN command, input is accepted from the keyboard at the `WAVE>` prompt until a complete program unit is entered. The values of all the variables are retained. See *Creating and Running Programs Interactively* on page 25.

Files containing PV-WAVE procedures, programs, and functions are assumed to have the filename extension (suffix) `.pro`. If the filename is the same as the actual function or procedure name, the function or procedure is compiled and executed.

The command arguments `–t` for terminal listing, or `–l` for listing to a named file, may be used after the command name, and before the program file names, to produce a numbered program listing directed to the terminal or to a file. For instance, to see a listing on the screen as a result of compiling a procedure contained in a file named `analyze.pro`:

```
.RUN -t analyze
```

To compile the same procedure and save the listing in a file named: `analyze.lis`:

```
.RUN -l analyze.lis analyze
```

In listings produced by PV-WAVE, the line number of each statement is printed at the left margin. This number is the same as that printed in PV-WAVE error statements, simplifying location of the statement causing the error.

Each level of block nesting is indented four spaces to the right of the preceding block level to improve the legibility of the program's structure.

Using .RNEW

The `.RNEW` command compiles and saves procedures and programs in the same manner as `.RUN`. However, all variables in the main program unit, including those in common blocks, are erased. The `–t` and `–l` switches have the same effect as with `.RUN`. See the examples below. Its syntax is:

.RNEW $file_1$, ..., $file_n$

Sample Usage of .RUN and .RNEW

Some examples of the `.RUN` and `.RNEW` commands are:

`.RUN`

Accept a program from the keyboard (WAVE> prompt). Retain the present variables.

`.RUN myfile`

Compile the file myfile.pro. If myfile.pro is not found in the current directory, PV-WAVE looks for the file in the directory search path.

`.RUN -t a, b, c`

Compiles the files a.pro, b.pro, and c.pro. Lists the programs on the terminal.

`.RNEW -l myfile.lis myfile, yourfile`

Erases all variables. Compiles the files myfile.pro and yourfile.pro. Produces a listing of myfile in the file myfile.lis.

Using .CON

The .CON command continues execution of a program that has stopped because of an error, a STOP statement, or a keyboard interrupt. PV-WAVE saves the location of the beginning of the last statement executed before an error. If it is possible to correct the error condition in the interactive mode, the offending statement may be re-executed by typing `.CON`. After STOP statements, .CON continues execution at the next statement.

Execution of a program interrupted by typing <Control>-C may also be resumed at the point of interruption with the .CON command.

Using .GO

The .GO command starts execution at the beginning of a previously compiled main program.

Using .STEP

The .STEP command executes one or more statements in the current program starting at the current position, stops, and returns control to the interactive mode. This command is useful in debugging programs. If the optional argument *n* is present, it gives the number of statements to execute, otherwise, a single statement is executed. The syntax of the .STEP command is:

.STEP [*n*]

or

.S [*n*]

Using .SKIP

The .SKIP command skips one or more statements and then single steps. This command is useful for continuing over a program statement which caused an error. If the optional argument *n* is present, it gives the number of statements to skip, otherwise, a single statement is skipped. The syntax is:

.SKIP [*n*]

For example, consider the following program segment:

```
... ... ...
   OPENR, 1, 'missing'
   READF, 1, xxx, ..., ...
... ... ...
```

If the OPENR procedure fails because the specified file does not exist, program execution will halt with the OPENR procedure as the current procedure. Execution may not be resumed with the executive command .CON because it attempts to re-execute the offending OPENR procedure, causing the same error.

The remainder of the program can be executed by:

- ❑ Opening the correct file manually by typing in a valid OPENR procedure.

- ❑ Entering `.SKIP`, which skips over the incorrect OPENR procedure.
- ❑ Entering `.CON`, which resumes execution of the program at the READF procedure.

Using .SIZE

The syntax of the .SIZE command is:

.SIZE *code_size data_size*

The .SIZE command resizes the *code area* and *data area*. These memory areas are used when PV-WAVE programs are compiled. The code area holds internal instruction codes that the compiler generates. The data area, also used by the compiler, contains variable name, common block, and keyword information for each compiled function, procedure, and main program.

After successful compilation, a new memory area of the required size is allocated to hold the newly compiled program unit.

By default, the size of the code area is 30,000 bytes. The initial size of the data area is 8,000 bytes (enough space to hold 500 local variables).

Caution Resizing the code and data areas erases the currently compiled main program and all main program variables.

For example, to extend the code and data areas to 40,000 and 10,000 bytes respectively:

```
.SIZE 40000 10000
```

The upper limit for both *code_size* and *data_size* is over 2 billion bytes.

Using .LOCALS

The syntax of the .LOCALS command is:

.LOCALS *local_vars common_symbols*

The .LOCALS command is similar to the .SIZE command, in that it resizes the data area (the data area is described in the previous section, *Using .SIZE*). The .LOCALS command, however, lets you specify the data area size in terms of local variables and common block symbols rather than in bytes. This command affects the size of the data area for the $MAIN$-level (commands entered from the `WAVE>` prompt), and the initial size of the data area for compiled procedures and functions.

The two parameters are positional, but not required. If you execute .LOCALS with no parameters, the data area is set back to its default value, which is 500 local variables. If you want to use 700 variables at the $MAIN$ level, enter:

```
.LOCALS 700
```

.LOCALS clears and frees the current $MAIN$ data area and code area. It then allocates a new code area of the same size as the previous one and a new data area of the specified size.

For compiled procedures and functions, the compiler initially allocates code and data areas of the same size as those that $MAIN$ is currently using. If you get compiler error messages stating that the code and/or data area of a procedure or function is full, you must first make the $MAIN$ code and/or data areas larger with the .SIZE or .LOCALS executive command. Then when you recompile the procedure or function, the compiler starts with the larger code and/or data areas.

See also *Using the ..LOCALS Compiler Directive* on page 244 of the *PV-WAVE Programmer's Guide.*

Using Command Recall

PV-WAVE saves the last 20 command lines you enter. These command lines can be recalled, edited, and re-entered. For example, the up cursor key on the keypad recalls the previous command you entered. Pressing it again recalls the line before that, and so on. When a command is recalled, it is displayed after the PV-WAVE prompt and may be edited or entered as is.

The command recall feature is enabled by setting the system variable !Edit_Input to 1, and is disabled by setting it to 0.

Using Line Editing Keys

The command line editing keys and their functions differ somewhat between UNIX and VMS. To interactively see how function keys are defined, enter:

```
INFO, /Keys
```

For more information, about these keys, see the DEFINE_KEY procedure in Chapter 2, *Function and Procedure Reference*, in the *PV-WAVE Reference.*

Special Characters

This section describes characters with special interpretation and their function in PV-WAVE. Each character is summarized in Table 2-2.

Table 2-2: Special Characters

UNIX	VMS	Both	Function
		!	First character of system variable names. Also precedes font commands.
		'	Delimit string constants or indicate part of octal or hex constant.
		;	Begin comment field.
		$	Continue current command or issue operating system command.
		"	Delimit string constants or precede octal constants.
		.	Indicate constant is floating-point or start executive command.
		&	Separate multiple statements on one line.
		:	End label identifiers.
		*	Array subscript range.
		@	Include file.

Table 2-2: Special Characters

UNIX	VMS	Both	Function
		^C	Interrupt.
^D	^Z		Exit.
^\	^Y		Abort.

- **exclamation point (!)** – Begins the names of PV-WAVE system-defined variables. System variables are predefined variables of a fixed type. Their purpose is to override defaults for system procedures, to return status information, and to control the action of PV-WAVE. For more information about system variables, see *System Variables* on page 26 of the *PV-WAVE Programmer's Guide.*
- **apostrophe (')** – Delimits string literals and indicates part of an octal or hexadecimal constant. See *String Constants* on page 21 of the *PV-WAVE Programmer's Guide* for information about using the apostrophe with strings. For the use of the apostrophe with octal or hexadecimal constants, see *Numeric Constants* on page 18 of the *PV-WAVE Programmer's Guide.*
- **semicolon (;)** – Begins a comment field of a PV-WAVE statement. All text on a line following a semicolon is ignored by PV-WAVE. A line may consist of just a comment or may contain both a valid statement followed by a comment.
- **dollar sign ($)** – At the end of a line indicates that the current statement is continued on the following line. The dollar sign character may appear anywhere a space is legal except within a string constant. Any number of continuation lines are allowed.

  When the `$` character is entered as the first character after the PV-WAVE prompt, the rest of the line is sent to the operating system as a command. To send an operating system command from within a procedure, use the `SPAWN` command. See *Accessing the Operating System Using SPAWN* on page 297 of the *PV-WAVE Programmer's Guide.*

- **quotation mark** (") – The quotation mark precedes octal numbers which are always integers and delimits string constants. Examples: `"100B` is a byte constant equal to 64_{10}, `"Don't drink the water."` is a string constant. See *String Constants* on page 21 of the *PV-WAVE Programmer's Guide* for information about using the quotation mark with strings. For the use of the quotation mark with octal or hexadecimal constants, see *Numeric Constants* on page 18 of the *PV-WAVE Programmer's Guide.*

- **period or decimal point** (.) – Indicates in a numeric constant that the number is of floating-point or double-precision type. Example: `1.0` is a floating-point number.

  Also, in response to the `WAVE>` prompt, the period, if it is the first character on the line, begins an executive command. For example:

  ```
  WAVE> .RUN myfile
  ```

  causes PV-WAVE to compile the file `myfile.pro`. If `myfile.pro` contains a main program, the program will also be executed.

  However, if the period is not the first character on the line as in the following example,

  ```
  WAVE>        .RUN myfile
  ```

  you receive a syntax error.

  Also, the period precedes the name of a tag when referring to a field within a structure. For example, a reference to a tag called `NAME` in a structure stored in the variable `A` is: `A.NAME`. See *Defining and Deleting Structures* on page 102 of the *PV-WAVE Programmer's Guide.*

- **ampersand (&)** – The ampersand separates multiple statements on one line. Statements may be combined until the maximum line length of 132 characters is reached. For example, the following line contains two statements:

  ```
     I = 1 & PRINT, 'VALUE: ', I
  ```

- **colon (:)** – Ends label identifiers. Labels may only be referenced by GOTO, and ON_ERROR statements. The following line contains a statement with the label LOOP1:

  ```
  LOOP1: x = 2.5
  ```

  The colon also separates the starting and ending subscripts in subscript range specifiers. For example A(3:6) designates the fourth, fifth, sixth, and seventh elements of the variable A. For more information on subscript ranges, refer to *Subscript Ranges* on page 84 of the *PV-WAVE Programmer's Guide.* In addition, the colon is used in CASE statements.

- **asterisk (*)** – In addition to denoting multiplication, designates an ending subscript range equal to the size of the dimension. For example, A(3:*) represents all elements of the vector A except the first three elements.

- **"at" sign (@)** – At the beginning of a line causes the PV-WAVE compiler to substitute the contents of the command file whose name appears after @. In addition to searching the current directory for the file, PV-WAVE searches a list of locations where procedures are kept. See *Running a Command File* on page 19.

Journaling

Journaling provides a record of an interactive PV-WAVE session. All text entered at the WAVE> prompt is entered directly into the file, and any text entered from the terminal in response to any other input request (such as with the READ procedure) is recorded as a comment. The result is a file that contains a complete description of the PV-WAVE session which can be rerun later.

The JOURNAL procedure has the form:

JOURNAL [, *param*]

where the string prameter *param* is either a filename (if journaling is not currently in progress), or an expression to be written to the file (if journaling is active).

The first call to JOURNAL starts the logging process. If no parameter is supplied, a journal file named `wavesave.pro` is created. If a filename is specified in *param*, the session's commands will be writting to a file of that name.

Caution Under UNIX, creating a new journal file causes any existing file with the same name to be lost. This includes the default file `wavesave.pro`. Use a filename parameter with the JOURNAL procedure to avoid destroying existing jounral files.

Programmatically Controlling the Journal File

When journaling is not in progress, the value of the system variable !Journal is 0. When the journal file is opened, the value of this system variable is set to the logical unit number of the journal file that is opened. This fact can be used by PV-WAVE routines to check if journaling is active. You can send any arbitrary data to this file using the normal PV-WAVE output routines. In addition, calling JOURNAL with a parameter while journaling is in progress results in the parameter being written to the journal file as if the PRINTF procedure had been used. In other words, the statement:

JOURNAL, *param*

is equivalent to:

PRINTF, !Journal, *param*

with one exception – the JOURNAL procedure is not logged to the file (only its output) while a PRINTF statement is logged to the file in addition to its output.

Journaling ends when the JOURNAL procedure is called again without an argument, or when you exit PV-WAVE.The journal file can be used later as a PV-WAVE command input file to repeat the session, and it can be edited with any text editor if changes are necessary.

JOURNAL Procedure – Sample Usage

As an example, consider the following PV-WAVE statements:

```
JOURNAL, 'demo.pro'
```

Start journaling to file demo.pro

```
PRINT, 'Enter a number: '
```

```
READ, Z
```

Read the user response into variable Z.

```
JOURNAL, '; This was inserted with JOURNAL.'
```

Send a PV-WAVE comment to the journal file using the JOURNAL procedure.

```
PRINTF, !Journal, '; This was inserted ' + $
   'with PRINTF.'
```

Send another comment using PRINTF.

```
JOURNAL
```

End journaling.

If these statements are executed by a user named *bobf* on a Sun workstation named *peanut*, the resulting journal file `demo.pro` will look something like:

```
; SUN WAVE Journal File for bobf@peanut
; Working directory: /home/bobf/wavedemo
; Date: Mon Aug 29 19:38:51 1992
PRINT, 'Enter a number: '
; Enter a number:
READ, Z
; 100
; This was inserted with JOURNAL.
PRINTF, !Journal, '; This was inserted ' +$
'with PRINTF.'
; This was inserted with PRINTF.
```

Note Note that the input data to the READ statement is shown as a comment. In addition, the statement to insert the text using JOURNAL does not appear.

Saving and Restoring Sessions

The SAVE and RESTORE procedures are used together to save the state of variables, system variables, and compiled user-written procedures and functions. The saved session can then be restored at a later time. This ability to "checkpoint" a session and then recover it later can be very convenient. Save files can be used for many purposes:

- Save files can be used to recover variables that are used from session to session. A startup file can be used to execute the RESTORE command every time PV-WAVE is started. See the discussion of startup files in *WAVE_STARTUP: Using a Startup Command File* on page 48 for more details.
- The state of a PV-WAVE session can be saved, then quickly restored to the same point, allowing you to stop working, and then later resume at a convenient time.
- Saved files relieve you of the need to remember the dimensions of arrays and other details. It is very convenient to store images this way. For example, if the three variables `R`, `G`, and `B` hold the colortable vectors, and the variable `Image` holds the image data, the PV-WAVE statement:

  ```
  SAVE, Filename='image.dat', R, G, B, Image
  ```

  saves everything required to display the image properly, in a file named `image.dat`. At a later time, the command:

  ```
  RESTORE, 'image.dat'
  ```

  will restore the four variables from the file.
- Long iterative jobs can save partial results in Save/Restore format to guard against losing data if some unexpected event such as a machine crash were to occur.

Using the SAVE Procedure

You can save user-generated variables, system variables, compiled procedures, and compiled functions for future PV-WAVE sessions.

Saving for Future PV-WAVE Sessions

The SAVE procedure saves variables, system variables, and compiled user-written procedures and functions in a file, using an efficient binary format, for later recovery by RESTORE. It has the form:

SAVE [, var_1, ..., var_n]

where var_n are the named variables to be saved. For information on keyword for the SAVE procedure, refer to Chapter 2, *Function and Procedure Reference* in the *PV-WAVE Reference.*

Caution Under UNIX, creating a new Save file causes any existing file with the same name to be lost. Use the *Filename* keyword with SAVE to avoid destroying desired files. For more information, see Chapter 2, *Function and Procedure Reference* in the *PV-WAVE Reference.*

Using the RESTORE Procedure

The RESTORE procedure restores the objects previously saved in a save file by the SAVE procedure.

RESTORE has the form:

RESTORE [, *filename*]

where *filename* is the name of the save file to be used. If *filename* is not supplied, the filename `wavesave.dat` is used. In addition, you can use keywords with RESTORE. For a description of these keywords, see Chapter 2, *Function and Procedure Reference* in the *PV-WAVE Reference.*

Situations in which the contents of the file will not be restored are:

- When attempting to restore a structure variable, the structure of the saved variable must either not exist, or must agree with the existing structure definition. If the structure is already defined and does not match, RESTORE issues an error message, skips the variable in question, and continues with the next variable in the file. This also applies to system variables.

Caution Visual Numerics, Inc., reserves the right to change the structure of PV-WAVE system variables, although such changes are not anticipated. Generally, there is little need to save system variables, so this restriction is not a problem.

- Read-only system variables are not restored. RESTORE quietly skips over such variables in the file unless the *Verbose* keyword is present. In this case an informative message is issued as the variable is skipped.

Using PV-WAVE in Runtime Mode

PV-WAVE can interpret and execute two kinds of files: source files and compiled files.

- **Source Files** – Functions and procedures saved as regular ASCII files with a `.pro` filename extension. When a function or procedure of this type is called, it is first compiled, then executed by PV-WAVE.
- **Compiled Files** – Functions and procedures that are first compiled in PV-WAVE, then saved with the COMPILE procedure. By default, such files are given a `.cpr` filename extension. Because a file of this type is already compiled, it can be executed more quickly than a `.pro` file.

  For detailed information on the COMPILE procedure, see the *PV-WAVE Reference.*

This ability to handle both source and compiled files allows PV-WAVE to be run in two different modes:

- **Interactive mode** – The mode normally used for PV-WAVE application development and direct access to the PV-WAVE command line.
- **Runtime mode** – Allows direct execution of PV-WAVE applications composed of compiled routines that have been saved with the COMPILE procedure. The runtime mode is described in the following sections.

Starting PV-WAVE in Runtime Mode

In runtime mode, you can run a PV-WAVE application directly from the operating system prompt. When the application is finished running, control returns to the operating system level.

The application must first be compiled in PV-WAVE and saved with the COMPILE procedure. For example, if the procedure called `images` is compiled in PV-WAVE, the command:

```
COMPILE, 'images'
```

saves a file containing the compiled procedure. By default, this file is called `images.cpr`, and it is saved in the current working directory. For detailed information on the COMPILE procedure, see the *PV-WAVE Reference*.

To execute the compiled, saved application called `images.cpr` from the operating system prompt, enter the following command, where `-r` is a flag that specifies runtime mode:

```
wave -r images
```

When the application is finished running, control is returned to the operating system prompt. Note that the `.cpr` extension is not used.

You can set the default mode to "runtime" with the environment variable `WAVE_FEATURE_TYPE` by typing on a UNIX system:

```
setenv WAVE_FEATURE_TYPE RT
```

On a VMS system, enter:

```
DEFINE WAVE_FEATURE_TYPE RT
```

Now, the `-r` flag is not needed, and you can run the application by entering:

```
wave images
```

The read-only system variable !Feature_Type allows you to distinguish between runtime mode and normal, interactive mode. This system variable simply reflects the current setting of the `WAVE_FEATURE_TYPE` environment variable (UNIX) or logical (VMS).

More than one saved compiled file can be executed at a time from the operating system prompt. Just separate the application filenames with spaces, as follows:

```
wave file_1 file_2 file_3 ...
```

The Search Path for Compiled Routine Files

Whenever a user-written procedure or function is called, PV-WAVE searches *first* for saved, compiled files (`.cpr` files) with the same name as the called routine. If a saved, compiled file is not found, PV-WAVE searches for a source file (`.pro` file). PV-WAVE searches the current directory and all directories specified in the !Path directory path.

Note If you place a `.pro` file in the current working directory that has the same name as a `.cpr` file further along the directory path, the `.cpr` file will always be found and executed first. To explicitly execute the `.pro` file, compile it with the .RUN command or remove the `.cpr` file from the !Path directory path.

Note The compiled (`.cpr`) file must have the same name as the called routine. If the calling name of an application program is `images`, then the saved, compiled file must be called `images.cpr`.

Developing Runtime Applications

Applications developed for operation in PV-WAVE's runtime mode must adhere to the following guidelines:

- Only PV-WAVE routines that are compiled and saved with the COMPILE command can be executed in runtime mode.

- The startup file pointed to by the `WAVE_RT_STARTUP` environment variable (UNIX) or logical (VMS) must be compiled and saved with the COMPILE command. The startup file must be in a directory pointed to by the `WAVE_PATH` environment variable (UNIX) or logical (VMS). For more information on this startup file, see *WAVE_RT_STARTUP: Using a Startup Procedure in Runtime Mode* on page 51.
- PV-WAVE executive commands .RUN, .RNEW, .GO, .STEP, .SKIP, .CON, and the STOP routine are not recognized in runtime mode.
- PV-WAVE breakpoints specified with the BREAKPOINT procedure are not recognized in runtime mode.
- Any errors that occur in runtime mode are reported as usual, and control is returned to the operating system prompt.

Modifying Your PV-WAVE Environment

Under UNIX, PV-WAVE uses environment variables to determine its initial state. Under VMS, logical names are used for the same purpose. In either case the names and functions are the same. This section explains how to modify or customize environment variables and logicals.

Note Normally you do not need to alter your environment. If PV-WAVE is installed properly, your environment will be already set up. The information in this section applies only if you wish to modify or customize your environment.

WAVE_DEVICE: Defining Your Terminal or Window System

In order to function properly, PV-WAVE must know the type of terminal or window system you wish to use. By default, it assumes X, the X Window System. If you wish, this default can be changed, as described below.

Changing the Default Device on a UNIX System

PV-WAVE reads the value of the environment variable `WAVE_DEVICE` when it starts. If `WAVE_DEVICE` is defined, PV-WAVE calls the procedure SET_PLOT with this string. For example, to use PV-WAVE with Tektronix terminals, include the following command in your `.login` (or `.profile`) file:

```
setenv WAVE_DEVICE tek
```

The device name can be entered in either upper or lower case. If `WAVE_DEVICE` is defined, it must contain the name of a valid PV-WAVE graphics device. See the description of SET_PLOT in *Selecting the Output Device with SET_PLOT* on page A-3 of this manual for a complete list of device names.

Changing the Default Device on a VMS System

PV-WAVE reads the value of the logical name `WAVE_DEVICE` when it starts. If `WAVE_DEVICE` is present, PV-WAVE calls the procedure `SET_PLOT` with this string. For example, to use PV-WAVE with Tektronix terminals, include the following command in your `LOGIN.COM` file:

```
$ DEFINE WAVE_DEVICE tek
```

WAVE_DIR: Ensuring Access to Required Files

`WAVE_DIR` is the root of the PV-WAVE directory structure. This environment variable is defined in `wvsetup`. All PV-WAVE files are located in subdirectories of `WAVE_DIR`.

Setting WAVE_DIR on a UNIX System

The `WAVE_DIR` environment variable must be correctly defined in order for PV-WAVE to run properly. If `WAVE_DIR` is not defined, PV-WAVE assumes a default of `/usr/local/lib/wave`.

WAVE_DIR is defined in the wvsetup file. To make sure that you have WAVE_DIR properly defined, enter the following command at the UNIX prompt:

```
source <maindir>/wave/bin/wvsetup
```

Setting WAVE_DIR on a VMS System

The WAVE_DIR logical must be correctly defined in order for PV-WAVE to run properly. For example, if the PV-WAVE distribution is located in DUA1:[WAVE] on your system, enter the following DCL command:

```
$ DEFINE WAVE_DIR DUA1:[WAVE.]/trans= $
   (conceal, term)
```

Note WAVE_DIR must be defined using the physical device name of the disk. Most sites use logical names to refer to disks. If you wish to define WAVE_DIR in terms of the disk's logical name, use the DCL F$TRNLNM lexical function to translate the name.

For example, if the main PV-WAVE directory is DISKA:[WAVE]:

```
$ DEFINE WAVE_DIR 'F$TRNLNM(""DISKA"")' $
   [WAVE.]/trans=(conceal, term)
```

WAVE_PATH: Setting Up a Search Path

WAVE_PATH sets the function and procedure library directory search path. This environment variable is also defined in wvsetup. The search path is a list of locations to search if the procedure or function is not found in the current directory. The current directory is always searched first. PV-WAVE then looks for the function or procedure in the locations specified by the PV-WAVE system variable !Path. The details of how !Path is initialized differ between UNIX and VMS, although the overall concept is similar. For more information on system variables, see *System Variables* on page 26 of the *PV-WAVE Programmer's Guide.*

Setting Up WAVE_PATH on a UNIX System

The environment variable, WAVE_PATH is a colon-separated list of directories. If WAVE_PATH is not defined, a default of /usr/local/lib/wave/lib is assumed. This is consistent with the default value assumed for WAVE_DIR as defined in the wvsetup file. Each user may add directories to WAVE_PATH that contain PV-WAVE programs, procedures, functions, and "include" files. You may find it convenient to add to the value that is already defined in your wvsetup file. For example:

```
setenv WAVE_PATH $WAVE_PATH":"/user/mylib
```

This command adds the directory /user/mylib to the existing variable WAVE_PATH.

!Path is a colon-separated list of directories, similar to the PATH environment variable that UNIX uses to locate commands. When PV-WAVE starts, !Path is initialized from the environment variable WAVE_PATH. The value of !Path may be changed once you are running PV-WAVE. For example, the following command adds the directory /usr2/project/wave_files to the beginning of the search path:

```
WAVE> !Path = '/usr2/home/wave_files:' + !Path
```

Setting Up WAVE_PATH on a VMS System

WAVE_PATH is comma-separated list of directories and library text files. Text libraries are distinguished by prepending an "@" character to their name. If WAVE_PATH is not defined, a default of @WAVE_DIR:[LIB]USERLIB is assumed. Each user may assign WAVE_PATH to a unique combination of directories and text libraries that contain PV-WAVE programs, procedures, functions, and "include" files. You may find it convenient to set up this variable in your LOGIN.COM file. For example:

```
$ DEFINE WAVE_PATH $
   "DISKA:[USER.WAVELIB], $
   @WAVE_DIR:[LIB]USERLIB.TLB"
```

causes PV-WAVE to search for programs first in the current directory, then in the directory DISKA:[USER.WAVELIB], and finally in the PV-WAVE Standard library, which is supplied by Visual Numerics, Inc., as a VMS text library. For more information on VMS text libraries, see *VMS Procedure Libraries* on page 251 of the *PV-WAVE Programmer's Guide.*

WAVE_PATH can also be defined as a multi-valued logical name (for example a search list logical). Therefore, the above example can also be written as:

```
$ DEFINE WAVE_PATH DISKA:[USER.WAVELIB], $
   "@WAVE_DIR:[LIB]USERLIB.TLB"
```

PV-WAVE simply takes the various translations and concatenates them together into a comma separated list. Note that the quotes around the second translation in this example are necessary to keep DCL from seeing the "@" character as an invitation to execute a command file.

Under VMS, !Path is a comma-separated list of directories and text libraries. Text libraries are distinguished by prepending an "@" character to their name. When PV-WAVE starts, !Path is initialized from the logical name WAVE_PATH. The value may be changed once you are running PV-WAVE. For example, the following command adds the DISKA:[PROJECTLIB] directory to the beginning of the search path:

```
WAVE> !Path = 'DISKA:[PROJECTLIB],' + !Path
```

WAVE_STARTUP: Using a Startup Command File

WAVE_STARTUP points to the name of a command file that is executed by PV-WAVE on initialization. The startup file contains a series of PV-WAVE statements and is executed each time PV-WAVE is started. Common uses are to compile frequently-used procedures or functions, to load data, and to perform other useful operations. It contains PV-WAVE statements which are individually compiled and executed, in the same manner as command file execution. For more information on command files, see *Running a Command File* on page 19.

The default startup file for UNIX is called wavestartup and is located in *<path>*/wave/bin. For VMS the default file is wavestartup.dat and is located in:

```
WAVE_DIR:[000000.BIN]
```

The wavestartup file turns off the compiler messages, sets up the WAVE> prompt, and then calls the Standard library routine setdemo.pro. This routine sets up the default key bindings for the function keys and displays their definitions upon entering PV-WAVE. For more information about setdemo.pro, see Chapter 2, *Function and Procedure Reference*, in the *PV-WAVE Reference*.

Using a Startup File Under UNIX

To use a PV-WAVE startup file under UNIX, set the environment variable WAVE_STARTUP to the name of the file to be executed. For example, assume the startup file named startfile contains the following statements:

```
.RUN add.pro
.RUN square.pro
INFO
```

Set the environment variable with the setenv command:

```
setenv WAVE_STARTUP startfile
```

When you start PV-WAVE by entering wave at the UNIX prompt, you get the following display:

```
PV-WAVE. Version ...
.
.
% Compiled module: ADD.
% Compiled module: SQUARE.
% At $MAIN$ .
Code area used: 0.00% (0/16384), Data area
   used: 0.05% (2/4096)
# local variables: 0, # parameters: 0
```

```
Saved Procedures:
 ADD
Saved Functions:
 SQUARE
WAVE>
```

The startup file compiles the ADD procedure and the SQUARE function, and displays general information about the current status of PV-WAVE before displaying the WAVE> prompt.

Using a Startup File Under VMS

To use a PV-WAVE startup file under VMS, assign the VMS logical name WAVE_STARTUP to the name of the file to be executed.

The procedure search path, !Path, is used to search for the file if it is not in the current directory.

To define a startup file named startfile, enter:

```
DEFINE WAVE_STARTUP startfile
```

When you enter wave at the operating system prompt, the file is executed.

WAVE_FEATURE_TYPE: Setting the Default Operating Mode

The environment variable WAVE_FEATURE_TYPE lets you set the default operating mode of PV-WAVE to "runtime". When this environment variable is set to RT, compiled PV-WAVE applications can be executed directly from the operating system prompt without using the -r option. For example:

```
% setenv WAVE_FEATURE_TYPE RT
```

Set the environment variable.

```
% wave appname
```

Run a compiled, saved PV-WAVE application called appname.

WAVE_RT_STARTUP: Using a Startup Procedure in Runtime Mode

`WAVE_RT_STARTUP` points to the name of a compiled procedure file that is executed when PV-WAVE initializes in runtime mode. The startup file may contain PV-WAVE routines that are executed each time PV-WAVE is started in runtime mode. For more information on saving and using compiled routines, see *Using PV-WAVE in Runtime Mode* on page 41.

On a UNIX system, the default startup file for runtime mode is:

```
$WAVE_DIR/lib/std/rtwavestartup.cpr
```

On a VMS system, the default startup file for runtime mode is:

```
WAVE_DIR:[000000.LIB.STD]RTWAVESTARTUP.CPR
```

WAVE_GUI: Selecting the GUI on Sun Workstations

`WAVE_GUI` allows you to explicitly set the look-and-feel of Graphical User Interface (GUI) used for WAVE Widgets and Widget Toolbox applications running on a Sun workstation. On a Sun workstation, the choices are either Motif or OPEN WINDOWS. On all other types of workstations, Motif is the only type of GUI available, and thus setting this environment variable has no effect on those platforms.

If you are running on a Sun workstation, you can set the GUI to OPEN WINDOWS (the default) by entering the following lines at the operating system prompt:

```
% setenv WAVE_GUI OLIT

% source $WAVE_DIR/bin/wvsetup
```

You can set the GUI to Motif by entering:

```
% setenv WAVE_GUI Motif

% source $WAVE_DIR/bin/wvsetup
```

Note that you *must* source the `wvsetup` file *after* setting the `WAVE_GUI` environment variable. Otherwise, the change will have no effect in the subsequent PV-WAVE session.

Changing the PV-WAVE Prompt

The text string PV-WAVE uses to prompt you for input is specified by the system variable !Prompt. You can change the prompt by setting this system variable to the new prompt string. The prompt is currently defined in the file `wavestartup` for UNIX and `WAVESTARTUP.DAT` for VMS.

Here's an example showing how to tailor your prompt to display text:

```
!Prompt = 'Hello World!> '
```

Here's another example making PV-WAVE ring the bell on the terminal without echoing visible text when prompting:

```
!Prompt = '\007'
```

(The ASCII code for the bell is 7.) It does not have a printable representation, so it is specified using the octal escape sequence `\007`. Such sequences are described in *Representing Non-printable Characters* on page 22 of the *PV-WAVE Programmer's Guide.*

Tip You can also place a prompt definition in your `WAVE_STARTUP` file, as described on *WAVE_STARTUP: Using a Startup Command File* on page 48.

For an alternate way to modify the prompt, see the PROMPT procedure in the *PV-WAVE Reference.*

Defining Keyboard Shortcuts

Function keys may be equated to a character string using the DEFINE_KEY procedure. For example, the `<R4>` key on a Sun-style keyboard or the `<PF4>` key on a DEC keyboard, can be equated to the string `PLOT`, as shown in the example below. This allows frequently used strings and commands to be entered with a single key stroke.

For detailed information on DEFINE_KEY, see the description in the *PV-WAVE Reference.*

The PV-WAVE command `INFO, /Keys` displays the current definition of all the function keys.

Tip A natural place to put your key definitions is in the startup file so that the function keys are defined when PV-WAVE is initialized.The defaults for the key definitions are set up with the `setdemo.pro` procedure in the `wavestartup` file. See *WAVE_STARTUP: Using a Startup Command File* on page 48.

Using PV-WAVE with X Windows

The interface to the X Windows system is described in detail in the Appendix A, *Output Devices and Window Systems.* This section explains briefly how to set up the X Windows system to work with PV-WAVE.

Note The X Window System is the default windowing system for all available PV-WAVE platforms.

If You Are Running Under X Windows

Little or no customizing is required to use PV-WAVE with the X Windows system. You can control the number of colors used by PV-WAVE, if and how windows are repainted, and the type of color system (visual class).

Be sure that your system is properly set up to display X graphics. For UNIX systems under the C shell, you may need to enter:

```
% setenv DISPLAY hostname:0.0
% xhost hostname
```

For VMS systems, you may need to enter:

```
    $ SET DISPLAY /CREATE /NODE=nodename -
        /SCREEN=0.0 /TRANSPORT=transport_type
```

where *hostname* or *nodename* is the name of the system on which you want graphics to be displayed.

3

CHAPTER

Displaying 2D Data

PV-WAVE provides routines for plotting data in a variety of ways. These routines allow general X versus Y plots, contouring, mesh surface plots, and perspective plotting, in an extremely flexible manner without requiring you to write complicated programs. These plotting and graphic routines are designed to allow easy visualization of data during data analysis.

Optional keyword parameters and system variables allow straightforward customization of the appearance of the results: (i.e., specification of scaling, axis style, colors, etc.).

This chapter contains numerous examples of scientific graphics in which one variable is plotted as a function of another. The procedures that display three-dimensional data, CONTOUR and SURFACE, are explained in detail in Chapter 4, *Displaying 3D Data*. Procedures used to display and process images are discussed in Chapter 5, *Displaying Images*.

Summary of 2D Plotting and General Graphics Routines

This section lists the 2D plotting procedures that are described in this chapter. In addition, related graphics procedures that are often used with the plotting procedures are listed.

Plotting Routines For Two-Dimensional Data

AXIS [, *x*, *y*, *z*]

Draw an annotated axis.

OPLOT, *x_array*, *y_array*

Overplot array over old axis.

PLOT, *x_array*, *y_array*

Plot array on new axis, set scaling.

PLOT_IO, *x_array*, *y_array*

Plot with linear-log scaling.

PLOT_OI, *x_array*, *y_array*

Plot with log-linear scaling.

PLOT_OO, *x_array*, *y_array*

Plot with log-log scaling.

General Graphics Routines

CURSOR, *x_pos*, *y_pos*

Read graphics cursor.

DEVICE

Control device-specific functions.

ERASE

Erase the screen.

PLOTS, *x*, *y* [, *z*]

Plot lines and points.

POLYFILL, *x*, *y* [, *z*]

Fill irregular polygons with color or pattern.

POLYSHADE, *vertices*, *polygons*

Draw shaded surface representation of solids.

SET_PLOT, *device_name*

Set graphics output device.

TVCRS, *x_pos, y_pos*
: Set cursor position and visibility.

USERSYM, *x, y, z*
: Define a symbol for plotting.

XYOUTS, *x, y, string*
: Display text at a specified location.

Customizing Plots with Keyword Parameters

The PV-WAVE plotting procedures are designed to produce acceptable results for most applications with a minimum amount of effort. The plotting and graphics keyword parameters and system variables, which are described in Chapter 4 *System Variables*, in the *PV-WAVE Reference*, allow you to customize your graphics output. Examples in this chapter show how to use many of the major keywords and system variables used to modify 2D graphics.

Keyword Correspondence with System Variables

Many of the plotting keyword parameters correspond directly to fields in the system variables !P, !X, !Y, !Z, or !PDT. When you specify a keyword parameter name and value in a call, that value affects only the current call — the corresponding system variable field is not changed. Changing the value of a system variable field changes the default for that particular parameter and remains in effect until explicitly changed. The system variables and the corresponding keywords that are used to modify graphics are described in Chapter 3 *Graphics and Plotting Keywords*, in the *PV-WAVE Reference*, and in Chapter 4 *System Variables*, in the *PV-WAVE Reference*.

Example of Changing the Default Color Index

The color index controls the color of text, lines, axes, and data in 2D plots. By default, the color index is set in the !P.Color field of the !P system variable. This default value is normally set to the number of available colors minus 1. (If your system supports 256 colors, !P.Color is set to 255 by default.)

Using the Color Keyword Parameter

You can override this default value at any time by including the *Color* keyword in the graphics routine call. For example, to set the color of a plot to color index 12, enter:

```
PLOT, X, Y, Color = 12
```

Because keyword parameters only modify the current function or procedure call, future plots are not affected.

Changing the !P.Color System Variable

To change the color for *all* plots produced during the current session, you can modify !P.Color. For example, to change the default color index to 12, enter:

```
!P.Color = 12
```

Interpretation of the Color Index

The interpretation of the color index varies among the devices supported by PV-WAVE. With color video displays, this index selects a color (normally an RGB triple) stored in a device table. You can control the color selected by each color index with the TVLCT procedure which loads the device color tables. TVLCT is described in the *PV-WAVE Reference.*

Other devices have a fixed color associated with each color index. With plotters, for example, the correspondence between colors and color index is established by the order of the pens in the carousel.

Three Graphics Coordinate Systems

You may specify coordinates to PV-WAVE in data, device, or normal coordinate systems. These systems are explained in the following sections.

Almost all the PV-WAVE graphics procedures will accept parameters in any of these coordinate systems. Most procedures use the data coordinate system by default. Routines beginning with the letters TV are notable exceptions. They use device coordinates by default. You can explicitly specify the coordinate system by including one of the keyword parameters *Data*, *Device*, or *Normal* in the call. For example:

```
PLOT, x, y, /Normal
```

Data Coordinate System

The data coordinate system is the coordinate system established by the most recent PLOT, CONTOUR, or SURFACE procedure. This system usually spans the plot window, the area bounded by the plot axes, with a range identical to the range of the plotted data. The system may have two or three dimensions, and may be linear, logarithmic, or semi-logarithmic.

Note Data is the default coordinate system for most graphics procedures.

Device Coordinate System

The device coordinate system is the physical coordinate system of the selected plotting device. Device coordinates are integers, ranging from (0,0) at the bottom-left corner, to ($V_x - 1$, $V_y - 1$) at the upper-right corner. V_x and V_y are the number of columns and rows addressable by the device.

Normal Coordinate System

The normalized coordinate system ranges from (0.0, 0.0) to (1.0, 1.0) over the three axes.

Coordinate System Conversion

This section describes how PV-WAVE converts from one coordinate system to another.

The system variables !D, !P, !X, !Y, and !Z contain the information necessary to convert from one coordinate system to another. The relevant fields of these system variables are explained below, and formulas are given for conversions to and from each coordinate system. Three-dimensional coordinates are discussed in Chapter 4, *Displaying 3D Data.*

In the following discussion, *D* is a data coordinate, *N* is a normalized coordinate, and *R* is a raw device coordinate.

The fields !D.X_VSize and !D.Y_VSize always contain the size of the visible area of the currently selected display or drawing surface. Let V_x and V_y represent these two sizes.

The field !X.S, is a two-element array that contains the parameters of the linear equation converting data coordinates to normalized coordinates. !X.S(0) is the intercept, and !X.S(1) is the slope. !X.Type is 0 for a linear X axis, and is 1 for a logarithmic X axis. The Y and Z axes are handled in the same manner, using the system variables !Y and !Z.

With the above variables defined, the two-dimensional coordinate conversions for the X coordinate may be written as follows:

D_x = Data coordinate

N_x = Normalized coordinate

R_x = Device coordinate

V_x = Device X size, in device coordinates

X_i = !X.S(i), scaling parameter

$$\left\{ X_0 + X_1 \log_{10} D_x \qquad \text{logarithmic} \right.$$

$$\text{Data to Device } R_x = \begin{cases} V_x(X_0 + X_1 D_x) & \text{linear} \\ V_x(X_0 + X_1 \log_{10} D_x) & \text{logarithmic} \end{cases}$$

Normal to Device $R_x = N_x V_x$

$$\text{Normal to Data } D_x = \begin{cases} (N_x - X_0) / X_1 & \text{linear} \\ 10^{(N_x - X_0) / X_1} & \text{logarithmic} \end{cases}$$

$$\text{Device to Data } D_x = \begin{cases} (R_x/V_x - X_0)/X_1 & \text{linear} \\ 10^{(R_x/V_x - X_0)/X_1} & \text{logarithmic} \end{cases}$$

Device to Normal $N_x = R_x / V_x$

The Y and Z axis coordinates are converted in exactly the same manner, with the exception that there is no Z device coordinate and logarithmic Z axes are not permitted.

Drawing X Versus Y Plots

This section illustrates the use of the basic X versus Y plotting routines, PLOT and OPLOT.

The PLOT procedure produces linear-linear plots. The procedures PLOT_IO, PLOT_OI, and PLOT_OO are identical to PLOT, except they produce linear-log, log-linear, and log-log plots, respectively.

Data from the U.S. Scholastic Aptitude Test (SAT), from the years 1967, 1970, and from 1975 to 1983, are used in the following examples.

Note Variables defined in the following examples are used in later examples in this chapter.

Producing a Basic XY Plot

The following PV-WAVE statements create and initialize the variables VERBM, VERBF, MATHM, and MATHF, which contain the verbal and math scores for males and females for the 11 observations:

```
VERBM = [463, 459, 437, 433, 431, 433, $
   431, 428, 430, 431, 430]

VERBF = [468, 461, 431, 430, 427, 425, $
   423, 420, 418, 421, 420]

MATHM = [514, 509, 495, 497, 497, 494, $
   493, 491, 492, 493, 493]

MATHF = [467, 465, 449, 446, 445, 444, $
   443, 443, 443, 443, 445]
```

A vector in which each element contains the year of the score is constructed with the statement:

```
YEAR = [1967, 1970, INDGEN(9) + 1975]
```

The PLOT procedure, which produces an X versus Y plot on a new set of axes, requires one or two parameters: a vector of Y values, or a vector of X values followed by a vector of Y values. Figure 3-1 was produced by the statement:

```
PLOT, YEAR, VERBM
```

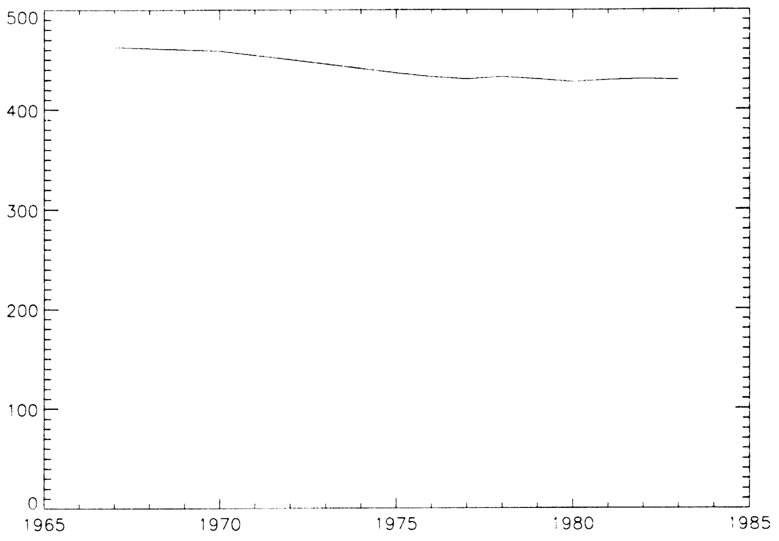


Figure 3-1 Initial 2D plot.

Note You can abort any of the higher-level graphics procedures (e.g., PLOT, OPLOT, CONTOUR, and SURFACE) by typing Control-C.

Scaling the Plot Axes and Adding Titles

The fluctuations in the data are hard to see because the scores range from 428 to 463, and the plot's Y axis is scaled from 0 to 500. Two factors cause this effect. By default, PV-WAVE sets the minimum Y axis value of linear plots to 0 if the Y data are all positive. The maximum axis value is automatically set by PV-WAVE from the maximum Y data value. In addition, PV-WAVE attempts to produce from 3 to 6 tick mark intervals that are in increments of an integer power of 10 times 2, 2.5, 5, or 10. In this example, this rounding effect causes the maximum axis value to be 500, rather than 463.

Using YNozero to Scale the Y Axis

The *YNozero* keyword parameter inhibits setting the Y axis minimum to 0 when given positive, non-zero data. Figure 3-2 illustrates the data plotted using this keyword. The Y axis now ranges from 420 to 480, because PV-WAVE selected 3 tick mark intervals of 20.

You can make `/Ynozero` the default in subsequent plots by setting bit 4 of !Y.Style to 1, (`!Y.Style = 16`).

Other bits in the Style field of the axis system variables !X, !Y, and !Z are described in the Chapter 4 *System Variables*, in the *PV-WAVE Reference*. Briefly: Other bits in the Style field extend the axes, (providing a margin around the data), suppress the axis and its notation, and suppress the box-style axes by drawing only a left and bottom axis.

Adding Titles

The *Title*, *XTitle*, and *YTitle* keywords are used to produce axis titles and a main title in the plot shown in Figure 3-2. This figure was produced with the statement:

```
PLOT, YEAR, VERBM, /YNozero, $
   Title = 'Verbal SAT, Male', $
   Xtitle = 'Year', Ytitle = 'Score'
```

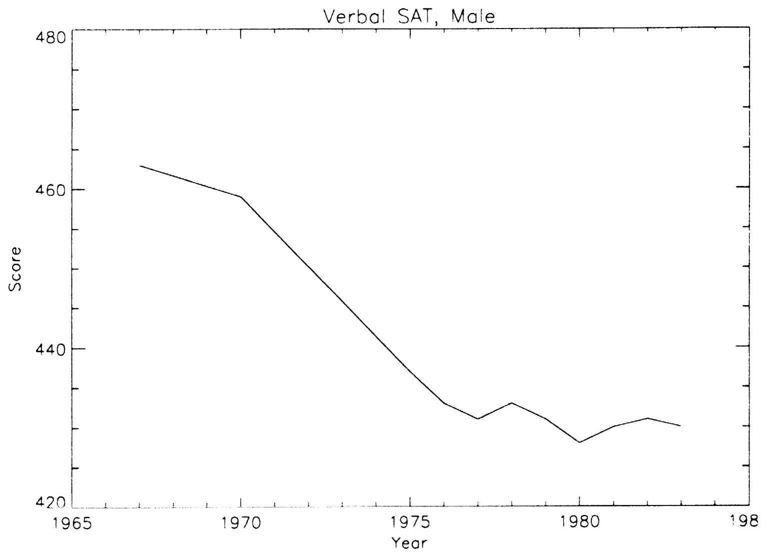


Figure 3-2 Properly scaled plot with added title annotation

Specifying the Range of the Axes

The range of the X, Y, or Z axes can be explicitly specified with the *XRange*, *YRange*, and *ZRange* keyword parameters. The argument of the keyword parameter is a two-element vector containing the minimum and maximum axis values.

For example, if we wish to constrain the X axis to the years 1975 to 1983, the following keyword parameter is included in the call to PLOT:

```
XRange = [1975, 1983]
```

Note The effect of the *Ynozero* keyword, explained in the previous section, is identical to that obtained by specifying the following *YRange* keyword parameter in the call to PLOT:

```
YRange = [MIN(Y), MAX(Y)]
```

Specifying Exact Tick Intervals with YStyle = 1

As explained in the previous section, PV-WAVE attempts to produce even tick intervals, and the axis range selected by PV-WAVE may be slightly larger than that given with the *XRange*, *YRange*, and *ZRange* keywords. To obtain the exact specified interval, set the X axis style parameter to 1 (`XStyle = 1`).

The call combining all these options is:

```
PLOT, YEAR, VERBM, /Ynozero, $
   Title = 'Verbal SAT, Male', $
   Xtitle = 'Year', Ytitle = 'Score', $
   Xrange = [1975, 1983], /Xstyle
```

Figure 3-3 illustrates the result.

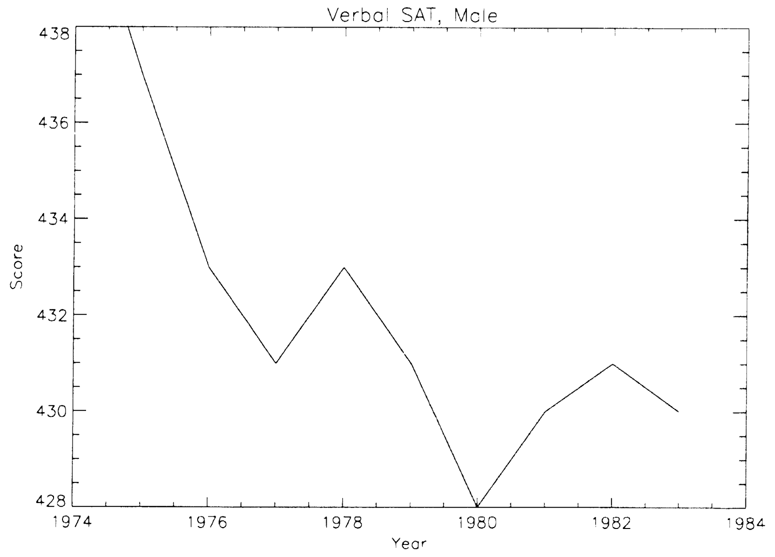


Figure 3-3 Plot with X axis range of 1975 – 1983.

Plotting Additional Data on the Same Axes

Additional data may be added to existing plots with the OPLOT procedure. Each call to PLOT establishes the plot window (the region of the display enclosed by the axes), the axis types (linear or log), and the scaling. This information is saved in the system variables !P, !X, and !Y, and used by subsequent calls to OPLOT.

It may be useful to change the color index, linestyle, or line thickness parameters in each call to OPLOT to distinguish the data sets. For a table describing the linestyle associated with each index, see the description of the !P.Linestyle system variable in Chapter 4 *System Variables*, in the *PV-WAVE Reference*.

Figure 3-4 illustrates a plot showing all four data sets, `VERBF`, `VERBM`, `MATHF`, and `MATHM`. Each data set except the first was plotted with a different line style and was produced by a call to OPLOT.

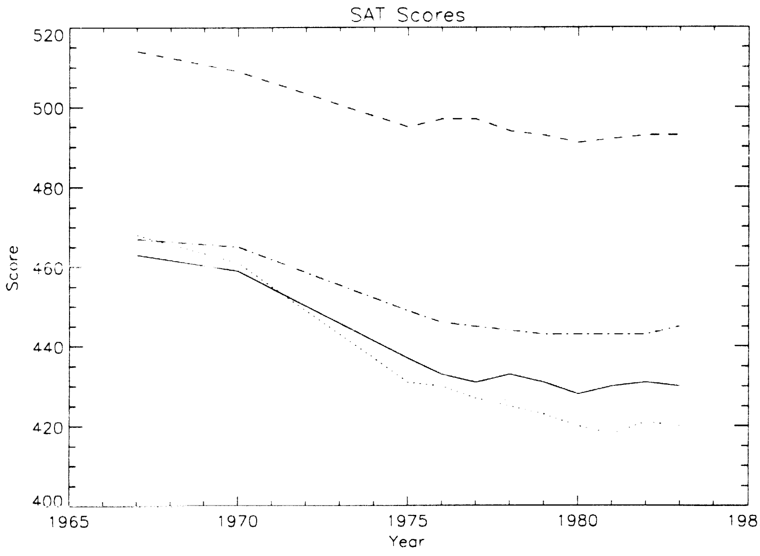


Figure 3-4 Overplotting using different line styles.

In this example, an 11-by-4 array called `allpts` is defined which contains all the scores for the four categories using the array concatenation operator. Once this array is defined, the PV-WAVE array operators and functions can be applied to the entire data set, rather than explicitly referencing the particular score.

Figure 3-4 was produced with the statements:

```
allpts = [[verbf], [verbm], [mathf], [mathm]]
```

Make an (n, 4) array containing the four score vectors.

```
PLOT, year, verbf, Yrange=[MIN(allpts), $
   MAX(allpts)]
```

Plot 1st graph. Set the Y axis min and max from the min and max of all data sets. Default line style is 0. The title keywords have been omitted from this example for clarity.

```
FOR i=1, 3 do OPLOT, year, allpts(*, i), $
   Line = i
```

Loop for the three remaining scores, varying the line style.

Plotting Date/Time Axes

Using PV-WAVE's Date/Time functions, you can create Date/Time variables and automatically plot multiple Date/Time axes. For detailed information on manipulating and plotting Date/Time data, see Chapter 7, *Working with Date/Time Data.*

Annotating Plots

An obvious problem with Figure 3-4 is that it lacks labels describing the different lines shown. To annotate a plot, select an appropriate font and then use the XYOUTS procedure.

Selecting Fonts

You can use software or hardware generated fonts to annotate plots. Chapter 9, *Software Fonts* explains the difference between these types of fonts and the advantages and disadvantages of each.

The annotation in Figure 3-5 uses the PostScript Times-Roman font. This was selected by first setting the default font, !P.Font, to the hardware font index of 0, and then calling the DEVICE procedure to set the Times-Roman font:

```
!P.Font = 0
SET_PLOT, 'ps'
DEVICE, /Times
```

Other PostScript fonts and their bold, italic, oblique and other variants are described in *Using PostScript Fonts* on page A-37.

Using XYOUTS to Annotate Plots

You can add labels an other annotation to your plots with the XYOUTS procedure. The XYOUTS procedure is used to write graphic text at a given location.

The basic call to XYOUTS to write a string starting at location (X, Y) is:

XYOUTS, *x*, *y*, *'string'*

For a detailed description of XYOUTS and its keywords, see the *PV-WAVE Reference.* For other tips on using XYOUTS, see *Getting Input from the Cursor* on page 90.

Figure 3-5 illustrates one method of annotating each graph with its name. The plot was produced exactly as was Figure 3-4, with the exception that the X axis range was extended to the year 1990 to allow room for the titles. To accomplish this, the keyword parameter `XRange = [1967, 1990]` was added to the call to PLOT. A string vector, `NAMES`, containing the names of each score is also defined. As noted in the previous section, the PostScript Times-Roman font was selected for this example.

The annotation in Figure 3-5 was produced using the statements:

```
names = ['Female Verbal', 'Male Verbal', $
   'Female Math', 'Male Math']
```

Vector containing the name of each score.

```
nl = N_ELEMENTS(year) - 1
```

Index of last point.

```
FOR i=0,3 do XYOUTS, 1984, allpts(nl,i), $
   names(i)
```

Append the title of each graph on the right.

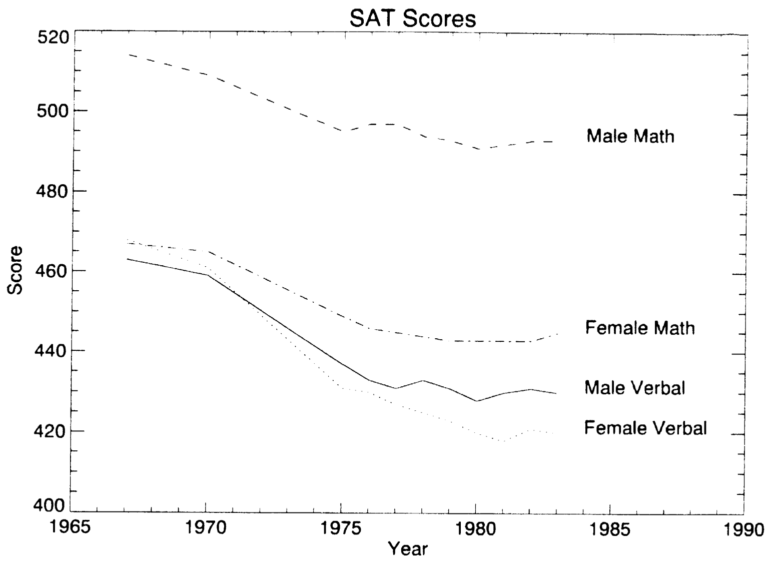


Figure 3-5 Example of annotating each line. The font used is the hardware-generated PostScript Times-Roman font.

Plotting in Histogram Mode

You can produce a histogram-style plot by setting the *Psym* keyword to 10 in the PLOT procedure call:

```
Psym = 10
```

This connects data points with vertical and horizontal lines, producing the histogram.

Figure 3-6 illustrates this by comparing the distribution of PV-WAVE's normally distributed random number function (RANDOMN), to the theoretical normal distribution:

$$(2\pi)^{-1/2}e^{-x^2/2}$$

This figure was produced by the following PV-WAVE statements:

```
X = FINDGEN(200) / 20. - 5.
```

200 values ranging from –5 to 5.

```
Y = 1 / SQRT(2. * !PI) * EXP(-X^2 / 2) *  $
   (10. / 200)
```

Theoretical normal distribution, scale so integral is one.

```
H = HISTOGRAM(RANDOMN(Seed, 2000), $
   BINSIZE = 0.4, min = -5., max = 5.)/2000.
```

Approximate normal distribution with RANDOM and then form the histogram.

```
PLOT, findgen(26) * 0.4 - 4.8, H, PSYM = 10
```

Plot the approximation using "histogram mode".

```
OPLOT, X, Y*8.
```

Overplot the actual distribution.

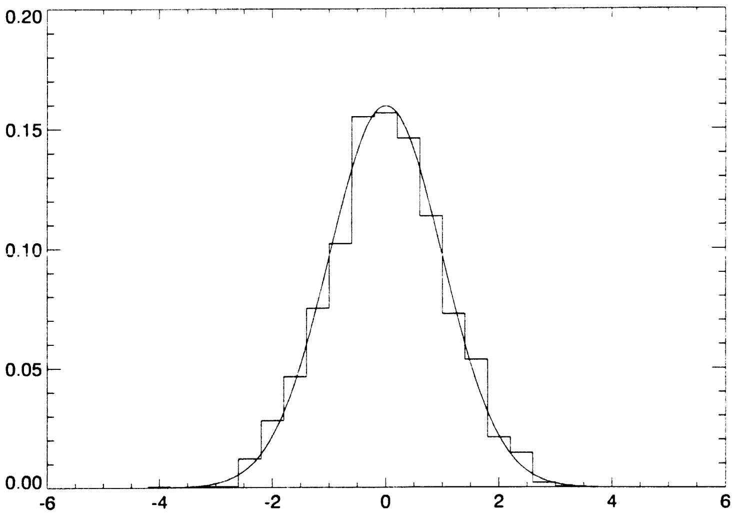


Figure 3-6 Plotting in histogram mode.

Using Different Marker Symbols

Each data point may be marked with a symbol and/or connected with lines. The value of the keyword parameter *Psym* selects the marker symbol. *Psym* is described in detail in *Graphics and Plotting Keywords* on page 497 of the *PV-WAVE Reference, Volume 2.*

For example, a value of 1 marks each data point with the plus sign, 2 is an asterisk, etc. Setting *Psym* to minus the symbol number marks the points with a symbol and connects them with lines. For example, a value of –1 marks points with a plus sign and connects them with lines.

Note also that setting *Psym* to a value of 10 produces histogram-style plots, as described in the previous section.

Frequently, when data points are plotted against the results of a fit or model, symbols are used to mark the data points while the model is plotted using a line. Figure 3-7 illustrates this, fitting the male verbal scores to a quadratic function of the year. The POLY_FIT function is used to calculate the quadratic. The statements used to construct this plot are:

```
COEFF = POLY_FIT(YEAR, VERBM, 2, YFIT)
```

Use the POLY_FIT function to obtain a quadratic fit.

```
PLOT, YEAR, VERBM, /YNozero, Psym = 4,  $
   TITLE = 'Quadratic Fit', XTITLE = 'Year', $
   YTITLE = 'SAT Score'
```

Plot the original data points with Psym = 4, for diamonds.

```
OPLOT, YEAR, YFIT
```

Overplot the smooth curve using a plain line.

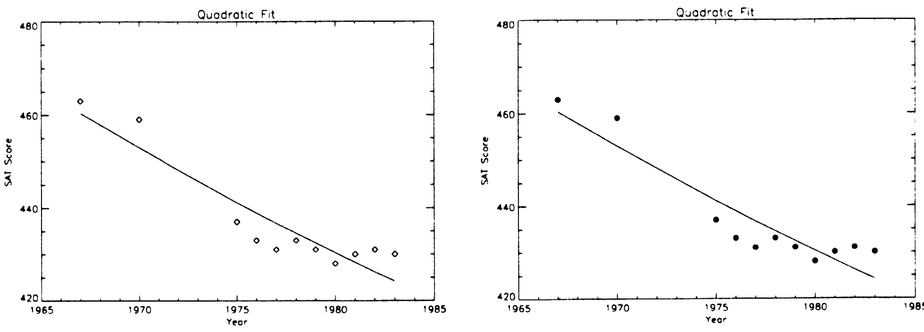


Figure 3-7 Plotting with predefined marker symbols (left) and user-defined symbols (right).

Defining Your Own Marker Symbols

The USERSYM procedure allows you to define your own symbols by supplying the coordinates of the lines used to draw the symbol. The symbol you define may be drawn using lines, or it may be filled using the polygon filling operator. USERSYM accepts two vector parameters: a vector of X values and a vector of Y values.

The coordinate system you use to define the symbol's shape is centered on each data point and each unit is approximately the size of a character. For example, to define the simplest symbol, a 1-character-wide dash, centered over the data point:

```
USERSYM, [-.5,.5],[0,0]
```

The color and line thickness used to draw the symbols are also optional keyword parameters of USERSYM.

The right half of Figure 3-7 illustrates the use of USERSYM to define a new symbol, a filled circle. It was produced in exactly the same manner as the example in the previous section, with the exception of the addition of the following statements that define the marker symbol and use it:

```
A = FINDGEN(16) * ( !PI * 2 / 16. )
```

Make a vector of 16 points, $a_i = 2\pi i / 16$.

```
USERSYM, COS(A), SIN(A), /Fill
```

Define the symbol to be a unit circle, with 16 points, set the filled flag.

```
PLOT, YEAR, VERBM, /YNozero, Psym = 8, ...
```

As in the previous section, but use symbol index 8 to select user-defined symbols.

Using Color and Pattern to Highlight Plots

Many scientific graphs use region filling to highlight the difference between two or more curves (i.e., to illustrate boundaries, etc.). Given a list of vertices, the PV-WAVE procedure POLYFILL fills the interior of an arbitrary polygon. The interior of the polygon may be filled with a solid color or, with some devices, a user-defined pattern contained in a rectangular array.

Figure 3-8 illustrates a simple example of polygon filling by filling the region under the male math scores with a color index of 75% the maximum, and then filling the region under the male verbal scores with a 50% of maximum index. Because the male math scores are always higher than the verbal, the graph appears as two distinct regions.

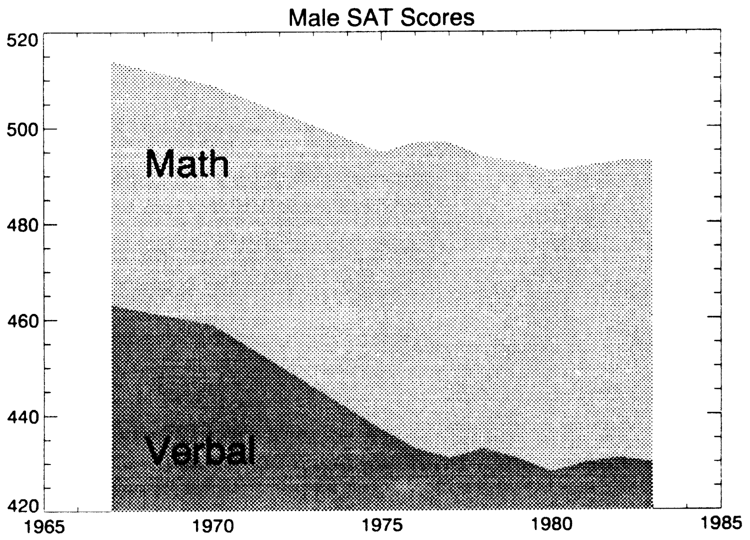


Figure 3-8 Filling regions using POLYFILL.

The program that produced Figure 3-8 is shown on the next page. It first draws a plot axis with no data, using the *Nodata* keyword. The minimum and maximum Y values are directly specified with the *YRange* keyword. Because the Y axis range does not always exactly include the specified interval (see *Scaling the Plot Axes and Adding Titles* on page 63), the variable `MINVAL`, is set to the current Y axis minimum, !Y.Crange(0). Next, the upper math score region is shaded with a polygon which contains the vertices of the math scores, preceded and followed by points on the X axis, (`YEAR(0), MINVAL`), and (`YEAR(`*n* `- 1), MINVAL`).

The polygon for the verbal scores is drawn using the same method with a different color. Finally, the XYOUTS procedure is used to annotate the two regions.

```
!P.FONT = 0
```

Use hardware fonts.

```
DEVICE, /Helvetica
```

Set font to Helvetica.

```
PLOT, year, mathm, YRANGE = [MIN(verbm), $
   MAX(mathm)], /NODATA, TITLE = $
   'Male SAT Scores'
```

Draw axes, no data, set the range.

```
pxval = [year(0), year, year(n1)]
```

Make a vector of X values for the polygon, by duplicating the first and last points.

```
minval = !y.crange(0)
```

Get Y value along bottom X axis.

```
POLYFILL, pxval, [minval, mathm, minval], $
   COL = 0.75 * !D.N_COLORS
```

Make a polygon by extending the edges of the math score down to the X axis.

```
POLYFILL, pxval, [minval, verbm, minval], $
   COL = 0.50 * !D.N_COLORS
```

Same with verbal.

```
XYOUTS, 1968, 430, 'Verbal', Size = 2
```

Label the polygons.

```
XYOUTS, 1968, 490, 'Math', Size = 2
```

Drawing Bar Charts

Bar charts are used in business-style graphics and are useful in comparing a small number of measurements within a few discrete data sets. Although not designed specifically to create business graphics, PV-WAVE can produce many seemingly complicated business-style plots with a little effort.

The following example produces a bar-style chart showing the four SAT scores as boxes of differing colors or shading. The program used to draw Figure 3-9 is shown below and annotated. A procedure called BOX is defined which draws a box given the coordinates of two diagonal corners.

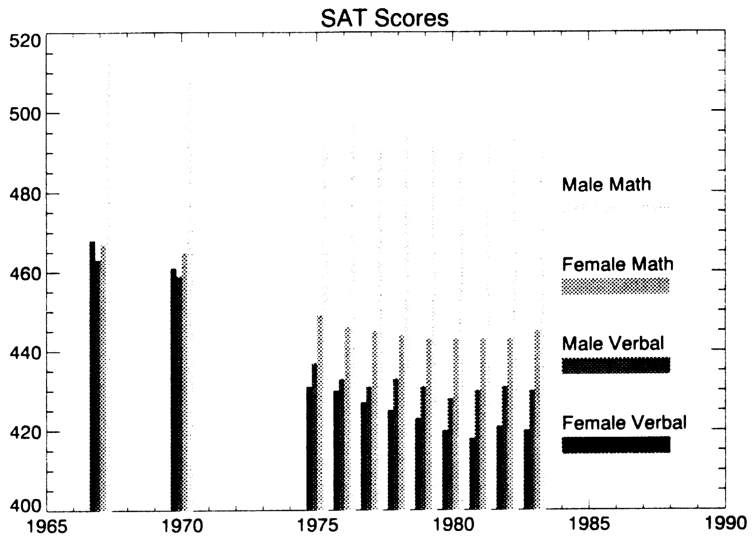


Figure 3-9 Bar chart drawn with POLYFILL.

As in the previous example, the PLOT procedure is used to draw the axes and establish the scaling using the *Nodata* keyword.

```
PRO BOX, x0, y0, x1, y1, color
```

Draw a box, using polyfill, whose corners are (x0, y0), and (x1,y1).

```
POLYFILL, [x0,x0,x1,x1], [y0,y1,y1,y0], $
   col = color
```

Call polyfill.

```
END
```

```
colors = 64 * INDGEN(4) + 32
```

Make a vector of colors for each score.

```
PLOT, year, mathm, yrange = [min(allpts), $
   max(allpts)], title = 'SAT Scores', $
   /nodata, xrange = [year(0), 1990]
```

Use PLOT to draw the axes and set the scaling. Draw no data points, explicitly set the X and Y ranges.

```
minval = !y.crange(0)
```

Get the Y value of the bottom X axis.

```
del = 1./5.
```

Width of bars in data units.

```
for iscore = 0,3 do begin
```

Loop for each score.

```
yannot = minval + 20 *(iscore+1)
```

Y value of annotation. Vertical separation is 20 data units.

```
xyouts, 1984, yannot, names(iscore)
```

Label for each bar.

```
BOX, 1984, yannot-6, 1988, yannot-2, $
   colors(iscore)
```

Bar for annotation.

```
xoff = iscore * del - 2 * del
```

X offset of vertical bar for each score.

```
for iyr = 0, N_ELEMENTS(year)-1 do $
   box, year(iyr)+xoff, minval, year(iyr)$
   + xoff+del, allpts(iyr, iscore), $
   colors(iscore)
```

Draw vertical box for each year's score.

```
ENDFOR
```

Controlling Tick Marks

You have almost complete control over the number, style, placement, and annotation of the tick marks. The following plotting keywords are used to control tick marks:

Gridstyle	XTicklen	YTickformat	ZMinor
Tickformat	XTickname	YTicklen	ZTickformat
Ticklen	XTicks	YTickname	ZTicklen
XGridstyle	XTickv	YTicks	ZTickname
XMinor	YGridstyle	YTickv	ZTicks
XTickformat	YMinor	ZGridstyle	ZTickv

For detailed descriptions of these keywords, see Chapter 3 *Graphics and Plotting Keywords*, in the *PV-WAVE Reference.*

Example 1: Specifying Tick Labels and Values

Figure 3-10 is a bar chart illustrating the direct specification of the X axis tick values, number of ticks, and tick names. Building upon the previous program, this program shows each of the four scores for the year 1967, the first year in our data. It uses the BOX procedure from the previous example to draw a rectangle for each score. Using the data and variables from above, the program is:

```
xval = FINDGEN(4)/5. + .2
```

Tick X values, 0.2, 0.4, 0.6, 0.8.

```
yval = [verbf(0), verbm(0), mathf(0), $
   mathm(0)]
```

Make a vector of scores from first year, corresponding to the name vector from above.

```
PLOT, xval, yval, /YNOZERO, XRANGE = [0,1],$
   XTICKV = xval, XTICKS = 3, $
   XTICKNAME = names, /NODATA, TITLE =  $
   'SAT Scores, 1967'
```

Make the axes with no data. Force X range to [0,1], centering xval, which also contains the tick values. Force three tick intervals making four tick marks. Specify the tick names from the names vector.

```
FOR i=0, 3 DO box, xval(i) - .08, $
   !y.crange(0), xval(i)+0.08, yval(i), $
   128
```

Draw the boxes, centered over the tick marks. !y.crange(0) is the Y value of the bottom X axis.

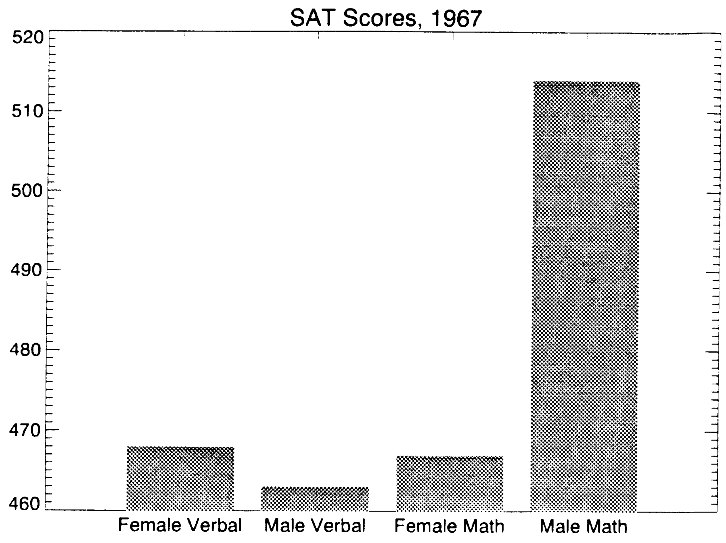


Figure 3-10 Controlling X axis tick marks and their annotation.

Example 2: Specifying Tick Lengths

Figure 3-11 illustrates effects of changing the *Ticklen* keyword. The left plot shows a full grid produced with tick mark lengths of 0.5. The right plot shows outward-extending tick marks produced by setting the *Ticklen* keyword to –0.02. Outward extending ticks are useful in that they do not obscure the data inside the window. These two plots were produced with the following code:

```
precip = [ ... ... ... ]
```

Define 12 monthly precipitation values.

```
temp = [ ... ... ... ]
```

Define 12 monthly average temperature.

```
month = ['Ja', 'Fe', 'Ma', 'Ap', 'Ma', $
    'Ju', 'Ju', 'Au', 'Se', 'Oc', 'No', $
    'De']
```

Define names of months.

```
day = findgen(12) * 30 + 15
```

Vector containing approximate day number of the middle of each month.

```
PLOT, day, precip, Xticks = 11, Xtickname = $
   month, Ticklen = 0.5, Xtickv = day, $
   Title = 'Average Monthly Precipitation', $
   Xtitle = 'Inches', Subtitle = 'Denver'
```

Plot, setting tick mark length to full, setting number, position and labels of ticks.

```
PLOT, day, precip, XTICKS = 11,XTICKNAME = $
   month, TICKLEN = -0.02, XTICKV = day, $
   TITLE = 'Average Monthly Precipitation', $
   XTITLE = 'Inches', SUBTITLE = 'Denver'
```

As above, setting tick mark length for outside ticks.

Tip Use the *Gridstyle*, *XGridstyle*, *YGridstyle*, and *ZGridstyle* keywords to change the linestyle of tick marks from solid to dashed, dotted, or other styles. One use for this is to create a dotted or dashed grid on the plot region. First set the *Ticklen* keyword to 0.5, and then set the *Gridstyle* keyword to the value of the linestyle you want to use. For more information on using the *Gridstyle* keywords, see Chapter 3 *Graphics and Plotting Keywords*, in the *PV-WAVE Reference*.

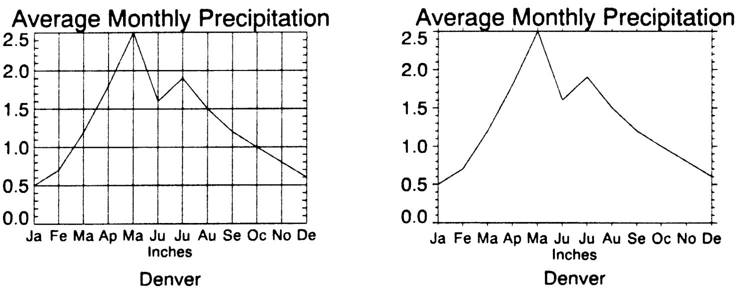


Figure 3-11 Full grid produced with tick marks (right) and outward-extending tick marks (left).

Example 3: Specifying Tick Label Formats

The *XTickformat*, *YTickformat*, and *ZTickformat* keywords let you change the default format of tick labels. These keywords use the `F` (floating-point), `I` (integer), and `E` (scientific notation) format specifiers to specify the format of the tick labels. These format specifiers are similar to the ones used in FORTRAN and are discussed in Appendix A, *FORTRAN and C Format Strings*, in the *PV‑WAVE Programmer's Guide.*

For example:

```
PLOT, mydata, XTickformat='(F5.2)'
```

The resulting plot's tick labels are formatted with a total width of five characters and carried to two decimal places. As expected, the width field expands automatically to accommodate larger values. For example, the X axis tick labels for this plot might look like this:

```
40.00400.004000.0040000.00
```

You can easily reformat the labels in scientific notation using the `E` format specifier. For example:

```
PLOT_OO, mydata, YTickformat='(E6.2)'
```

The resulting Y axis tick labels for this plot might look like this:

```
1.00e-081.00e-061.00e-041.00e-02
```

Like many of the keywords used with the plotting procedures, corresponding system variables allow you to change the normal defaults. The corresponding system variables for the *Tickformat* keywords are: !X.Tickformat, !Y.Tickformat, and !Z.Tickformat. The system variable !P.Tickformat lets you set the tick label format for all three axes.

Note Only the `I` (integer), `F` (floating-point), and `E` (scientific notation) format specifiers can be used with the *Tickformat* keywords. Also, you cannot place a quoted string inside a tick format. For example, `("<", F5.2, ">")` is an invalid *Tickformat* specification.

Drawing Multiple Plots on a Page

Plots may be grouped on the display or page in the horizontal and/ or vertical directions using the !P.Multi system variable field. PV-WAVE sets the plot window to produce the given number of plots on each page and moves the window to a new sector at the beginning of each plot. If the page is full, it is first erased. If more than two rows or columns of plots are produced, PV-WAVE decreases the character size by a factor of 2.

!P.Multi controls the output of multiple plots and is an integer vector in which:

- !P.Multi(0) – The number of empty sectors remaining on the page. The display is erased if this field is 0 when a new plot is begun.
- !P.Multi(1) – The number of plots across the page.
- !P.Multi(2) – The number of plots per page in the vertical direction.

For example to set up PV-WAVE to stack two plots vertically on each page:

```
!P.Multi = [0,1,2]
```

Note that the first element, !P.Multi(0), is set to zero to cause the next plot to begin a new page. To make four plots per page, with two columns and two rows:

```
!P.Multi = [0,2,2]
```

Figure 3-12 illustrates this format. To reset back to the default of one plot per page:

```
!P.Multi = 0
```

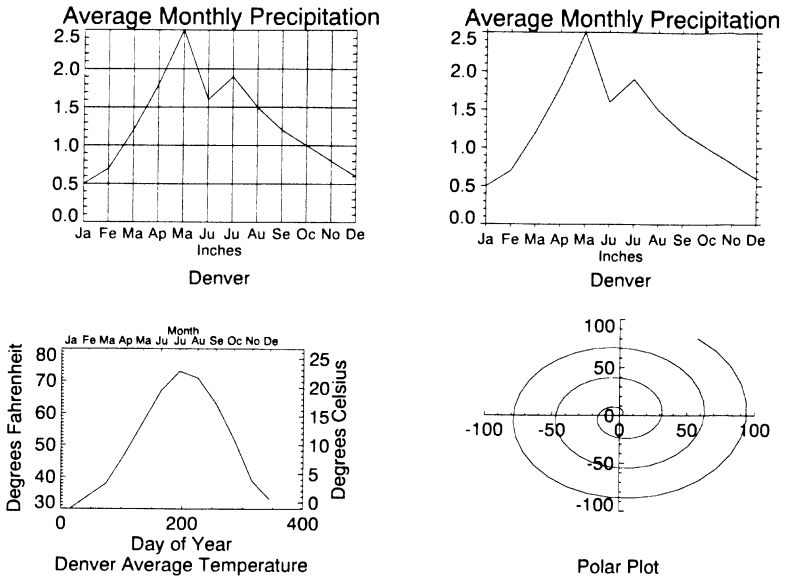


Figure 3-12 Multiple plots per page.

Plotting with Logarithmic Scaling

The *XType*, *YType*, and *ZType* keywords can be used with the PLOT routine to get any combination of linear and logarithmic axes. In addition, logarithmic scaling may be achieved by calling PLOT_IO (*li*near X axis, *lo*g Y axis), PLOT_OI (*lo*g X, *li*near Y), or PLOT_OO (*lo*g X, *lo*g Y). The OPLOT procedure uses the same scaling and transformation as did the most recent plot.

Figure 3-13 illustrates using PLOT_IO to make a linear-log plot. It was produced with the following statements:

```
X = FLTARR(256)
```

Create data array.

```
X(80:120) = 1
```

Make a step function.

```
FREQ = FINDGEN(256)
```

Make a filter.

```
FREQ = FREQ < (256-FREQ)
```
.. symmetrical about x = 64.

```
FIL = 1. / (1+(FREQ / 20) ^2)
```
2nd order Butterworth, cutoff freq = 20.

```
PLOT_IO, FREQ, ABS(FFT(X,1)), XTITLE = $
   'Relative Frequency', YTITLE = 'Power', $
   xstyle = 1
```
Plot with a logarithmic X axis. Use exact axis range.

```
OPLOT, FREQ, FIL
```
Show it.

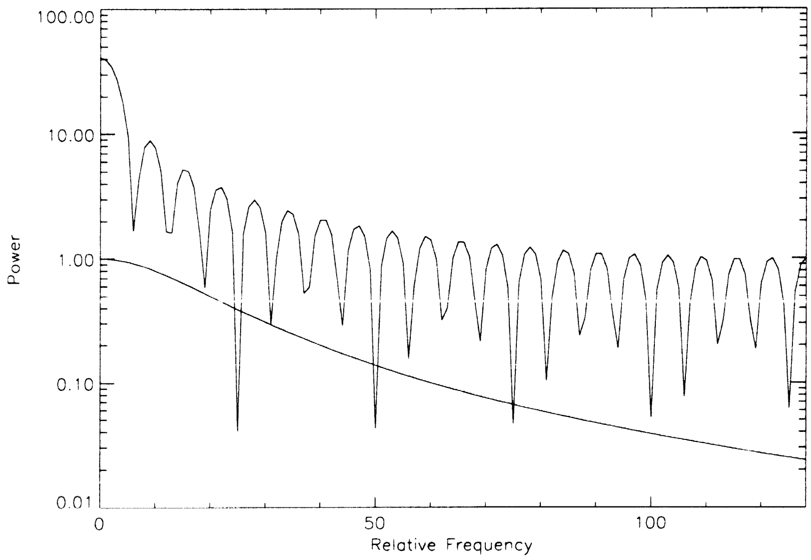


Figure 3-13 Example of logarithmic scaling.

Specifying the Location of the Plot

The *plot data window* is the region of the page or screen enclosed by the axes. The *plot region* is the box enclosing the plot data window and the titles and tick annotation. Figure 3-14 illustrates the relationship of the plot data window, plot region, and the entire device area (or window if using a windowing device).

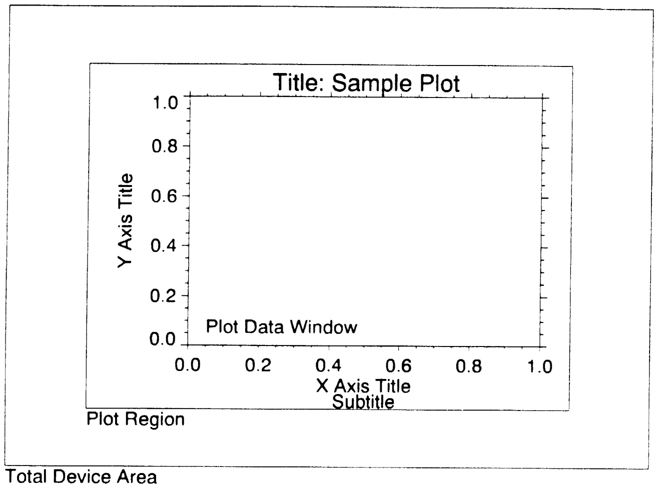


Figure 3-14 Relationship of the plot data window, plot region, and the device area.

These areas are determined by the following system variables and keyword parameters, in order of decreasing precedence. Each of these keywords and system variables are described in Chapter 3, *Graphics and Plotting Keywords* and Chapter 4, *System Variables*, in the *PV-WAVE Reference*.

- Position keyword
- !P.Position system variable
- !P.Region system variable
- !P.Multi system variable
- *XMargin*, *YMargin*, and *ZMargin* keywords
- !X.Margin, !Y.Margin, and !Z.Margin system variables

Drawing Additional Axes on Plots

The AXIS procedure draws and annotates an axis. It optionally saves the scaling established by the axis for use by subsequent graphics procedures. It may be used to add additional axes to plots, or to draw axes at a specified position.

The AXIS procedure accepts the set of plotting keyword parameters that govern the scaling and appearance of the axes. In addition, the keyword parameters *XAxis*, *YAxis*, and *ZAxis* specify the orientation and position (if no position coordinates are present), of the axis. The values of these parameters are: 0 for the bottom or left axis, and 1 for the top or right. The tick marks and their annotation extend away from the plot window. For example, specify `YAXIS = 1` to draw a Y axis on the right of the window.

The optional keyword parameter *Save* saves the data-scaling parameters established for the axis in the appropriate axis system variable, !X, !Y, or !Z.

The call to AXIS is:

AXIS [[, *x*, *y*], *z*]

where *x*, *y*, and optionally *z* specify the coordinates of the axis. By including the appropriate keyword parameter (*Device*, *Normal*, or *Data*) you can specify a coordinate system. The coordinate corresponding to the axis direction is ignored when specifying an X axis, the *X* coordinate parameter is ignored, but must be present if there is a Y coordinate.

Drawing Additional Axes Example

Figure 3-15 illustrates using AXIS to draw axes with a different scale, opposite the main X and Y axes.

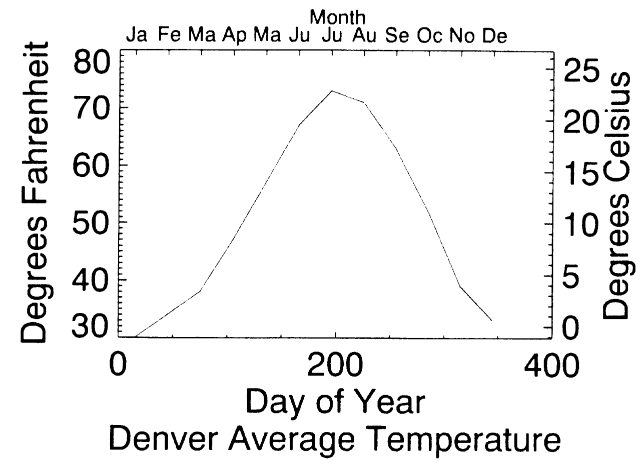


Figure 3-15 Plot containing axes with different scales, created with the AXIS procedure.

The plot is produced using PLOT with the bottom and left axes annotated and scaled in units of days and degrees Fahrenheit, respectively. The *XMargin* and *YMargin* keyword parameters are specified to allow additional room around the plot window for the new axes. The keyword parameters `XStyle = 8` and `YStyle = 8` inhibit drawing the top and right axes.

Next, the AXIS procedure is called to draw the top axis, (`XAxis = 1`), labeled in months. Eleven tick intervals, with 12 tick marks are drawn. Each monthly tick mark's X value is the day of the year of approximately the middle of the month. Tick mark names come from the `MONTH` string array.

The right Y axis, `YAxis = 1`, is drawn in the same manner. The new Y axis range is set by converting the original Y axis minimum and maximum values, saved by PLOT in !Y.Crange, from Fahrenheit to Celsius, using the formula C = 5(F – 32) / 9. The keyword parameter `YStyle = 1` forces the Y axis range to match the given range exactly. The commands are:

```
PLOT, day, temp, /YNOZERO, SUBTITLE = $
```

```
    'Denver Average Temperature', $
    XTITLE  = 'Day of Year', YTITLE = $
    'Degrees Fahrenheit', XSTYLE = 8, $
    YSTYLE = 8, XMARGIN = [8,8], YMARGIN = [4,4]
```

Plot the data, omit right and top axes.

```
AXIS, XAXIS = 1, XTICKS = 11, XTICKV = day, $
    XTICKN = month, XTITLE = 'Month', $
    XCHARSIZE = 0.7
```

Draw the top X axis, supplying labels, etc. Make the characters smaller so they will fit.

```
AXIS, YAXIS = 1, YRANGE = $
    (!y.crange—32)*5. /9., YSTYLE = 1, $
    YTITLE = 'Degrees Celsius'
```

Draw right Y axis. Scale current Y axis minimum values from Fahrenheit to Celsius, and make them the new min and max values. Set YSTYLE to 1 to make the axis exact.

Drawing Polar Plots

The PLOT procedure converts its coordinates from cartesian to polar coordinates when plotting if the *Polar* keyword parameter is set. The first parameter to plot is the radius, *R*, and the second is θ, expressed in radians. Polar plots are produced using the standard axis and label styles — with box axes enclosing the plot area.

Figure 3-16 illustrates using AXIS to draw centered axes, dividing the plot window into the four quadrants centered about the origin. This method uses PLOT to plot the polar data and to establish the coordinate scaling, but suppresses the axes. Next, two calls to AXIS add the X and Y axes, drawn through data coordinate (0,0):

```
r = findgen(100)
```

Make a radius vector.

```
theta = r/5
```

And a theta vector.

```
PLOT, r, theta, SUBTITLE = 'Polar Plot', $
    XSTY = 4, YSTY = 4, /POLAR
```

Plot the data, suppressing the axes by setting their styles to 4.

```
AXIS, XAX = 0, 0, 0
   AXIS, YAX = 0, 0, 0
```

Draw the X and Y axes through (0,0).

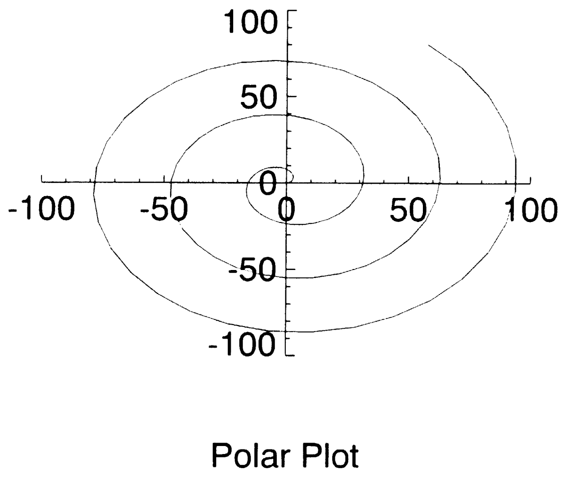


Figure 3-16 A polar plot.

Getting Input from the Cursor

The CURSOR procedure reads the position of the graphics cursor of the current graphics device. It enables the graphic cursor on the device, optionally waits for the user to position it and press a mouse button to terminate the operation (or type a character if the device has no mouse), and then reports the cursor position.

The form of a call to CURSOR, where *x* and *y* are output variables that hold the X and Y position of the cursor, and *wait* specifies when CURSOR returns is:

CURSOR, *x*, *y* [, *wait*]

For detailed information on the CURSOR procedure, its parameters and optional keywords, see the description in the *PV-WAVE Reference.*

The following code lets you draw lines between points marked with the left or middle mouse button. Press the right mouse button to exit the routine.

```
ERASE
```

Start with a blank screen.

```
CURSOR, X, Y, /Normal, /Down
```

Get the initial point in normalized coordinates.

```
WHILE (!ERR NE 4) DO BEGIN
```

Repeat until the right button is pressed.

```
CURSOR, X1, Y1, /Normal, /Down
```

Get the second point.

```
PLOTS, [X, X1], [Y, Y1], /Normal
```

Draw the line.

```
X = X1 & Y = Y1
```

Make the current second point be the new first.

```
ENDWHILE
```

For another example, the following simple procedure can be used to label plots using the cursor to position the text:

```
PRO ANNOTATE, TEXT
```

Text is the string to be written on the screen.

```
PRINT, 'Use the mouse to mark the' + $
   ' text starting point:'
```

Ask the user to mark the position.

```
CURSOR, X, Y, /Normal, /Down
```

Get the cursor position after any button press.

```
XYOUTS, X, Y, TEXT, /Normal, /Noclip
```

Write the text at the specified position. The NOCLIP keyword is used to ensure that the text will appear even if it is outside of the plotting region.

```
END
```

To place the annotation on a device with an interactive pointer, call this procedure with the command:

```
ANNOTATE, 'Text for label'
```

Then move the mouse to the desired spot and press the left button.

4

CHAPTER

Displaying 3D Data

This chapter shows how to display graphic representations of three-dimensional data. The two main procedures for doing this are CONTOUR and SURFACE. Procedures for displaying data as an image, another type of three-dimensional data representation, are discussed in Chapter 5, *Displaying Images*. The 3D plotting procedures include:

CONTOUR, *z* [, *x*, *y*]
: Draws contour plots.

SURFACE, *z* [, *x*, *y*]
: Draws 3D surface plots.

SHADE_SURF, *z* [, *x*, *y*]
: Draws shaded 3D surface plots.

CONTOUR and SURFACE both use line graphics to depict the value of a two-dimensional array. As its name implies, CONTOUR draws contour plots. SURFACE depicts the surface created by interpreting each array element as an elevation. SURFACE projects this three-dimensional surface, after an arbitrary rotation about the X and Z axis, into two dimensions. It then connects each point with its neighbors using hidden line removal.

Almost all of the information concerning coordinate systems, keyword parameters, and system variables that are discussed in

Chapter 3, *Displaying 2D Data*, also apply to CONTOUR and SURFACE. The keywords and system variables discussed in this chapter are described in detail in the *PV‑WAVE Reference*.

Drawing Contour Plots

The CONTOUR procedure draws contour plots from data stored in a rectangular array. In its simplest form, CONTOUR makes a contour plot given a two-dimensional array of Z values. In more complicated forms, CONTOUR accepts, in addition to Z values, arrays containing the X and Y locations of each column, row, or point, plus many keyword parameters. In more sophisticated applications, the output of CONTOUR may be projected from three dimensions to two dimensions, superimposed over an image, or combined with the output of SURFACE.

The simplest call to CONTOUR is:

```
CONTOUR, z
```

This call labels the X and Y axes with the subscript along each dimension. For example, when contouring a 10-by-20 array, the X axis ranges from 0 to 9, and the Y from 0 to 19.

You can explicitly specify the X and Y locations of each cell with the call:

```
CONTOUR, z, x, y
```

The *x* and *y* arrays may be either vectors or two-dimensional arrays of the same size as *z*. If they are vectors, the element $Z_{i,j}$ has a coordinate location of (X_i, Y_j). Otherwise, if the *x* and *y* arrays are two-dimensional, the element $Z_{i,j}$ has the location $(X_{i,j}, Y_{i,j})$ Thus, vectors should be used if the X location of $Z_{i,j}$ does not depend upon *j* and the Y location of $Z_{i,j}$ does not depend upon *i*.

Dimensions must be compatible. In the one-dimensional case, *x* must have a dimension equal to the number of columns in *z*, and *y* must have a dimension equal to the number of rows in *z*. In the two-dimensional case, all three arrays must have the same dimensions.

PV-WAVE uses linear interpolation to determine the X and Y locations of the contour lines that pass between grid elements. The cells must be regular, in that the X and Y arrays must be monotonic over rows and columns, respectively. The lines describing the quadrilateral enclosing each cell and whose vertices are $(X_{i,j}, Y_{i,j})$, $(X_{i+i,j}, Y_{i+1,j})$, $(X_{i+1,j+i}, Y_{i+i,j+i})$, and $(X_{i,j+1}, Y_{i,j+1})$ must intersect only at the four corners.

Alternative Contouring Algorithms

In order to provide a wide range of options, CONTOUR uses either the cell drawing or the follow method of drawing contours.

Cell Method

The cell drawing method is used by default. It examines each array cell and draws all contours emanating from that cell before proceeding to the next cell. This method is efficient in terms of computer resources but does not allow such options as contour labeling or smoothing.

Follow Method

The follow method searches for each contour line and then follows the line until it reaches a boundary or closes. This method gives better looking results with dashed linestyles, and allows contour labeling and bicubic spline interpolation, but requires more computer time. It can be used in with the POLYCONTOUR procedure to shade closed contour regions with specified colors, as explained in *Filling Contours with Color* on page 109. The follow method is used if any of the following keywords is specified: *C_Annotation*, *C_Charsize*, *C_Labels*, *Follow*, *Path_Filename*, or *Spline*.

Note Because of their differing algorithms, these two methods will often draw slightly different correct contour maps for the same data. This is a direct result of the fact that there is often more than one valid way to draw contours, and should not be a cause for concern.

Controlling Contour Features with Keywords

In addition to most of the keyword parameters accepted by PLOT, the following keywords apply to CONTOUR.

C_Annotation	C_Labels	Follow	NLevels
C_Charsize	C_Linestyle	Levels	Path_Filename
C_Colors	C_Thick	Max_Value	Spline

For a detailed description of these keywords, see Chapter 3 *Graphics and Plotting Keywords*, in the *PV-WAVE Reference.*

Contouring Example

Digital elevation data of the Maroon Bells area, near Aspen, Colorado, are used to illustrate the CONTOUR procedure. This data provides terrain elevation data over a 7.5 minute square (approximately 11-by-13.7 kilometers at the latitude of Maroon Bells), with 30 meter sampling measured in Universal Transverse Mercator (UTM) coordinates.

The data are read into a 350-by-460 array A. The rectangular array is not completely filled with data, because the 7.5 minute square is not perfectly oriented to the UTM grid system. Missing data are represented as zeroes. Elevation measurements range from 2658 to 4241 meters, or from 8720 to 13,914 feet.

Figure 4-1 is the result of applying the CONTOUR procedure to the data, using the default settings:

```
CONTOUR, A
```

A number of problems are apparent:

- PV-WAVE selected six contour levels, by default, of (4241 – 0) / 7 meter intervals, or approximately 605 meters. The levels are 605, 1250, ..., 3635, meters, even though the range of valid data is from 2658 to 4241 meters. This is because the missing data values of 0 were considered when selecting the intervals. It is more appropriate to select levels only within the range of valid data.

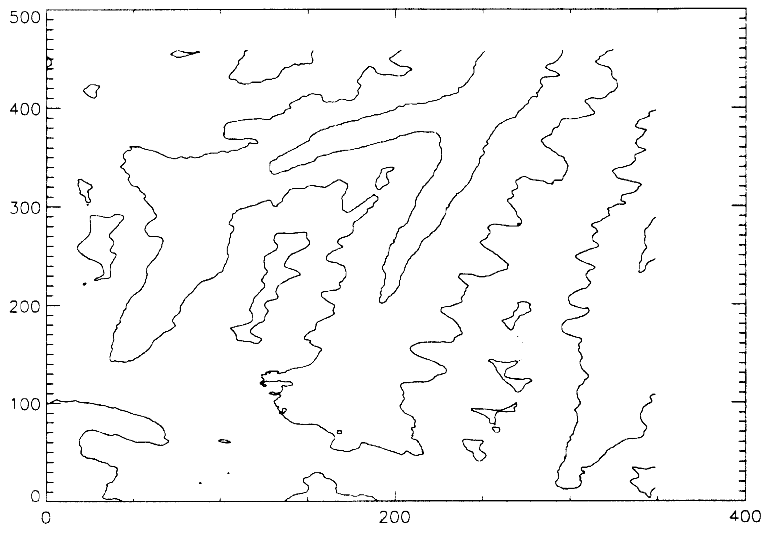


Figure 4-1 Simple contour plot of Maroon Bells.

- For most display systems, and for contour intervals of approximately 200 meters, the data are oversampled in the XY direction. This oversampling has two adverse effects: the contours appear jagged, and a large number of short vectors are produced. This can cause performance problems when you attempt to plot the data on a graphics device, especially if the graphic output is directed to a serial terminal or PostScript printer.
- The axes are labeled by point number, but should be in UTM coordinates.
- It is difficult to visualize the terrain and to discern maxima from minima because each contour is drawn with the same type of line.

Each of the above problems is readily solved using the following simple techniques:

- ❑ Specify the contour levels directly using the *Levels* keyword parameter. Selecting contour intervals of 250 meters, at elevation levels of [2750, 3000, 3250, 3500, 3750, 4000], results in six levels.
- ❑ Change the missing data value to a value well above the maximum valid data value. Then use the *Max_Value* keyword parameter to exclude missing points. In this example, we set missing data values to one million with the statement:

  ```
  A(WHERE(A EQ 0)) = 1.0E6
  ```

- ❑ Use the REBIN function to decrease the sampling in X and Y by a factor of 5:

  ```
  B = REBIN(A, 350/5, 460/5)
  ```

  This smooths the contours, because the call to REBIN averages $5^2 = 25$ bins when resampling. The number of vectors transmitted to the display are also decreased by a factor of approximately 25. The variable `B` is now a 70-by-92 array.

  Care was taken, in the second step, to ensure that the missing data are not confused with valid data after REBIN is applied. As, in this example, REBIN averages bins of $5^2 = 25$ elements, the missing data value must be set to a value of at least 25 times the maximum valid data value. After application of REBIN any cell with a missing original data point will have a value of at least $10^6/25 = 40000$, well over the largest valid data value of approximately 4500.
- ❑ X and Y vectors are constructed containing the UTM coordinates for each row and column. From the USGS data tape, the UTM coordinate of the lower-left corner of the array is (326850, 4318500) meters. As the data spacing is 30 meters in both directions, the X and Y vectors, in kilometers, are easily formed using the FINDGEN function, as shown in the following example.
- ❑ Contour levels at each multiple of 500 meters (every other level), are drawn with a solid line style, while levels in between are drawn with a dotted line style. In addition, the

4000 meter contour is drawn with a triple thick line, emphasizing the top contour.

The result of these improvements is Figure 4-2. It was produced with the following PV-WAVE statements:

```
a(WHERE(a eq 0)) = 1e6
```

Set missing data points to a large value.

```
b = REBIN(a, 350/5, 460/5)
```

Rebin down to a 70-by-92 matrix.

```
x = 326.850 + .030 * FINDGEN(70)
```

```
y = 4318.500 + .030 * FINDGEN(92)
```

Make the X and Y vectors giving the position of each column and row.

```
CONTOUR, b, x, y, LEVELS = 2750+FINDGEN(6) * $
   250., XSTYLE = 1, YSTYLE = 1, MAX_VALUE = $
   5000, C_LINESTYLE = [1,0,1,0,1,0], $
   C_THICK = [1,1,1,1,1,3], $
   TITLE = 'Maroon Bells Region', $
   SUBTITLE = '250 meter contours', $
   XTITLE = 'UTM Coordinates (KM)'
```

Make the plot, specifying the contour levels, missing data value, line styles, etc. Set the style keywords to 1, obtaining exact axes.

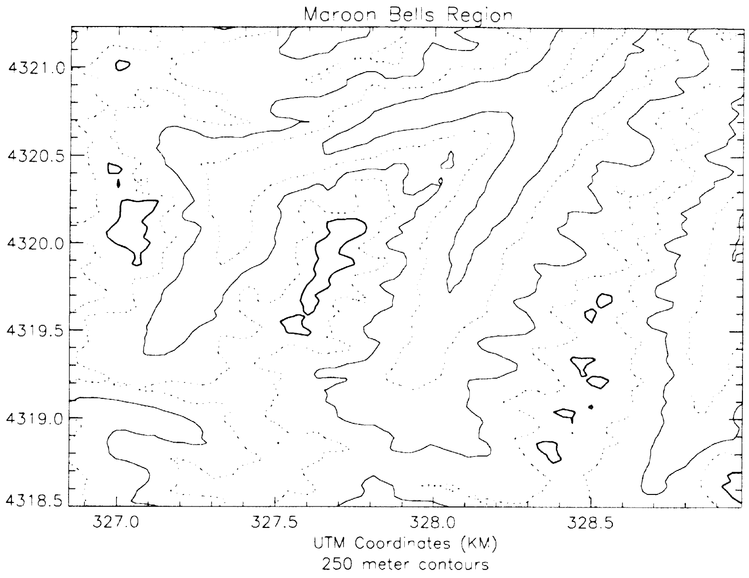


Figure 4-2 Improved contour plot.

Overlaying Images and Contour Plots

Figure 4-3 illustrates the data displayed as a gray-scale image. Higher elevations are white. This image demonstrates that contour plots do not always provide the best qualitative visualization of many two-dimensional data sets.

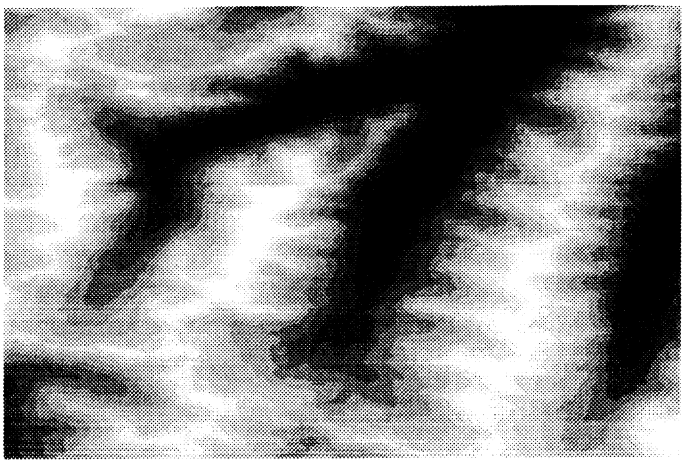


Figure 4-3 Maroon Bells data displayed as an image.

Superimposing an image and its contour plot combines the best of both worlds; the image allows easy visualization, and the contour lines provide a semi-quantitative display.

Note Beginners may want to skip the programs presented in the rest of this section. A combined contour and image display, such as that discussed in this section, can be created using the IMAGE_CONT procedure. The following material is intended to illustrate the many ways in which images and graphics may be combined using PV-WAVE.

The technique used to overlay plots and images depends on whether or not the device is able to represent pixels of variable size, as does PostScript, or if it has pixels of a fixed size. If the device does not have scalable pixels the image must be resized to fit within the plotting area (if it is not already of a size suitable for viewing). This leads to three separate cases which are illustrated in the following examples.

Overlaying on Devices with Scalable Pixels

Certain devices, notably PostScript, can display pixels of any given size. With these devices, it is easy to set the size and position of an image so that it exactly overlays the plot window. For example, the following statements were used to produce Figure 4-4:

```
c = BYTSCL(a, MIN = 2658, MAX=4241)
```

Scale the range of valid elevations into intensities.

```
TV, c, !x.window(0), !y.window(0), $
   XSIZE = !x.window(1) - !x.window(0), $
   YSIZE = !y.window(1) - !y.window(0), /NORM
```

Display the image with its lower-left corner at the origin of the plot window, and with its size scaled to fit the plot window.

```
CONTOUR, b, x, y, LEVELS = 2750+FINDGEN(6) $
   *250., MAX_VALUE = 5000, XSTYLE = 1, $
   YSTYLE =1, /NOERASE, $
   TITLE = 'Maroon Bells Region', $
   SUBTITLE = '250 meter contours', $
   XTITLE = 'UTM Coordinates (KM)'
```

Write the contours over the image, being sure to use the exact axis styles so that the contours fill the plot window. Inhibit erasing.

Be sure that the position of the plot window contained in the field Window in !X, !Y, and !Z, is set, using CONTOUR or PLOT, before executing the above statements.

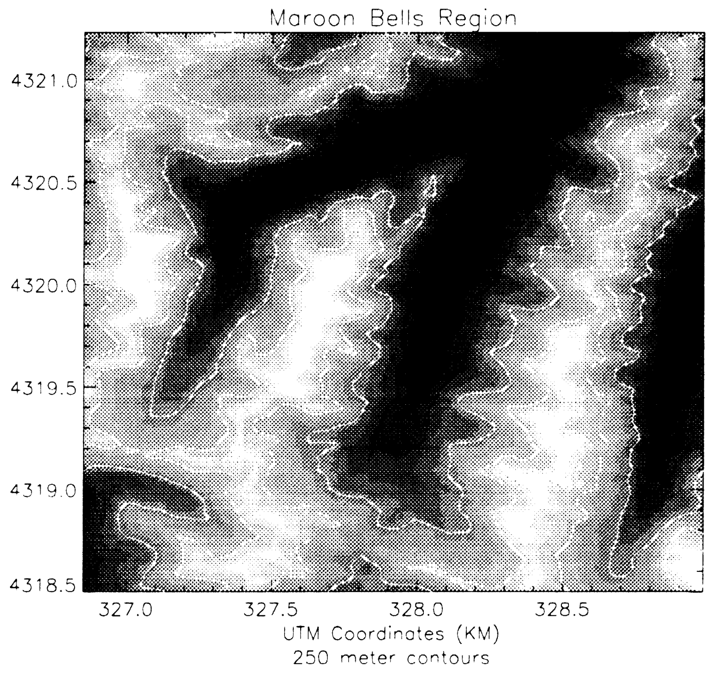


Figure 4-4 Overlay of image and contour plot.

Also, note that in Figure 4-4 that the aspect ratio of the image was changed to fit that of the plot window. If it is desired to retain the original image aspect ratio, the plot window must be resized to an identical aspect ratio using the *Position* keyword parameter.

Overlaying on Devices with Fixed Pixels

Method 1

If the pixel size can't be changed, for example on a Sun workstation monitor, an image of the same size as the plotting window must be created using the POLY_2D function. The REBIN function can also be used to resample the original image, if the plot window dimensions are an integer multiple or factor of the original image dimensions. REBIN is always faster than POLY_2D.

The following commands create an image of the same size as the window, display it, and then overlay the contour plot. These commands perform the same basic function as the IMAGE_CONT procedure, which is described in the *PV-WAVE Reference.*

```
px = !x.window * !d.x_vsize
py = !y.window * !d.y_vsize
```

Get size of plot window in device pixels.

```
sx = px(1)-px(0)+1
sy = py(1)-py(0)+1
```

Desired size of image in pixels.

```
sz = SIZE(a)
```

Get size of original image. SZ(1) = number of columns, SZ(2) = number of rows.

```
ERASE
```

Erase the display.

```
TV, POLY_2D(BYTSCL(a), [[0,0], $
   [sz(1)/sx,0]], [[0,sz(2)/sy],[0,0]], $
   0, sx, sy), px(0), py(0)
```

Create a sx-by-sy image stretched from the original. Display it with same lower-left corner coordinate as the window. Note that we BYTSCL *before* changing the size, as it is more efficient to apply POLY_2D to byte images. Also, it is likely that the original image is smaller than the stretched image.

```
CONTOUR, a, /NOERASE, XSTYLE=1,YSTYLE=1
```

Draw the contour without first erasing the screen.

Method 2

If the image is already close to the proper display size, it is simpler and more efficient to change the plot window size to that of the image. The following commands display the image at the window origin, and then set the plot window to the image size, leaving its origin unchanged:

```
px = !x.window * !d.x_vsize
```

Get size of the plot window in device pixels.

```
py = !y.window * !d.y_vsize
sz = size(a)
```
Size of original image.

```
ERASE
```
Clear the display.

```
TVSCL, a, px(0), py(0)
```
Scale and display the image at the lower left corner of the plot window.

```
CONTOUR, a, /Noerase, XStyle = 1, YStyle = 1,$
   Position = [px(0), py(0),  px(0)+sz(1)-1,$
   py(0)+sz(2)-1], /Device
```
Make the contour, explicitly set the plot window, in device coordinates to the size of the image. Make the axes exact. Don't erase.

Of course, by using other keyword parameters with the CONTOUR procedure, you can further customize the results.

Labeling Contours

In the following discussion, a variable named DATA is contoured. This variable contains uniformly distributed random numbers obtained using the following statement:

```
DATA = RANDOMU(SEED, 6, 6)
```

Note The default SEED value is used to create the DATA variable. Because of this, if you try to run these examples, your output will probably differ somewhat from the illustrations shown.

To label contours using the defaults for label size and contours to label, it is sufficient to simply select the *Follow* keyword. In this case, CONTOUR labels every other contour using the default label size (3/4 of the plot axis label size). Each contour is labeled with its value. Figure 4-5 was produced using the statement:

```
CONTOUR, /Follow, DATA
```

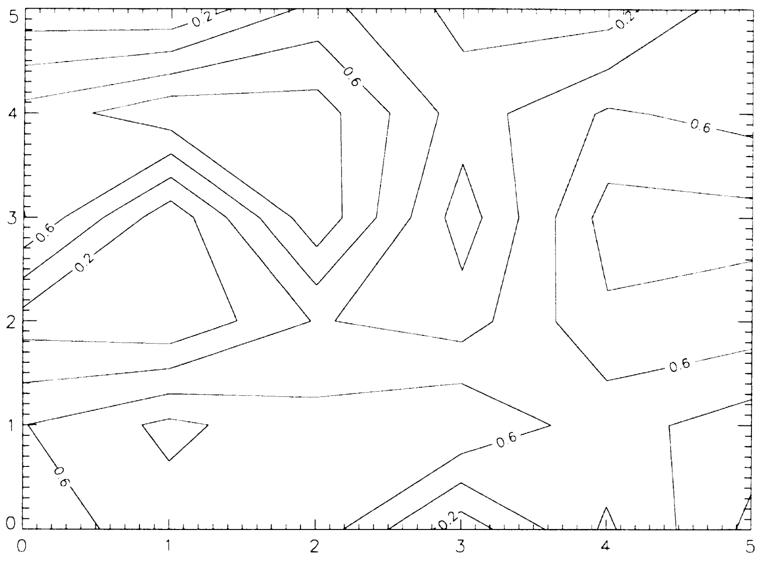


Figure 4-5 Simple labeled contour plot.

The *C_Charsize* keyword is used to specify the size of the characters used for labeling, in the same manner that *Size* is used to control plot axis label size. The *C_Labels* keyword can be used to select the contours to be labeled. For example, suppose that we want to contour the variable DATA at 0.2, 0.5, and 0.8, and we want all three levels labeled. In addition, we wish to make each label larger, and use PostScript fonts. This can be accomplished with the statement:

```
CONTOUR, LEVEL=[0.2,0.5,0.8], $
   C_LABELS=[1,1,1], C_CHARSIZE=1.25, DATA, $
   FONT=0
```

Note that Font=0 is used to specify the use of hardware fonts. For more information on hardware fonts, see *Software vs. Hardware Fonts: How to Choose* on page 291.

The result of this statement is shown in Figure 4-6.

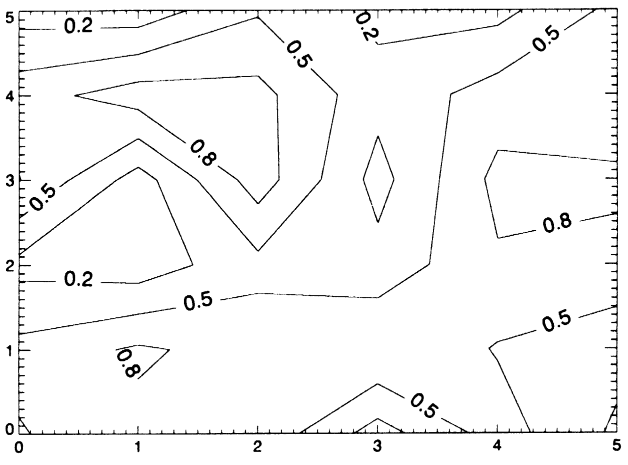


Figure 4-6 Label size and levels specified.

Finally, it is possible to specify the text to be used for the contour labels using the *C_Annotation* keyword.

```
CONTOUR, LEVEL=[0.2,0.5,0.8], C_LABELS= $
   [1,1,1], C_ANNOTATION= ["Low" , "Medium", $
   "High"], DATA, FONT=0
```

The result is shown in Figure 4-7.

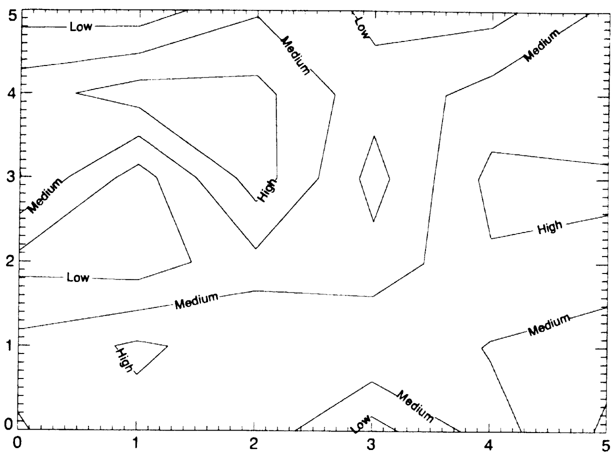


Figure 4-7 Explicitly specified labels.

Smoothing Contours

When the SPLINE keyword is specified, CONTOUR smooths the contours using cubic splines. This is especially effective when used with sparse data sets — the effectiveness of smoothing diminishes if enough data points are present and the cost of the spline calculations increases. Use of spline interpolation is not recommended when the array dimensions are more than approximately 15.

The effect of smoothing the variable DATA using the statement:

```
CONTOUR, LEVEL=[0.2,0.5,0.8], C_LABELS= $
   [1,1,1], /SPLINE, DATA
```

can be seen in Figure 4-8. Compare it with the non-smoothed versions in Figure 4-6 and Figure 4-7.

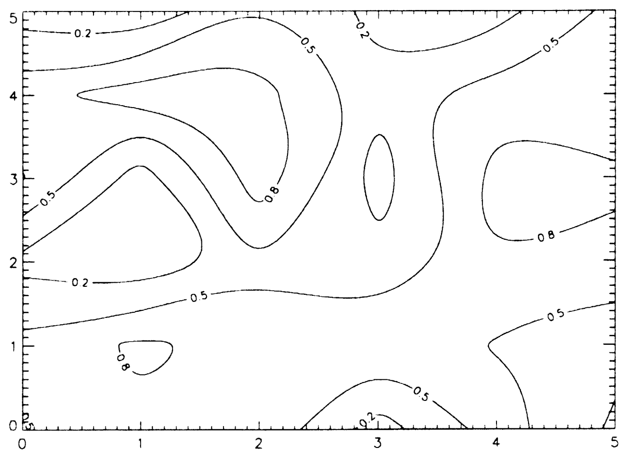


Figure 4-8 Contour plot with smoothing via cubic splines.

Filling Contours with Color

It is possible to fill closed contours with color by using the keyword *Path_Filename* in conjunction with the procedure POLYCONTOUR. *Path_Filename* specifies the name of a file to contain the contour positions. If *Path_Filename* is present, CONTOUR does not draw the contours, but rather, opens the specified file and writes the positions, in normalized coordinates, into it. The file thus produced is used by POLYCONTOUR to fill the closed contours with different colors. POLYCONTOUR has the form:

POLYCONTOUR, *filename [, Color_Index = cin]*

where *filename* is the name of the file written by CONTOUR an *cin* is the color index array. Element 0 of *cin* contains the background color, and each of the following elements contains the color that the corresponding contour level should be filled with. If the *Color_Index* keyword is not specified, POLYCONTOUR supplies a default set of colors.

The problem with directly producing a plot in this manner is that most of the contours are not closed, as they run beyond the borders of the plot. Since POLYCONTOUR can only fill closed contours, many of the contours will not be filled. This can be avoided by creating an array with two more columns and two more rows than our data array. The data array is placed into the center of this new array, and the outer rows and columns are set to a value that is not specified in the *Levels* keyword. This will ensure that there are no open contours. To demonstrate with our DATA variable:

```
data2 = REPLICATE(-1.0, 8, 8)
```

DATA2 has two more rows and two more columns than DATA, and is filled with –1.0, which is not a value that will be specified as a contour level.

```
data2(1,1) = data
```

DATA is copied into the center of DATA2. The edges remain at –1.0.

Using DATA2, the following statements will produce a contour plot of DATA with the contours filled:

```
clev = [0.2, 0.5, 0.8]
```

Levels to contour.

```
cin= [192, 208, 224, 240]
```

Colors to fill with.

```
clab=[1, 1, 1]
```

Contours to Label (all three specified in clev).

```
CONTOUR, /SPLINE, LEVELS = clev, $
   C_LABEL=clab, PATH_FILENAME =  $
   'cpaths.dat', data2, xrange= [0, 7], $
   xstyle = 1, yrange = [0, 7], ystyle = 1
```

Create a file named cpaths.dat containing the contour paths. The range keywords avoid plotting the top and right border. The style keywords prevent PV-WAVE from rounding the plot range to a different value from that specified.

```
POLYCONTOUR, 'cpaths.dat', color_index=cin
```

Use POLYCONTOUR to fill the closed contours.

```
CONTOUR, /SPLINE, LEVELS = clev, C_LABEL = $
   clab, /NOERASE, data2, xrange = [0, 7], $
   xstyle = 1, yrange = [0, 7], ystyle = 1
```

Use CONTOUR a second time to draw the contours over the filled regions.

The result is shown in Figure 4-9.

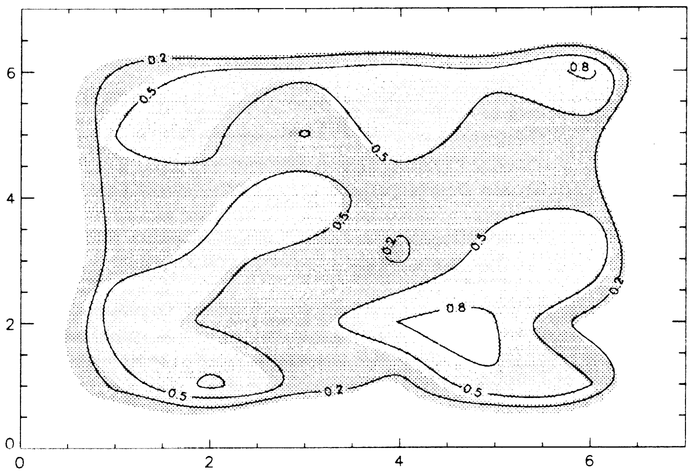


Figure 4-9 Filled contour plot with closed contours.

Drawing a Surface

The SURFACE procedure draws “wire mesh” representations of functions of X and Y, just as CONTOUR draws their contours. Parameters to SURFACE are similar to CONTOUR. SURFACE accepts a two-dimensional array of Z (elevation) values, and optionally *x* and *y* parameters indicating the location of each Z element.

SURFACE projects the three-dimensional array of points into two dimensions after rotating about the Z and then the X axes. Each point is connected to its neighbors by lines. Hidden lines are suppressed. The rotation about the X and Z axes can be specified with keywords, or a complete three-dimensional transformation matrix can be stored in the field !P.T, for use by SURFACE. Details concerning the mechanics of 3D projection and rotation are covered in the next sections.

The following PV‑WAVE code illustrates the most basic call to SURFACE. It produces a two-dimensional Gaussian function and then calls SURFACE to produce Figure 4-10:

```
z = SHIFT(DIST(40), 20, 20)
```

Create a 40-by-40 array and shift the origin to the center of the array.

```
SURFACE, EXP(-(z/10)^2)
```

Form a Gaussian with a 1/e width of 10, and call SURFACE to display it.

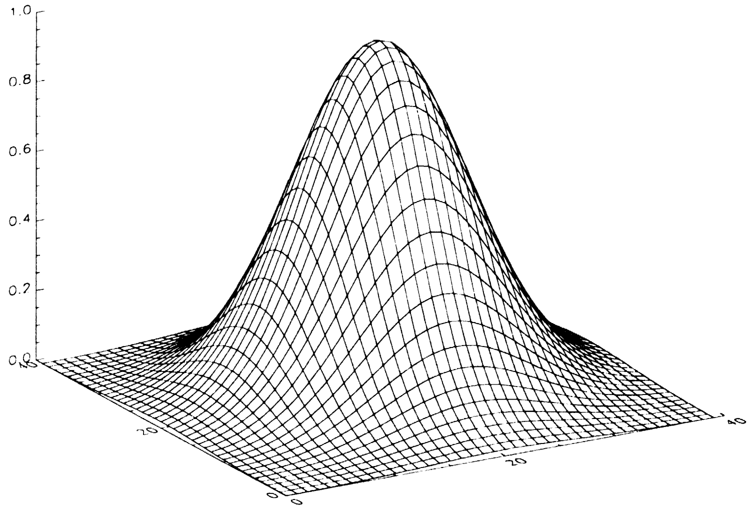


Figure 4-10 Simple SURFACE plot of a Gaussian.

In the above example, the DIST function creates an (*n, n*) array. DIST is a useful function for creating data, and is described in detail in the *PV‑WAVE Reference.*

Controlling Surface Features with Keywords

The following keywords are unique to, or have particular relevance to, the SURFACE procedure. For a complete list of the SURFACE keywords, see the description of SURFACE in the *PV‑WAVE Reference.*

Ax	Horizontal	Skirt
Az	Lower_Only	Upper_Only
Bottom	Save	ZAxis

For a detailed description of these keywords, see Chapter 3 *Graphics and Plotting Keywords*, in the *PV‑WAVE Reference*.

Example of Drawing a Surface

Figure 4-11 illustrates the application of the SURFACE procedure to the Maroon Bells data discussed earlier in this chapter (see *Drawing Contour Plots* on page 94).

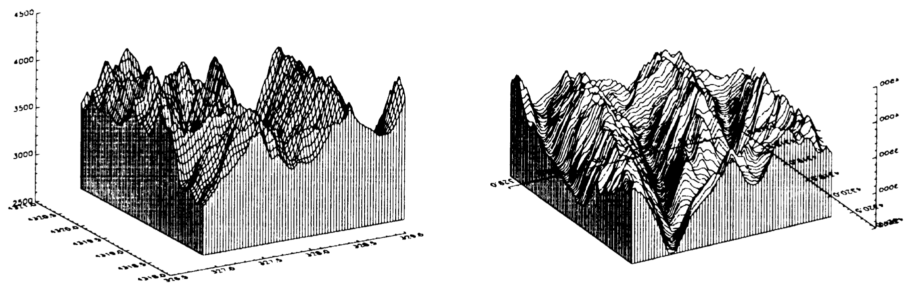


Figure 4-11 Maroon Bells surface plots.

The left illustration was produced by the following statements:

```
c = REBIN(a > 2650, 350/5, 460/5) $
SURFACE, c, x, y, SKIRT=2650
```

The first statement rebins the original data into a 70-by-92 array, as discussed in *Contouring Example* on page 96, while setting all missing data values (which are 0) to 2650, the lowest elevation we wish to show. As with CONTOUR, there can be too many data values, obscuring the surface with too much detail, and requiring more computation and drawing time.

The right illustration shows the Maroon Peaks area looking from the back row to the front row (north to the south), `AZ = 210`, and from a slightly steeper azimuth `AX = 45`. Also, only the horizontal lines are drawn because the *Horizontal* keyword assignment is present in the call:

```
SURFACE, c, x, y, SKIRT=2650, /HOR, AZ=210, $
   AX = 45
```

Because the axes were rotated 210 degrees about the original Z axis, the annotation is reversed and the X axis is behind and obscured by the surface. This undesirable effect can be eliminated by reversing the data array c about its Y axis. Also the y vector of element locations must be reversed, and the *YRange* keyword used to reverse the Y axis ordering.

```
SURFACE, reverse(c ,2), x, reverse(y), $
   Skirt=2650, /HOR, AX = 45, YRange = $
   [MAX(y), MIN(y)]
```

Perform as previously, but reverse the data rather than the axes.

Drawing Three-dimensional Graphics

Points in XYZ space are expressed by vectors of homogeneous coordinates. These vectors are translated, rotated, scaled, and projected onto the two-dimensional drawing surface by multiplying them by transformation matrices. The geometrical transformations used by PV-WAVE, and many other graphics packages, are taken from Chapters 7 and 8 of *Fundamentals of Interactive Computer Graphics* by J. D. Foley and A. Van Dam (Addison Wesley Publishing Co., 1982). Consult this book for a detailed description of homogeneous coordinates and transformation matrices, as this section presents only an overview.

Overview of Homogeneous Coordinates

A point in homogeneous coordinates is represented as a four-element column vector of three coordinates and a scale factor $w \neq 0$:

$$P(wx, wy, wz, w) \equiv P(x / w, y / w, z / w, 1) \equiv (x, y, z) \qquad (13.1)$$

One advantage of this approach is that translation, which normally must be expressed as an addition, may be represented as a matrix multiplication. Another advantage is that homogeneous coordinate representations simplify perspective transformations.

The notion of rows and columns used by PV-WAVE is opposite that of Foley and Van Dam. In PV-WAVE, the column subscript is the first, while in Foley and Van Dam the row subscript is first. This changes all row vectors to column vectors and transposes matrices.

PV-WAVE Uses a Right-handed Coordinate System

The coordinate system is right-handed so that when looking from a positive axis to the origin a positive rotation is counterclockwise. As usual, the X axis runs across the display, the Y axis is vertical, and the positive Z axis extends out from the display to the viewer. A 90 degree positive rotation about the Z axis transforms the X axis to the Y axis.

Overview of Transformation Matrices

For most applications, it is not necessary to create, manipulate, or to even understand transformation matrices. The T3D procedure, explained below, implements most of the common transformations.

Transformation matrices, which post-multiply a point vector to produce a new point vector, must be (4,4). A series of transformation matrices may be concatenated into a single matrix by multiplication. If A_1, A_2, and A_3 are transformation matrices to be applied in order, and the matrix A is the product of the three matrices:

$$((P \cdot A_1) \cdot A_2) \cdot A_3 \equiv P \cdot ((A_1 \cdot A_2) \cdot A_3) = P \cdot A$$

$$A = (A_1 \cdot A_2) \cdot A_3$$

PV-WAVE stores the concatenated transformation matrix in the system variable field !P.T.

Each of the operations of translation, scaling, rotation, and shearing may be represented by a transformation matrix.

Translating Data

The transformation matrix to translate a point by (D*x*, D*y*, D*z*) is:

$$\begin{bmatrix} 1 & 0 & 0 & \mathrm{D}x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling Data

Scaling by factors of S*x*, S*y*, and S*z*, about the X, Y and Z axes respectively is represented by the matrix:

$$\begin{bmatrix} \mathrm{S}x & 0 & 0 & 0 \\ 0 & \Sigma y & 0 & 0 \\ 0 & 0 & \Sigma z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotating Data

Rotation about the X, Y, and Z axes is represented respectively by the three matrices:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \chi o\sigma\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \chi o\sigma\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sigma\iota\nu\theta_z & \chi o\sigma\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using the T3D Procedure to Transform Data

The T3D procedure creates and accumulates transformation matrices, storing them in the system variable field !P.T. It can be used to create a transformation matrix composed of any combination of translation, scaling, rotation, perspective projection, oblique projection, and axis exchange.

Keywords that affect transformations are applied in the order of their description below:

- ❑ *Reset* – Resets the transformation matrix to the identity matrix to begin a new accumulation of transformations. If this keyword is not present, the current transformation matrix !P.T is post-multiplied by the new transformation. The final transformation matrix is always stored back in !P.T.
- ❑ *Translate* – Translates by the three-element vector $[T_x, T_y, T_z]$.
- ❑ *Scale* – Scales by factor $[S_z, S_y, S_z]$.
- ❑ *Rotate* – Rotates about each axis by the amount $[\theta_x, \theta_y, \theta_z]$, in degrees.
- ❑ *Perspective* – A scalar (p) indicating the Z distance of the center of the projection in the negative direction. Objects are projected into the XY plane, at Z = 0, and the "eye" is at point $(0, 0, -p)$.
- ❑ *Oblique* – A two-element vector, $[d, \alpha]$, specifying the parameters for an oblique projection. Points are projected onto the XY plane at Z = 0 as follows:

  $$x' = x + z(d \cos \alpha)$$

  $$y' = y + z(d \sin \alpha)$$

  An oblique projection is a parallel projection in which the normal to the projection plane is the Z axis, and the unit vector (0, 0, 1) is projected to $(d \cos \alpha, d \sin \alpha)$.
- ❑ *XYexch* – If set, exchanges the X and Y axes.
- ❑ *XZexch* – If set, exchanges the X and Z axes.
- ❑ *YZexch* – If set, exchanges the Y and Z axes.

An Example of Transformations Created by SURFACE

The SURFACE procedure creates a transformation matrix from its keyword parameters *AX* and *AZ* as follows:

- ❑ It translates the data so that the center of the normalized cube is moved to the origin.
- ❑ It rotates –90 degrees about the X axis to make the +Z axis of the data the +Y axis of the display. The +Y data axis extends from the front of the display to the rear.
- ❑ It rotates about the Y axis *AZ* degrees. This rotates the result counterclockwise as seen from above the page.
- ❑ It rotates about the X axis *AX* degrees, tilting the data towards the viewer.
- ❑ It then translates back to the origin and scales the data so that the data are still contained within the normal coordinate unit cube after transformation.

These transformations can be created using T3D as shown below. The SURFR (SURFace Rotate) procedure mimics the transformation matrix created by SURFACE using this method.

```
T3D, /Reset, Translate=[-.5, -.5, -.5]
```

Translate to move center of cube to origin.

```
T3D, Rotate=[-90, az, 0]
```

Rotate –90 degrees about X axis, so +Z axis is now +Y. Then rotate about Y axis AZ degrees.

```
T3D, Rotate=[ax, 0, 0]
```

Rotate AX about X axis.

```
SCALE3D
```

This procedure scales !P.T so that the unit cube still fits within the unit cube after transformation.

Converting from 3D to 2D Coordinates

To convert from a three-dimensional coordinate to a two-dimensional coordinate, PV-WAVE follows these steps:

- Data coordinates are converted to three-dimensional normalized coordinates. As described in *Coordinate System Conversion* on page 60, to convert the X coordinate from data to normalized coordinates:

$$N_x = X_0 + X_1 D_x$$

where X_1 is !X.S(*i*). The same process is used to convert the Y and Z coordinates using !Y.S and !Z.S.

- The three-dimensional normalized coordinate, $P = (N_x, N_y, N_z)$, whose homogeneous representation is $(N_x, N_y, N_z, 1)$, is multiplied by the concatenated transformation matrix !P.T:

$$P' = P \cdot !P.T$$

- The vector P' is scaled, as in Equation 13.1 on page 115, by dividing by *w*, and the normalized 2D coordinates are extracted:

$$N'_x = P'_x \,/\, P'_w \text{ and } N'_y = P'_y \,/\, P'_w$$

- The normalized XY coordinate is converted to device coordinates as described in *Coordinate System Conversion* on page 60.

This process can be written as a PV-WAVE function:

```
FUNCTION CVT_TO_2D, x, y, z
```

Accept a 3D data coordinate, return a two-element vector containing the coordinate transformed to 2D normalized coordinates using the current transformation matrix.

```
p = [ !x.s(0) + !x.s(1) * x, !y.s(0) + !y.s(1)$
    * y, !z.s(0) + !z.s(1) * z, 1]
```

Make a homogeneous vector of normalized 3D coordinates.

```
p = p # !P.T
```

Transform by !P.T.

```
RETURN, [ p(0) / p(3), p(1) / p(3) ]
```

Return the scaled result as a two-element, 2D, XY vector.

```
END
```

Establishing Your Own 3D Coordinate System

Usually, scaling parameters for coordinate conversion are set up by the higher-level plotting procedures. To set up your own 3D coordinate system with a given transformation matrix and X, Y, Z data range, follow these steps:

- Establish the scaling from your data coordinates to normalized coordinates — the (0,1) cube. Assuming your data are contained in the range (X_{min}, Y_{min}, Z_{min}) to (X_{max}, Y_{max}, Z_{max}), set the data scaling system variables as follows:

  !X.S = [$-X_{min}$, 1] / (X_{max} - X_{min})
  !Y.S = [$-Y_{min}$, 1] / (Y_{max} - Y_{min})
  !Z.S = [$-Z_{min}$, 1] / (Z_{max} - Z_{min})

- Establish the transformation matrix which determines the view of the unit cube. This can be done by either calling T3D, explained above, or by directly manipulating !P.T yourself. If you wish to simply mimic the rotations provided by the SURFACE procedure, call the SURFR procedure.
- Call the SCALE3D procedure to re-scale the projected unit cube back to the (0,1) 2D normalized coordinate square. SCALE3D transforms a unit cube by the current !P.T and uses the extrema of each axis to translate and rescale the result back to the unit square.

Example of Data Transformations

This example draws four views of a simple house. The procedure HOUSE defines the coordinates of the front and back faces of the house. The data to normal coordinate scaling is set, as shown above, to a volume about 25% larger than that enclosing the house. The PLOTS procedure is called to draw lines describing and connecting the front and back faces. XYOUTS is called to label the front and back faces.

The main program contains four sequences of calls to T3D to establish the coordinate transformation, followed by a call to SCALE3D to center the transformed unit cube in the viewing area, and then by a call to HOUSE.

Note Remember that a valid data coordinate system must be established before calling PLOTS. This coordinate system can be established by a call to PLOT, or by explicitly setting values of the system variables !X, !Y, and !Z.

Procedure Used to Draw a House

```
PRO HOUSE
```
Define a procedure to draw a house.

```
house_x = [0, 16, 16, 8, 0, 0, 16, 16, 8, 0]
```
X coordinates of 10 vertices. First 5 are front face, second 5 are back face. Range is 0 to 16.

```
house_y = [0, 0, 10, 16, 10, 0, 0, 10, 16, 10]
```
Corresponding Y values. Range is 0 to 16.

```
house_z = [54, 54, 54, 54, 54, 30, 30, 30, 30,$
   30]
```
Z values, from 30 to 54.

```
!X.S = [-(-4), 1.)] / (20 - (-4))
```
Set X data scale to range from –4 to 20.

```
!Y.S = !x.s
```
Same for Y.

```
!Z.S = [-10, 1. ] / (70 - 10)
```
Z range is from 10 to 70.

```
face = [INDGEN(5), 0]
```
Indices of front face.

```
PLOTS, house_x(face), house_y(face), $
   house_z(face), /T3D, /DATA
```
Draw front face.

```
PLOTS, house_x(face+5), house_y(face+5),$
   house_z(face+5), /T3D, /DATA
```
Draw back face.

```
FOR i=0, 4 DO PLOTS, [house_x(i),$
    house_x(i+5)], [house_y(i),$
    house_y(i+5)], [house_z(i),$
    house_z(i+5)], /T3D, /DATA
```

Connecting lines from front to back.

```
XYOUTS, house_x(3), house_y(3), $
   Z=house_z(3), 'Front', /T3D, /DATA, $
   SIZE=2
```

Annotate front peak.

```
XYOUTS, house_x(8), house_y(8), $
   Z=house_z(8), 'Back', /T3D, /DATA, $
   SIZE=2
```

Annotate back.

```
END
```

End of HOUSE procedure.

Commands that Perform Transformations on the House

```
T3D, /reset & SCALE3D & house
```

Set up no rotation, scale, and draw house.

```
T3D, /reset, rot=[30, 30, 0] & SCALE3D & HOUSE
```

Straight projection after rotating 30 degrees about X and the Y axes.

```
T3D, /reset, rot=[0, 0, 0], oblique=[.5, -45]$
   & SCALE3D & HOUSE
```

No rotation, oblique projection, Z factor = 0.5, angle = 45.

```
T3D, /reset, rot=[0, 0, 0], perspective=4 $ &
   SCALE3D & HOUSE
```

No rotation, perspective at 4.

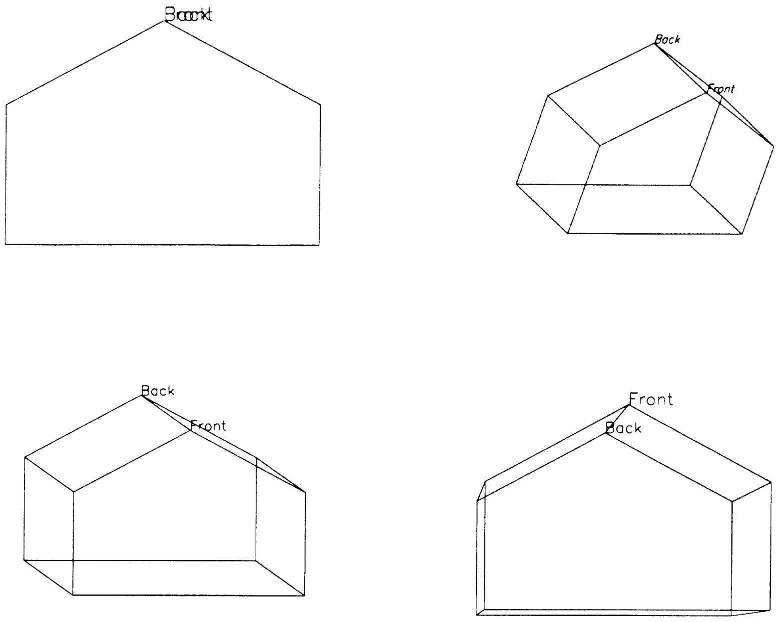


Figure 4-12 Illustration of different 3D transformations. From upper left: No rotation, plain projection; Rotation of 30 degrees about both the X and Y axes, plain projection; Oblique projection, factor = 0.5, angle = –45; and in the bottom right, 30 degrees rotation with the eye at 50.

3D Transformations with 2D Procedures

The CONTOUR and PLOT procedures output their results using the three-dimensional coordinate transformation contained in !P.T, if the keyword *T3d* is specified

Note !P. T must contain a valid transformation matrix prior to using the *T3d* keyword.

PLOT and its variants output graphs in the XY plane at the normal coordinate Z value given by the keyword *ZValue*. If this keyword is not specified, the plot is drawn at the bottom of the unit cube, at Z=0.

CONTOUR draws its axes at Z=0, and its contours at their Z data value if *ZValue* is not specified. If *ZValue* is present, CONTOUR draws both the axes and contours in the XY plane at the given Z value.

Combining CONTOUR and SURFACE Procedures

It is easy to combine the results of SURFACE with the other PV-WAVE graphics procedures. The keyword parameter *Save* causes SURFACE to save the graphic transformation it used in !P.T. Then, when CONTOUR or PLOT are called with the keyword parameter *T3d*, their output is transformed with the same projection.

For example, Figure 4-13 illustrates SURFACE combined with CONTOUR. In essence, this a combination of Figure 4-2 and Figure 4-11. Using the same variables as discussed in *Drawing Contour Plots* on page 94 and *Drawing a Surface* on page 112, this figure was produced with the following statements:

```
SURFACE, c, x, y, SKIRT=2650, /Save
```

Make the mesh as in Figure 4-11.

```
CONTOUR, b, x, y, /T3d, /Noerase, Title= $
   'Contour Plot', Max_val=5000., Zvalue= $
   1.0, /Noclip, Levels = 2750. + $
   FINDGEN(6)*250
```

Make the Contour plot as in Figure 4-2. Specify T3D to align with Surface, at ZVALUE of 1.0. Suppress clipping as the plot is outside the normal plot window.

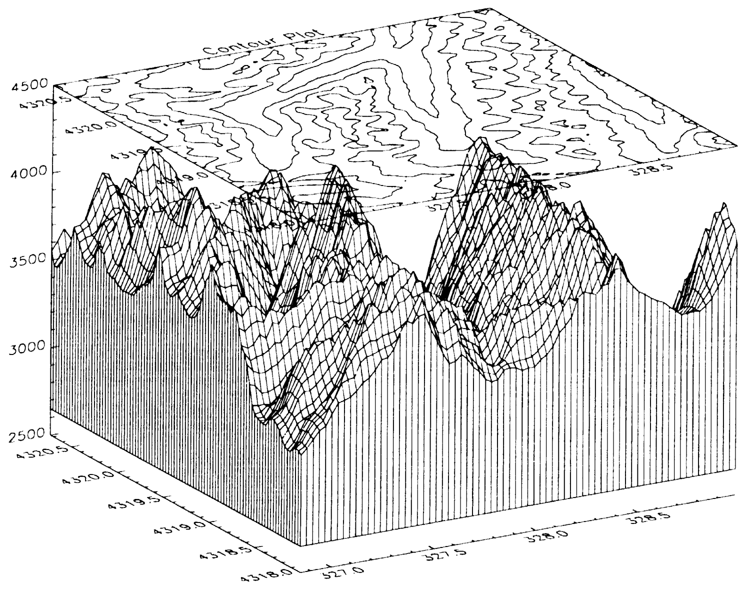


Figure 4-13 Combining CONTOUR with SURFACE, Maroon Bells data.

Even More Complicated Transformations are Possible

Figure 4-14 illustrates the application of three-dimensional transforms to the output of CONTOUR and PLOT. It shows a three-dimensional contour plot with the contours stacked above the axes in the Z direction, the sum of the columns, also a Gaussian, in the XZ plane, and the sum of the rows in the YZ plane.

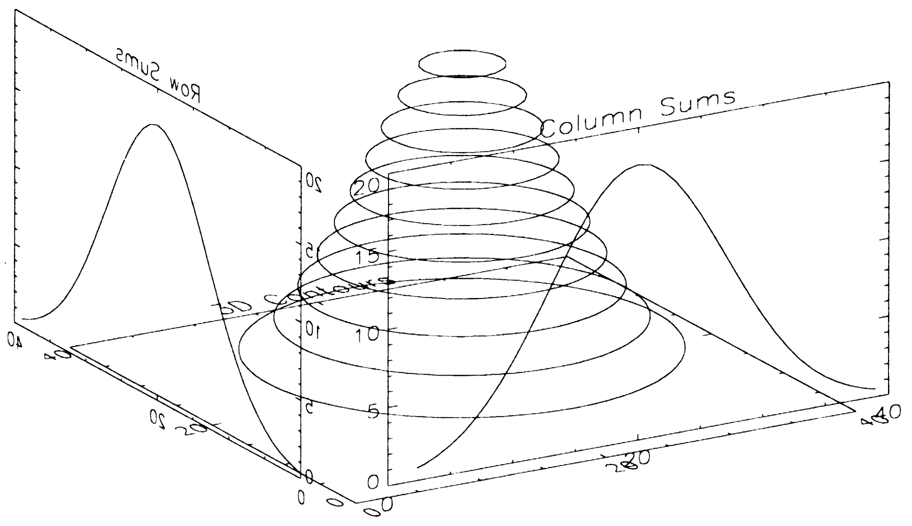


Figure 4-14 Example of using PLOT and CONTOUR with a three- dimensional transform.

The PV-WAVE code used to draw Figure 4-14 is:

```
nx=40
temp = SHIFT(DIST(40), 20, 20)
z = EXP(-(temp/10)^2)
```

Create a 2D Gaussian array, z

```
SURFR
```

Set up !P.T with default SURFACE transformation.

```
pos = [.1, .1, 1, 1, 0, 1]
```

Define the 3D plot window. X = .1 to 1, Y = .1 to 1, 1 and Z = 0 to 1.

```
CONTOUR, z, /T3D, NLEVELS=10, /NOCLIP, $
   POSITION=pos, CHARSIZE=2
```

Make the stacked contours. Use 10 contour levels.

```
T3D, /YZEXCH
```

Swap Y and Z axes. The original XYZ system is now XZY.

```
PLOT, z # REPLICATE( 1., nx), /Noerase, $
   /Noclip, /T3d, Title='Column Sums', $
   Position=pos, Charsize=2
```

Plot the column sums in front of the contour plot.

```
T3D, /Xzexch
```

Swap X and Z, original XYZ is now YZX.

```
PLOT, REPLICATE( 1., nx) # z, /Noerase, /T3D,$
   /Noclip, Title='Row Sums', Position=pos,$
   Charsize=2
```

Plot the row sums along the right side of the contour plot.

The basic steps are:

- ❑ First, the SURFR procedure is called to establish the default three-to two-dimensional transformation used by SURFACE, as explained above. The default rotations are 30 degrees about both the X and Z axes.
- ❑ Next, a vector, `pos`, defining the cube containing the plot window is defined with normalized coordinates. The cube extends from 0.1 to 1.0 in the X and Y directions, and from 0 to 1 in the Z direction. Each call to CONTOUR and PLOT must explicitly specify this window to align the plots. This is necessary because the default margins around the plot window are different in each direction.
- ❑ CONTOUR is called to draw the stacked contours with the axes at Z=0. Clipping is disabled to allow drawing outside the default plot window which is only two-dimensional.
- ❑ The T3D procedure is called to exchange the Y and Z axes. The original XYZ coordinate system is now XZY.
- ❑ PLOT is called to draw the column sums which appear in front of the contour plot. The expression:

  ```
  z # REPLICATE(1., nx)
  ```

  creates a row vector containing the sum of each column in the two-dimensional array `z`. The *Noerase* and *Noclip* keywords are specified to prevent erasure and clipping. This plot appears in the XZ plane because of the previous axis exchange.

- ❑ T3D is called again to exchange the X and Z axes. This makes the original XYZ coordinate system, which was converted to XZY, now correspond to YZX.
- ❑ PLOT is called to produce the row sums in the YZ plane in the same manner as the first plot. The original X axis is drawn in the Y plane, and the Y axis is in the Z plane. One unavoidable side effect of this method is that the annotation of this plot is backwards. If the plot is transformed so the letters read correctly, the X axis of the plot would be reversed in relation to the Y axis of the contour plot.

Combining Images with 3D Graphics

Images are combined with 3D graphics, as shown in Figure 4-15, using the transformation techniques described in the previous section.

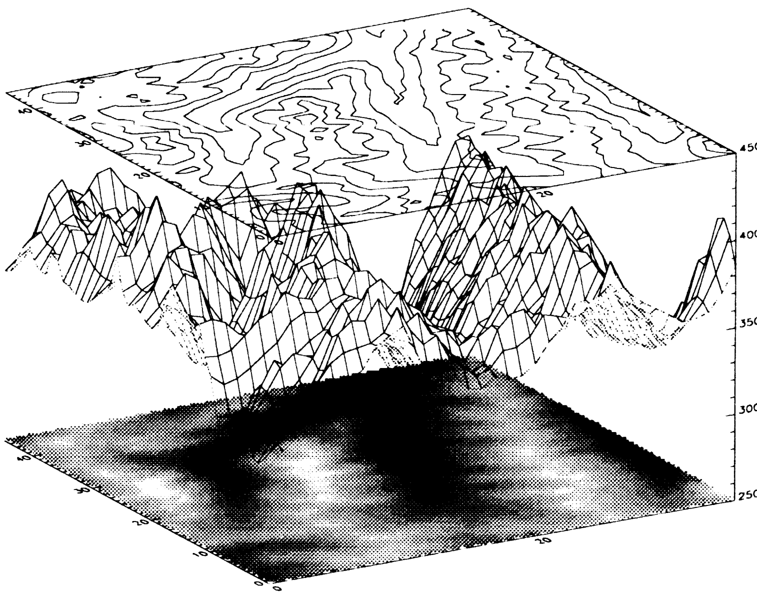


Figure 4-15 Using the SHOW3 procedure to overlay an image, surface mesh, and contour.

The rectangular image must be transformed so that it fits underneath the mesh drawn by SURFACE. The general approach is as follows:

- ❑ Use SURFACE to establish the general scaling and geometrical transformation. Draw no data, as the graphics made by SURFACE will be over-written by the transformed image.
- ❑ For each of the four corners of the image, translate the data coordinate, which is simply the subscript of the corner, into a device coordinate. The data coordinates of the four corners of an (m, n) image are (0,0), $(m - 1, 0)$, $(0, n - 1)$, and $(m - 1, n - 1)$. Call this data coordinate system (X, Y). Using a procedure or function similar to CVT_TO_2D, described on page 120, convert to device coordinates, which in this discussion are called (U, V).
- ❑ The image is transformed from the original *XY* coordinates to a new image in *UV* coordinates using the POLY_2D function. POLY_2D accepts an input image and the coefficients of a polynomial in *UV* giving the *XY* coordinates in the original image. The equations for *X* and *Y* are:

  $$X = S_{0,0} + S_{0,1}U + S_{1,0}V + S_{1,1}UV$$

  $$Y = T_{0,0} + T_{0,1}U + T_{1,0}V + T_{1,1}UV$$

  We solve for the four unknown *S* coefficients using the four equations relating the *X* corner coordinates to their *U* coordinates. The *T* coefficients are similarly found using the *Y* and *V* coordinates. This can be done using matrix operators and inversion or, more simply, with the POLYWARP procedure.
- ❑ The new image is a rectangle which encloses the quadrilateral described by the *UV* coordinates. Its size is:

  $$(\max(U) - \min(U) + 1, \max(V) - \min(V) + 1)$$
- ❑ POLY_2D is called to form the new image which is displayed at device coordinate $(\min(U), \min(V))$.
- ❑ SURFACE is called once again to display the mesh surface over the image.

- ❑ Finally, CONTOUR is called, with *ZValue* set to 1.0, placing the contour above both the image and the surface.

The SHOW3 procedure performs these operations. Look at the code for the SHOW3 procedure in the Standard Library for details of how images and graphics can be combined.

Drawing Shaded Surfaces

The SHADE_SURF procedure creates a shaded representation of a surface made from regularly gridded elevation data. The shading information may be supplied as a parameter or computed using a light source model. Displays are easily constructed depicting the surface elevation of a variable shaded as a function of itself or another variable. This procedure is similar to the SURFACE routine, but it renders the visible surface as a shaded image rather than a mesh.

Parameters are identical to those of the SURFACE procedure, described in the section *Drawing a Surface* on page 112, with the addition of two optional keyword parameters:

Shades – Specifies an array of the same dimensions as the Z parameter, which contains the shading color indices. This array should be scaled into the range of color indices, normally 0 to 255.

Image – Specifies the name of a variable into which the image created by SHADE_SURF is placed. Normally, the image is displayed on the currently selected graphics device and then discarded.

Alternative Shading Methods

The shading applied to each polygon, defined by its four surrounding elevations, may be either *constant* over the entire cell, or *interpolated.* Constant shading is faster because only one shading value needs to be computed for the entire polygon. Interpolated shading gives smoother, usually more pleasing, results. The Gouraud method of interpolation is used: the shade values are computed at each elevation point, coinciding with each polygon vertex; then the shading is interpolated along each edge; and finally between edges along each vertical scan line.

Light source shading is computed using a combination of depth cueing, ambient light, and diffuse reflection, adapted from Chapter 16 of *Fundamentals of Computer Graphics*, Foley and Van Dam:

$$I = Ia + dI_p(L \cdot N)$$

where:

I_a is the term due to ambient light. All visible objects have at least this intensity, which is approximately 20% of the maximum intensity.

$I_p(L \cdot N)$ is the term due to diffuse reflection. The reflected light is proportional to the cosine of the angle between the surface normal vector N, and the vector pointing to the light source L. I_p is approximately 0.9.

d is the term for depth cueing, causing surfaces further away from the observer to appear dimmer. $d = (z+2)/3$, where z is the normalized depth, ranging from zero for the most distant point, to one for the closest.

Setting the Shading Parameters

Parameters affecting the method of shading interpolation, light source direction, and rejection of hidden faces are set with the SET_SHADING procedure, described in the *PV-WAVE Reference*. Defaults are: Gouraud interpolation, light source direction is [0, 0, 1], and rejection of hidden faces enabled.

See the description of SET_SHADING in *PV-WAVE Reference* for a more complete description of the parameters. Note that the *Reject* keyword has no effect on the output of SHADE_SURF – it is used only with solids.

Sample Shaded Surfaces

The left side of Figure 4-16 illustrates the application of SHADE_SURF, with light source shading, to the two-dimensional Gaussian, *z*, used to produce Figure 4-10 on page 113. This figure was produced by the statement:

```
SHADE_SURF, z
```

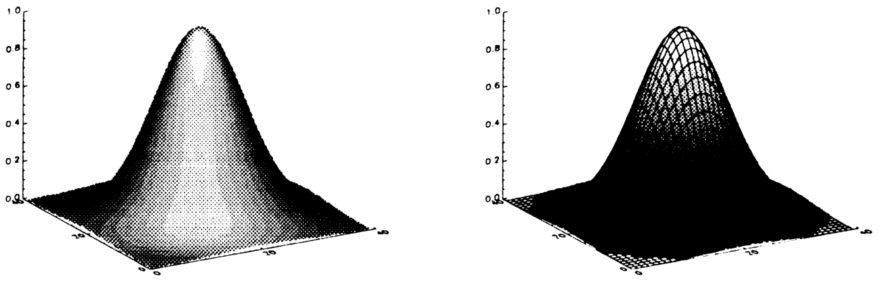


Figure 4-16 Shaded representations of two-dimensional Gaussian.

The right half of Figure 4-16 shows the use of an array of shades, which in this case is simply the surface elevation scaled into the range of bytes. The output of SURFACE is superimposed over the shaded image with the statements:

```
SHADE_SURF, z, SHADE=BYTSCL(z)
```

Show Gaussian with shades created by scaling elevation into the range of bytes.

```
SURFACE, z, XST=4, YST=4, ZST=4, /NOERASE
```

Draw the mesh surface over the shaded figure. Suppress the axes.

Figure 4-17 shows the Maroon Bells data, also shown in the right half of Figure 4-11 on page 114, as a light source shaded surface. It was produced by the statement:

```
SHADE_SURF, b, x, y, AZ=210, AX=45, XST=4,$
   YST=4, ZST=4
```

The *AX* and *AZ* keywords specify the orientation. The axes are suppressed by the axis style keyword parameters, as in this orientation the axes are behind the surface.

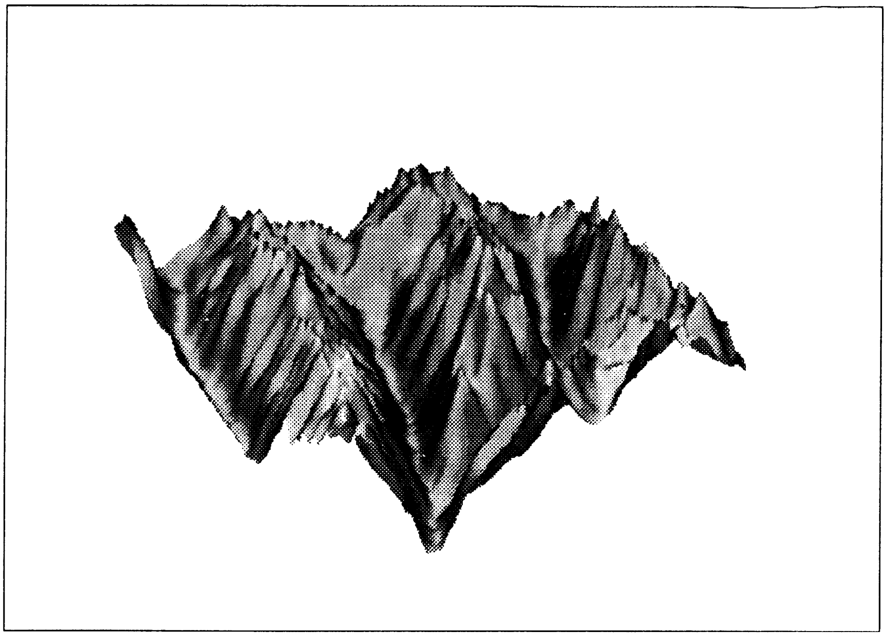


Figure 4-17 Maroon Bells data shown as a shaded surface.

5

CHAPTER

Displaying Images

PV-WAVE is a powerful environment for image processing and display. The routines described in this chapter are the interface between PV-WAVE and the image display system. The first part of this chapter describes how images are displayed and controlled. The second part describes a few of the simple ways images can be transformed or processed.

What is an Image?

An image is a two-dimensional array of pixels. The value of each pixel represents the intensity and/or color of that position in the array. Images of this form are known as sampled or raster images, because they consist of a discrete grid of samples. Such images come from many sources and are a common way of representing scientific and medical data.

Working with Images

The following PV-WAVE routines are used to display and control images.

TV Procedure

Displays images on the image display.

TVSCL Procedure

Scales the intensity values of the image into the range of the display device, and then displays the result.

TVRD Function

Reads image pixels back from the display device.

TVCRS Procedure

Manipulates the image device cursor. TVCRS allows the cursor to be enabled and disabled, and allows it to be positioned.

TVLCT Procedure

Loads a user-defined color table into the display device.

LOADCT Procedure

Loads a predefined color table into the display device.

These routines are described further in this chapter, and also in the *PV-WAVE Reference*. Many other routines that are useful in viewing and processing images are also introduced in this chapter, such as REBIN, CONGRID, SMOOTH, HIST_EQUAL, MEDIAN, CONVOL and many others.

In addition, most plotting and graphics routines can be used with images. These routines are described in Chapter 3, *Displaying 2D Data* and Chapter 4, *Displaying 3D Data*. For example, you can overlay an image on a contour plot by combining the output of the CONTOUR and TV procedures. Or, the CURSOR routine, which is ordinarily used to read the position of the interactive pointer device, can be used to determine the location of image pixels.

Image Display Routines: TV and TVSCL

The TV and TVSCL procedures display images on the screen. They take the same arguments and keywords, and differ only in that TVSCL scales the image into the intensity range of the display device, while TV displays the image directly.

Usage

TV, *image* [, *position*]

TV, *image* [, *x*, *y* [, *channel*]]

TVSCL, *image* [, *position*]

TVSCL, *image* [, *x*, *y* [, *channel*]]

Parameters

image – A vector or two-dimensional array to be displayed as an image. If it is not already of byte type, it is converted before it is used.

x, *y* – If *x* and *y* are present, they specify the lower-left coordinate of the displayed image. The default is the device coordinate system.

position – Position number of the image. Image positions are discussed in detail in this section.

channel – Some image display devices are capable of storing more than a single image or can combine three single color images to form a true color image. *channel* specifies the memory channel to be written. It is assumed to be zero if not specified. This parameter is ignored on display systems that have only one memory channel.

Keywords

If no optional parameters are present, the image is output to the display with its lower-left corner at window or display coordinate (0, 0). The optional keyword parameters may be used to specify the screen position of the image in a variety of ways. For details on the keywords for a particular routine, see the routine's description in the *PV-WAVE Reference*.

Image Orientation on the Display Screen

The coordinate system of the image display screen is oriented with the origin, (0, 0), in the lower-left corner. The upper-right-hand corner has the coordinate $(X_{size} - 1, Y_{size} - 1)$, where X_{size} and Y_{size} are the dimensions of the visible area of the window or display. The descriptions of the image display routines that follow assume a window size of 512-by-512, although other sizes may be used.

!Order is a system variable that controls the order in which the image is written to the screen. Images are normally output with the first row at the bottom (i.e., in bottom to top order), unless !Order is one, in which case, images are written on the screen from top to bottom. The !Order keyword can also be specified with TV and TVSCL. It works in the same manner as !Order except that its effect only lasts for the duration of the single call — the default is that specified by !Order.

An image may be displayed with any of the eight possible combinations of axis reversal and transposition by combining the display procedures with the ROTATE function.

Image Position on the Display Screen

Image positions run from the left of the screen to the right and from the top of the screen to the bottom. If a position number is used instead of X and Y, the position of the image is calculated from the dimensions of the image as follows:

X_{size}, Y_{size} = size of display or window

X_{dim}, Y_{dim} = dimensions of array

$N_x = X_{size} / X_{dim}$ = number of images across the screen

$X = X_{dim} Position_{\mathrm{modulo}Nx}$ = starting X value

$Y = Y_{size} - Y_{dim}[1 + Position / N_x]$ = starting Y value

For example, when displaying 128-by-128 images on a 512-by-512 window or display, the position numbers run from 0 to 15 as follows:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

Image Size

Most image devices have a fixed number of display pixels. Common sizes are 512 x 512, and 1280 x 1024. Such pixels have a fixed size which cannot be changed. For such devices, the area written on the screen is the same size as the dimensions of the image array. One-dimensional vectors are considered as row vectors. The X and Y parameters specify the coordinates of the lower-left corner of the area written on the display.

There are some devices, however, that can place an image with any number of pixels into an area of arbitrary size. PostScript devices are a notable example. These devices are said to have scalable pixels, because there is no direct connection between the number of pixels in the image and the physical space it occupies in the displayed image. When the current image device has scalable pixels, PV-WAVE sets the first bit of !D.Flags. The following PV-WAVE statement can be used to determine if the current device has scalable pixels:

```
SP = !D.Flags AND 1
```

`SP` will be nonzero if the device has scalable pixels. When displaying an image on a device with scalable pixels, the default is to use the entire display surface for the image. The *XSize* and *YSize* keywords can be used to override this default and specify the width and height that should be used.

The *XSize* and *YSize* keywords should also be used when positioning images with the *position* argument to TV or TVSCL. Position normally uses the size of the image in pixels to determine the placement of the image, but this is not possible for devices with scalable pixels. Instead, the default for such devices is to assume a single position that fills the entire available display surface. However, if *Xsize* and *Ysize* are specified, *Position* will use them to determine image placement.

Examples

```
TV, REPLICATE(100B, 512, 512)
```
Set all display memory to 100.

```
ABC = BYTARR(50,100)
```
Define a 50-column by 100-row array.

```
TV, ABC, 300, 400
```
Display array ABC starting at location x=300, y=400. Display pixels in columns 300 to 349, and rows 400 to 499 are zeroed.

```
TV, ABC/2, 12
```
Display image divided by 2 at position number 12.

```
TV, ABC, 256, 256, 2
```
Output image to memory channel 2, lower-left corner at (256, 256).

```
AA = ASSOC(1, BYTARR(64,64))
```
Assume file one contains a sequence of 64-by-64 byte arrays

```
FOR I=0, 63 DO TV, AA(I), I
```
Display 64 images from file, from left to right and top to bottom, filling a 640-by-512 area.

Image Magnification and Reduction

The size of the area written on the screen, measured in pixels, is identical to the dimensions of the image expression. Some output devices have hardware zoom and pan, which can blow up small images to full screen.

Note For PostScript output, the size of a pixel may be varied, eliminating the need for zoom, pan, or software resampling.

Other displays, including most of those with window systems, have no hardware zoom. On these displays, images must be magnified in software. The REBIN and CONGRID functions provide two ways to magnify or reduce an image to an arbitrary size.

Use REBIN for Integral Multiples (or Factors) of Images

With REBIN, the final dimensions must be integral multiples or factors of the original dimensions.

The call to REBIN is:

new_image = REBIN*(old_image, cols, rows, /Sample)*

where *old_image* is the array expression to be resampled, *cols* and *rows* specify the size of the result and are integral multiples or factors of the original dimensions, and the keyword parameter *Sample* is set to use "nearest neighbor" sampling. If *Sample* is not set, REBIN uses bilinear interpolation when magnifying, and neighborhood averaging when reducing. Bilinear interpolation avoids the "chunky" appearance of magnified pixels but takes more computer time.

For example, to display a 64-by-64 image named `area`, in a 512-by-512 pixel area using bilinear interpolation:

```
TV, REBIN(area, 512, 512)
```

or without bilinear interpolation:

```
TV, REBIN(area, 512, 512, /Sample)
```

When reducing by a factor of *n*-by-*m*, REBIN averages each *n*-by-*m* pixel neighborhood. An image may be magnified along one dimension, while at the same time be reduced along the other dimension. For more information on REBIN, see the *PV-WAVE Reference.*

Use CONGRID for Arbitrary Multiples (or Factors) of Images

CONGRID works in a similar fashion, except that the final dimensions can be any arbitrary size. The call to CONGRID using bilinear interpolation in the resampling algorithm is:

new_image = CONGRID*(old_image, cols, rows, /Interp)*

where the parameters *cols* and *rows* specify the number of columns and rows desired in the output image. If the *Interp* keyword is not set (i.e., it is equal to 0), the nearest neighbor sampling method is used instead. For more information on CONGRID, see the *PV-WAVE Reference.*

The ZOOM Function

There is one other way to magnify an image. On a window system display, the contents of a window (or image) centered about the mouse position can be magnified with the ZOOM procedure. For more information on ZOOM, see the *PV-WAVE Reference.*

Retrieving Information from Images

Reading Images from the Display Device

The TVRD function reads the contents of the display device memory back into a PV-WAVE variable. One use for this capability is to build up a complex display using many PV-WAVE statements, and then read the resulting image back as a single unit for storage in a file.

The TVRD function returns the contents of the specified rectangular portion of the display subsystem's memory. For example, if (X_0 and Y_0) are the starting column and row, respectively, of the data to be read, and *Nx* and *Ny* are the number of columns and rows, respectively, to read, then an *Nx*-by-*Ny* byte array can be stored in the variable *new_image* with the command:

new_image = TVRD(X_0, Y_0, *Nx*, *Ny*)

If the system variable !Order is set to 0 then data are read from the bottom up, otherwise data are read from the top down.

Example of How to Use the TVRD Function

The following statement inverts the 100-by-100 area of the display starting at coordinate position (200, 300):

```
TV, NOT TVRD(200, 300, 100, 100)
```
Reverse area.

Not All Devices can Read from the Display

Not all image devices are able to support reading pixels back from device memory. If the current device has this ability, PV-WAVE sets the eighth bit of !D.Flags. The following PV-WAVE statement can be used to determine if the current device allows reading from display memory:

```
TEST = !D.Flags AND 128
```

TEST will be nonzero if the device allows such operations.

Using the Cursor with Images: TVCRS

The TVCRS procedure manipulates the cursor of the image display. Normally, the cursor is disabled and is not visible. Supplying TVCRS with one parameter enables or disables the cursor; supplying TVCRS with two parameters enables the cursor and places it on pixel location (X, Y).

Usage

TVCRS *[, on_off]*

TVCRS*[, x, y]*

Parameters

on_off – Specifies whether the cursor should be on or off. If present and is nonzero, the cursor is enabled. If *on_off* is zero or no parameters are specified, the cursor is turned off.

x – The column to which the cursor will be set.

y – The row to which the cursor will be set.

Keywords

TVCRS also takes various keywords that affect how it positions the cursor. Notably, the keywords *Data*, *Device*, and *Normal* specify the coordinate system. For details, see the entry for TVCRS in the *PV-WAVE Reference*.

Using Color with Images

Color is a valuable aid in the visual analysis process, because it can be used to take advantage of the human brain's capability to distinguish fine gradations of shade and intensity. For this reason, color plays a very important role when viewing images.

For color and gray-scale devices, the default is to display 8-bit images using the PV-WAVE color table B–W Linear (standard color table number 0). On a monochrome display, by default, color images are dithered. For more information about dithering, see *Displaying Images on Monochrome Devices* on page 148.

Color Systems

Most devices capable of displaying color use the RGB (red, green, blue) color system. By default, PV-WAVE represents images in the RGB color system using triplets of values for the red, the green, and the blue components of a particular pixel's color.

For more information about how image data is stored and transferred, refer to *Input and Output of Image Data* on page 189 of the *PV-WAVE Programmer's Guide.*

If you are interested in seeing a more complete discussion of color systems, refer to *Understanding Color Systems* on page 305.

Using Color Tables to View Images

PV-WAVE provides two commands for viewing images: TV and TVSCL. These two commands were introduced earlier in section *Image Display Routines: TV and TVSCL* on page 136.

By default, images are displayed using color table number 0, B–W Linear. To use a color table other than the default, load it prior to displaying the image, as shown in the following example:

```
LOADCT, 5
```

Load the predefined color table number 5, Standard Gamma-II.

```
TV, image
```

```
-or-

TVSCL, image
```

Display the image using either the TV or the TVSCL command.

PV-WAVE includes an assortment of 16 predefined color tables with enough variety to produce visually pleasing results for many applications, or you can define your own color table. To see a list of the color tables that come standard with PV-WAVE, refer to *Loading a Predefined Color Table: LOADCT* on page 312.

Loading a Different Color Table

Most color workstations cannot display more than a certain number of colors (usually 256) at once. For this reason, color tables are used to map red, green, and blue values into the available colors on the workstation.

You can use either the TVLCT or the LOADCT procedures to load the color table on the current device:

- **LOADCT** – This procedure loads predefined color tables stored in the file `colors.tbl` (found in `$WAVE_DIR/bin`, one portion of the main PV-WAVE install area).
- **TVLCT** – This procedure loads color tables stored in user-defined variables. Once the variables are loaded into the color table, it is used like any other color table.

For more information about loading the various color tables, see *Experimenting with Different Color Tables* on page 310. For more information about creating custom color tables that emphasize some special trend or effect, see *Modifying the Color Tables* on page 314.

Color Tables for Viewing Images

Be sure to experiment with the sixteen color tables that are included with PV-WAVE. Frequently, a trend that is "hidden" when viewing an image with one color table stands out with clarity when viewing the image with another color table.

The color tables that work best for viewing images are the ones that do not have sudden transitions from one color table index to the next. Otherwise, you will probably see a strong banding or "contouring" effect that is created by the rapid transitions between colors.

For an example of how to de-emphasize and moderate the color transitions in a color table, refer to *Smoothing the Color Transitions in a Color Table* on page 318.

Not all Color Images are True-color Images

Images may be output with 1, 2, 3, 4 or 8 bits per pixel, yielding 1, 2, 16, or 256 possible colors. In addition, color images are either: 1) pseudo-color or 2) true color. These two approaches to storing image information are contrasted in the following sections.

Note Not all output devices allow you to control the number of bits used to represent each pixel. To see if your device supports this capability, refer to Appendix A, *Output Devices and Window Systems.*

Pseudo-color Images

A pseudo-color image is a two-dimensional image, each pixel of which is used to index the color table, thereby obtaining an RGB value for each possible pixel value. An 8-bit workstation monitor usually displays pseudo-color images.

Note In the case of pseudo-color images of less than 8 bits, the number of columns in the image should be an exact multiple of the number of pixels per byte. In other words, when displaying a 2-bit image the number of columns should be even, and 4-bit images should have a number of columns that is a multiple of 4. If the image column size is not an exact multiple, extra pixels with a value of 255 are output at the end of each row. This causes no problems if the color white is loaded into the last color table entry, otherwise a stripe of the last (index number 255) color is drawn to the right of the image.

True-color Images

A true-color image consists of an array with three dimensions, one of which has a size of three, containing the three color components. It may be considered as three two-dimensional images, one each for the red, green, and blue components. For example a true-color *n*-by-*m* element image can be ordered in three ways: *pixel interleaved* (3, *n*, *m*), *row interleaved* (*n*, 3, *m*), or *image interleaved* (*n*, *m*, 3). By convention the first color is always red, the second green, and the last is blue.

True-color images are routed through the color table, just like pseudo-color images. The red color table array contains the intensity translation table for the red image, and so forth. Assuming that the color table has been loaded with the vectors R, G, and B, a pixel with a color value of (*r*, *g*, *b*) is displayed with a color of (R_r, G_g, B_b). A color table value of 255 represents maximum intensity, while 0 indicates an absence of the color.

To pass the RGB pixel values without change, load the red, green, and blue color tables with a ramp with a slope of 1.0:

```
TVLCT, INDGEN(256), INDGEN(256), INDGEN(256)
```

or with the LOADCT procedure:

```
LOADCT, 0
```

Load the standard black/white color table, B–W Linear.

Use the *True* keyword of the TV and TVSCL procedures to indicate that the image is a true-color image and to specify the dimension over which color is interleaved. Allowed values are:

1	pixel interleaving
2	row interleaving
3	image interleaving

Tip Image interleaving is also known as band interleaving.

To see specific examples showing how to use the *True* keyword, see the examples in the section *Displaying Images on Monochrome Devices* on page 148.

For more information about the different ways that image data may be stored, refer to *Input and Output of Image Data* on page 189 of the *PV-WAVE Programmer's Guide.*

Displaying Images on Monochrome Devices

Images are automatically dithered when sent to some monochrome devices. Dithering is a technique which increases the number of apparent brightness levels at the expense of spatial resolution. Images with 256 gray levels are displayed on a display with only two brightnesses, black and white, using halftoning techniques.

PV-WAVE supports dithering for output devices if their DEVICE procedures accept the keywords described below:

Floyd — If present and nonzero, selects the Floyd-Steinberg method of dithering. This algorithm distributes the error, caused by displaying intermediate shades in either black or white, to surrounding pixels. This method generally gives the most pleasing results but requires the most computation time.

Ordered — If present and nonzero, selects the Ordered Dither method of dithering. This introduces a pseudo-random error into the display by using a 4-by-4 "dither" matrix, yielding 16 apparent intensities. The Ordered Dither method is enabled by default.

Threshold — If present and nonzero, specifies use of the threshold algorithm — the simplest dithering method. The value of this keyword is the threshold to be used. This algorithm simply compares each pixel against the given threshold, usually 128. If the pixel equals or exceeds the threshold, the display pixel is set to white; otherwise, it is black.

Note PostScript handles dithering directly, and does not recognize the keywords listed above.

Displaying Images on 24-bit Devices

You can use PV-WAVE to display images in 24-bit color. Naturally, your workstation must support 24-bit color mode if you intend to view 24-bit images with PV-WAVE. Similarly, hardcopy devices must support 24-bit color mode if you intend to send 24-bit color output to them. To find out if your device has this capability, refer to Appendix A, *Output Devices and Window Systems*, in the *PV-WAVE User's Guide*.

Note 24-bit images may be either square or rectangular; they can be either pixel, row, or image interleaved. There is no restriction placed on the size of images by PV-WAVE; the limiting factors are the maximum amount of virtual memory available to you by the operating system and the processing time required.

Refer to the examples later in this section for more detailed information about how to read and display 24-bit images with PV-WAVE. For a comparison of true-color and pseudo-color images, refer to *Not all Color Images are True-color Images* on page 146.

Example: Read and Display a 24-bit Image-interleaved Image

This example reads 24-bit image data from a file, and then displays the image in a PV-WAVE window using 24-bit color. The 24-bit image is stored in a file as a set of stacked images, 512-by-512-by-3 deep (first the 512-by-512 red plane, then the 512-by-512 green plane, and then the 512-by-512 blue plane). The display device is an X-compatible device, and is capable of displaying 24-bit color:

```
DEVICE, Direct_Color=24
```

Define a PV-WAVE DirectColor graphics window.

```
status = DC_READ_24_BIT('jl.img', img, Org=1)
```

Read the 24-bit image from a file; DC_READ_24_BIT handles the opening and closing of the file. The variable 'img' now contains a 512-by-512-by-3 image array. Org=1 tells DC_READ_24_BIT that the file is image interleaved (as opposed to pixel interleaved).

```
TV, img, True=3
```

-or-

```
TVSCL, img, True=3
```

Display the 24-bit image using either the TV or the TVSCL procedures. The True keyword specifies the dimension over which the color is interleaved.

Example: Read and Display a 24-bit Image Stored in Three Different Files

This example reads 24-bit image data that has been stored in three separate image files — one red, one green, and one blue. Each file is read separately and then combined in one 3D array prior to displaying the 24-bit image. The data used in the example comes from the red, green, and blue images of Boulder in the `$WAVE_DIR/data` area. The use of environment variables in this example makes it a UNIX-specific example, although it can easily be adapted for use in a VMS environment, as well:

```
DEVICE, Direct_Color=24
```

Define a PV-WAVE DirectColor graphics window.

```
red = MAKE_ARRAY(477, 512, /Byte)
green = red
blue = red
```

Define three 477-by-512 variables to hold the image data. Each variable holds one "plane" of the data.

```
OPENR, 1, GETENV('WAVE_DATA') + $
   '/boulder_red.img'
READU, 1, red
CLOSE, 1
OPENR, 1, GETENV('WAVE_DATA') + $
   '/boulder_grn.img'
READU, 1, green
CLOSE, 1
OPENR, 1, GETENV('WAVE_DATA') + $
   '/boulder_blu.img'
```

```
READU, 1, blue
CLOSE, 1
```

Read each plane (red, green, and blue) of the image, placing the data in three different variables.

```
img = MAKE_ARRAY(477, 512, 3, /Byte)
img(*, *, 0) = red
img(*, *, 1) = green
img(*, *, 2) = blue
```

Create a 3D 24-bit image array and transfer each plane of the image into it.

```
TV, img, True=3
```

```
-or-
```

```
TVSCL, img, True=3
```

Display the 24-bit image using either the TV or the TVSCL procedures. The True keyword specifies the dimension over which the color is interleaved.

Note This example could have also used DC_READ_8_BIT to read the image data, and then the data files would not have had to be explicitly opened and closed. For more information about this function, see the DC_READ_8_BIT description in the *PV-WAVE Reference.*

Gray Level Transformations

Each pixel, or cell, in an image exhibits an intensity. By modifying the distribution of intensities it is possible to produce an image more suitable for a given application than the original. Of course, a suitable image for one application is not necessarily the best image for another application. The viewer is the ultimate judge of how well a particular method works. Evaluating image quality is a highly subjective process.

There are two ways to modify image intensities:

- modify the pixels and re-write the image on the display, or
- modify the color translation tables without changing the pixels.

The second method is faster because the color translation tables contain less information than the pixel memory, but it is not always practical because the original image may contain more discrete values than are representable in the display memory.

Thresholding, the Simplest Gray-level Transformation

The simplest example of a gray-level transformation is to produce a two-level mapping from all the possible intensities into black and white. If an image stored in a variable named A contains an object in which each pixel has an intensity value greater than S, a scalar, and pixels that are not part of the object have a value less than S, then the statement:

```
TVSCL, A GT S
```

will display all pixels in the object as full white and all background pixels as black.

The relational operators, EQ, NE, GE, GT, LE and LT, produce a value of 1 if the relation is true and 0 if the relation is false. When applied to images, the relation is applied to each pixel and an image of 1's and 0's results.

For example, the expression A GT S is an image with a value of 1 in each element where the corresponding element of A is greater than S; otherwise the element is set to 0. The TVSCL procedure then scales the image of 1's and 0's into 255's and 0's.

Of course, the opposite effect is obtained by the statement:

```
TVSCL, A LE S
```

All pixels whose value is greater than S but less than T are displayed as white with the following statement:

```
TVSCL, (A GT S) AND (A LT T)
```

Thresholding using Color Table Modification

If the original image scales into the range of integers representable in the display memory, the thresholding operators in the previous section may be implemented more efficiently by changing the color translation tables. For example, if a 256-element gray scale color table is appropriate, elements less than S become white with the following statements:

```
T = 255 * (INDGEN(256) LT S)
```

Elements less than S are 255, others are 0.

```
TVLCT, T, T, T
```

Load the color table from T.

Contrast Enhancement

An image may be contrast-enhanced so that any subrange of pixel values are scaled to fill the entire range of displayed brightnesses. For instance, if the image in variable A contains an object superimposed on a varying background, and the pixel values in the object range from a value of S to the brightest value in the entire image, the statement:

```
TVSCL, A > S
```

will use the entire range of display brightnesses to display the object.

The > operator, called the maximum operator, yields a result equal to the larger of its two operands. The expression `A > S` is an image in which each pixel in `A` less than `S` is set to `S`. `S` becomes the new minimum intensity. The TVSCL procedure then scales the new image from 0 to 255 before loading it into the display. Again, the image `A` is not changed.

If, for example, the object in `A` has values from 2.6 to 9.4, the statement:

```
TVSCL, A > 2.6 < 9.4
```

truncates the image so that 2.6 is the new minimum and 9.4 is the new maximum before scaling and display. Pixels with intensities of 9.4 or larger will be displayed at full brightness, while those with intensities of 2.6 or less are converted to minimum brightness.

Using BYTSCL to Enhance Contrast

The BYTSCL function can be used to enhance the contrast of images in a more efficient manner than the examples above. The result of this function is a byte image made by scaling the input image as follows:

$$R_{x,y} = T\,(I_{x,y} - Min)\;/\;(Max - Min)$$

where $I_{x,y}$ is the intensity value at image location (x, y). The value of T may be specified using the *Top* keyword parameter. Its default value is 255.

If BYTSCL is called with only one parameter, the maximum and minimum values are obtained by scanning the image parameter. You may directly specify the minimum and maximum values with keyword parameters. For example, the statement

```
TV, BYTSCL(A, MIN = 2.6, MAX = 9.4)
```

has exactly the same effect as the TVSCL statement in the previous section, stretching the contrast of pixels ranging from 2.6 to 9.4, but this statement is considerably quicker. Using BYTSCL is more efficient because the range truncation and scaling are performed in one pass, rather than in the four required by the TVSCL statement.

Modifying Color Tables to Enhance Contrast

If the image contains pixels in the range of 0 to 255, as in the case of an 8-bit display, or it can be transformed to 256 or fewer values, it is faster to modify the display color lookup tables rather than transforming the image in the computer and then loading the display. The STRETCH procedure allows any range of values between 0 and 255 to be linearly expanded to fill the display range.

For more information about stretching color tables, see *Stretching the Color Table* on page 319, or refer to the description for STRETCH in the *PV-WAVE Reference.*

Histogram Equalization

In many images, most pixels reside in a few small subranges of the possible values. By spreading the distribution so that each range of pixel values contains an approximately equal number of members, the information content of the display is maximized.

To equalize the histogram of display values, the count-intensity histogram of the image is required. This is a vector in which the *i*th element contains the number of pixels with an intensity equal to the minimum pixel value of the image plus *i*. The vector is of longword type and has one more element than the difference between the maximum and minimum values in the image. (This assumes a binsize of 1 and an image that is not of byte type.) The sum of all the elements in the vector is equal to the number of pixels in the image.

The HISTOGRAM function directly returns the count-intensity histogram. For example, to define a new variable H that contains the count-intensity histogram of the image A, type:

```
H = HISTOGRAM(A)
```

Optional keyword parameters may be included to specify the range and binsize, determine the minimum value of the image, etc.

From the count-intensity histogram, the cumulative distribution function is computed with the statements:

```
P = H
FOR i = 1, N_ELEMENTS(P)-1 DO P(i)= P(i) + $
   P(i-1)
```

P_i now contains the number of pixels in the original image with intensities less than or equal to *i*:

$$P_i = \sum_{j=0}^{i} Hj$$

P_i increases monotonically from the minimum value of the image to the number of pixels in the image.

By simply normalizing *P* so that its maximum element has a value of 255 and its minimum element has a value of 0, the gray level transformation necessary to display the image with histogram equalization is obtained:

```
P = BYTSCL(P)
```

The statement:

```
TVLCT, P, P, P
```

loads the three display color translation tables with the transformed function. The result is a black and white histogram-equalized display.

The HIST_EQUAL_CT procedure loads the color tables with a histogram equalized distribution, as described above. If called with no parameters, this procedure allows the user to mark a rectangular region of the display with the mouse, which is then used to form the distribution. It can also be called with an image as its parameter, in which case it uses the pixel distribution of the entire image.

Example of Histogram Equalization

The top plot of Figure 5-1 shows the count-intensity histogram of an original aerial image of the New York city area. The dashed line is the cumulative integral of this function, showing the number of pixels in the image with values less than or equal to each pixel value.

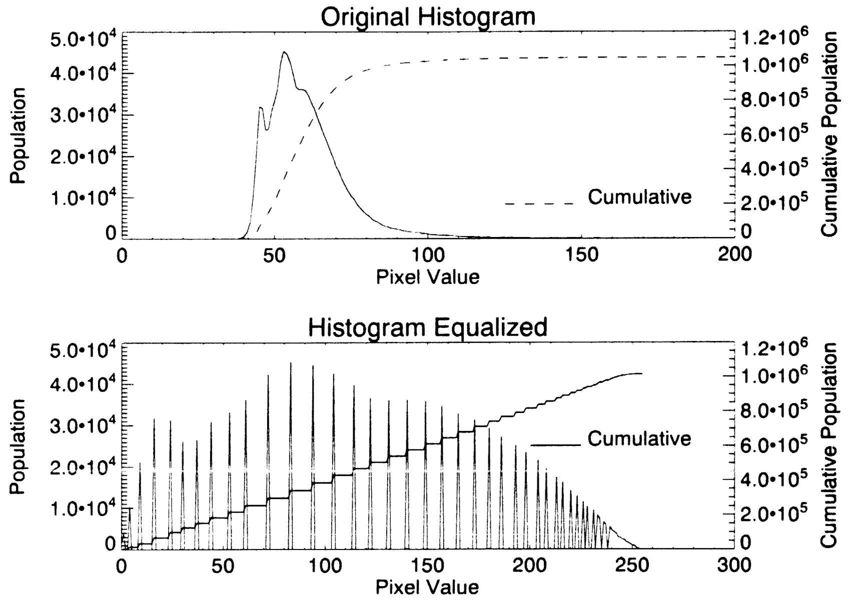


Figure 5-1 Histograms of original and histogram equalized image.

It is apparent that the pixel intensities range from approximately 40 to 140, implying that only about 40% of the usable brightness range of the display is used.

The bottom plot shows the pixel distribution histogram of the histogram-equalized image. Note that the histogram is spread over a much larger range and that the shape is somewhat flattened. Not all the values in this histogram are equal. This is due to the discrete bin size of the histogram and because there are unpopulated pixel ranges.

The cumulative integral of the histogram is nearly a straight line, from the origin to an X value of the maximum pixel value and a Y value equal to the number of pixels, as it should be.

The HIST_EQUAL function performs histogram equalization using this method. The following statement uses the function HIST_EQUAL to transform the image A and then display the result:

```
TV, HIST_EQUAL(A)
```

As described above, the HIST_EQUAL_CT procedure is more efficient because it modifies the color tables, rather than the image. The following two statements display the image, and then load the modified color tables:

```
TV, A
HIST_EQUAL_CT, A
```

Note that this method will only work if the original image contains integers in the range of 0 to 255.

Image Smoothing

The SMOOTH and MEDIAN functions are used to smooth images.

The SMOOTH Function

Images may be rapidly smoothed with the SMOOTH function. SMOOTH performs equally weighted smoothing using a square neighborhood of a given odd width. If A is an image of any type or size, the statement:

```
TVSCL, SMOOTH(A, 3)
```

displays the result of smoothing the image A with a 3-by-3 boxcar average. Smoothing with the triangular kernel:

$$\begin{vmatrix} 1 & 2 & 3 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix}$$

which approximates a two-dimensional triangle is simply implemented using the CONVOL function by the statement:

```
TVSCL, CONVOL(A,[[1,2,1],[2,4,2],[1,2,1]])
```

In the above function call the first parameter, usually an image, is convolved with the second parameter, usually a much smaller kernel array of weights. The second parameter:

```
[[1,2,1],[2,4,2],[1,2,1]]
```

is the PV-WAVE notation for a 3-by-3 array containing the above kernel. The bracket `[ ]` symbols are the array concatenation operators. Elements between the brackets, which may be scalars, vectors or arrays, are concatenated.

The same technique may be used for other types of smoothing, interpolation, or differentiation by merely changing the size and weights of the kernel parameter.

Median Smoothing with the MEDIAN Function

Median smoothing is a useful technique that is similar to mean smoothing as it is implemented by the SMOOTH function, except that the value of each pixel is replaced by the median of the N-by-N neighborhood rather than by the average.

Median smoothing, unlike mean smoothing, does not blur edges or features whose size is larger than the neighborhood. Also, median smoothing eliminates, without spreading, "salt and pepper" noise (isolated pixels containing extreme values). Median smoothing is implemented in PV-WAVE with the MEDIAN function:

```
TVSCL, MEDIAN(A, 3)
```

Figure 5-2 shows the effect of median and mean filters on a one-dimensional vector containing an impulse step function. Notice how the impulse is eliminated by the median filter rather than spread over the neighborhood of the filter as it is in the mean filter.

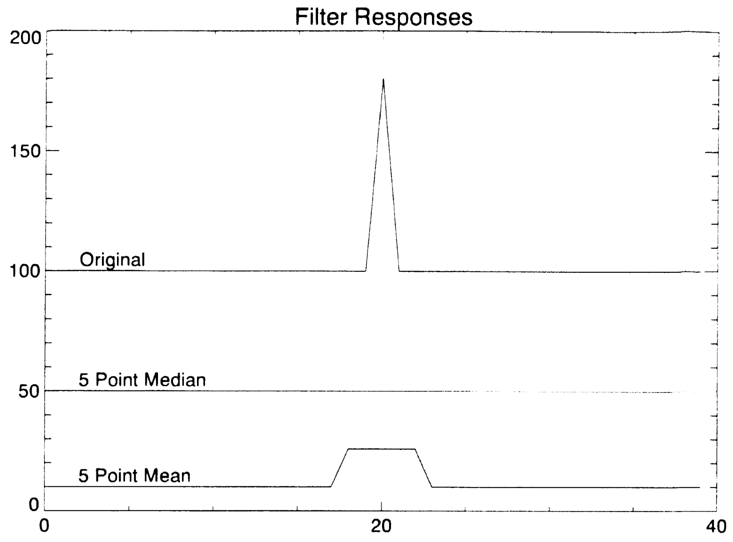


Figure 5-2 Responses of median and mean filters.

Image Sharpening

This section discusses some of the image sharpening methods available in PV-WAVE. For more details on the functions described here, see the *PV-WAVE Reference*.

The ROBERTS Function

An image may be sharpened (its edges or high spatial frequency components enhanced) by differentiation. One approximation to the derivative or gradient of the image is the Roberts Gradient, a form of cross difference, which is computed using the formula:

$$G(Fx,y) = |F_{x,y} - F_{x+1,\, y+1}| + |F_{x+1,y} - F_{x,y+1}|$$

The ROBERTS function returns this result.

The SOBEL Function

Another commonly used gradient operator is the Sobel operator. It operates over a 3-by-3 region, making it less sensitive to noise than an operator with a smaller neighborhood. The SOBEL function returns an approximation to the Sobel operator:

$$S_{x,y} = \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix} + \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}$$

where the notation,

$$\begin{vmatrix} w_0 & w_1 & w_2 \\ w_3 & w_4 & w_5 \\ w_6 & w_7 & w_8 \end{vmatrix}$$

indicates the absolute value of the sum of the pixels in the 3-by-3 neighborhood surrounding the pixel at X, Y, multiplied by the respective weights. The first term approximates the gradient in the Y direction and the second term approximates the gradient in the X direction.

Unsharp Masking Method

Another method of sharpening images is unsharp masking. This method subtracts a smoothed image (which contains only low frequency components) from the original image, leaving an image containing only high frequency components. This process emphasizes the edges and small, sharp features. To unsharp mask and display an image using a 3-by-3 neighborhood, use the command:

```
TVSCL, A - SMOOTH(A,3)
```

The CONVOL Function

The same result can be obtained by convolving the image with the kernel:

$$\begin{vmatrix} -1/9 & -1/9 & -1/9 \\ -1/9 & 8/9 & -1/9 \\ -1/9 & -1/9 & -1/9 \end{vmatrix}$$

or:

```
TVSCL, CONVOL(A, [[-1,-1,-1],[-1,8,-1], $
   [-1,-1,-1]],9)
```

The time required by the CONVOL function can become excessively long when the kernel or image is large. The time required is proportional to n^2m^2, where n is the size of the kernel and m is the size of the square image. Doubling the size of the kernel increases the time by a factor of four. The algorithm used in the SMOOTH function requires time in proportion to $2nm^2$, implying that it is almost always more efficient to use SMOOTH rather than CONVOL where possible.

Frequency Domain Techniques

Filtering in the frequency domain is a flexible technique that is used for smoothing, sharpening, deblurring, and image restoration. The three basic steps in image filtering are:

- ❑ Transforming the image into the frequency domain.
- ❑ Multiplying the resulting complex image by a filter that usually has only real values.
- ❑ Re-transforming this product back into the spatial domain, yielding the filtered image.

Assuming that `A` is the image to be filtered and `filter` is the variable containing the filter, this process is expressed in PV-WAVE by:

```
result = FFT(FFT(A, -1) * filter, 1)
```

The variable `A` may be of any data type except string; `filter` is a floating type filter and has the same dimensions as `A`; and `result` is the resulting image which is of complex type and has the same size as `A`. The second parameter of the FFT function specifies the direction of the transform: –1 for space to frequency; and +1 for frequency to space.

This process is equivalent to convolving the image with the spatial equivalent of the filter in the spatial domain, but is much quicker than simple convolution for kernels larger than approximately 9-by-9.

Caution Try to avoid wrap-around artifacts when filtering and convolving in the frequency domain. In particular, images must be properly windowed and sampled before applying the Fourier Transform or false and misleading values will result. For one example of windowing, see the source code for the HANNING procedure in the Standard Library.

Filtering Images

Many types of images can be improved by filtering. PV-WAVE's array-oriented operators and functions make it particularly easy to design and use filters. Many commonly used filters take advantage of what is called the frequency image. The frequency image, *D*, of an *n*-by-*n* array in which each pixel element contains the spatial frequency of the pixel in units of cycles per pixel is given by:

$$D_{x,y} = \sqrt{\left(\frac{x'}{n}\right)^2 + \left(\frac{y'}{n}\right)^2}$$

where:

$x' = x$	if $x \le n/2$, otherwise $n - x$
$y' = x$	if $x \le n/2$, otherwise $n - y$

The Standard Library function DIST evaluates the function above and returns a frequency image. For example, to obtain a frequency image to use with a filter for a 256-by-256 image, use the command:

```
D = DIST(256)
```

Some of the many filters which can be computed from the frequency image in one step are given below. The mathematical description of the filter is given first, followed by the PV-WAVE code to implement it.

- Ideal low pass filter, absolute cutoff at frequency D_0:

$$filter_{u,v} = \begin{cases} 1 \text{ if } D_{u,v} < D_0 \\ 0 \text{ otherwise} \end{cases}$$

```
filter = D LT D0
```

- Ideal high pass filter, absolute cutoff at D_0:

$$filter_{u,v} = \begin{cases} 1 \text{ if } D_{u,v} > D_0 \\ 0 \text{ otherwise} \end{cases}$$

```
filter = D GT D0
```

- Ideal bandpass filter, absolute cutoff at D_L and D_H:

$$filter_{u,v} = \begin{cases} 1 \text{ if } D_L < D_{u,v} < D_H \\ 0 \text{ otherwise} \end{cases}$$

```
filter = (D GT DL) AND (D LT DH)
```

- Butterworth low pass filter of order n, cutoff at D_0: (The frequency response at the cutoff frequency is equal to 50% of the maximum.)

$$filter_{u,v} = 1/(1 + [D_{u,v}/D_0]^{2n}$$

```
filter = 1/(1+(D/D0) ^(2*N))
```

- Butterworth high pass filter of order n, cutoff at $D - 0$:

$$filter_{u,v} = 1/(1 + [D_0/D_{u,v}]^{2n}$$

```
filter = 1/ (1 + (D0/D) ^ (2*N))
```

- Butterworth bandpass filter, order n, center frequency is C, width of D_0:

$$filter_{u,v} = 1/(1 + [(D_{u,v} - C)/D_0]^{2n}$$

```
filter = 1/ (1+((D-C)/D0) ^ (2*N))
```

- Exponential low pass filter of order n:

$$filter_{u,v} = e^{-(D_{x,y}/D_0)^n}$$

```
filter = EXP(-(D/D0) ^ N)
```

- Exponential high pass filter of order *n*:

$$filter_{u,v} = e^{-(D_0/D_{x,y})^n}$$

```
filter = EXP(- (D0/D) ^ N)
```

The filters described here must be applied in the frequency domain. To use these filters the image must be transformed to the frequency domain with the Fast Fourier Transform, multiplied by the filter, and then transformed back to the spatial domain.

The following command is used to apply a filter to the variable image:

```
filtered_image = FFT(FFT(image, -1) * $
   filter, 1)
```

Displaying the Fourier Spectrum

PV-WAVE makes it easy to calculate the Fourier spectrum of an image. Figure 5-3 shows an aerial photograph on the left and its logarithmically scaled Fourier spectrum on the right. Note that the diagonal, vertical, and horizontal lines in the Fourier spectrum correspond to the roads in the original image and are perpendicular to them.

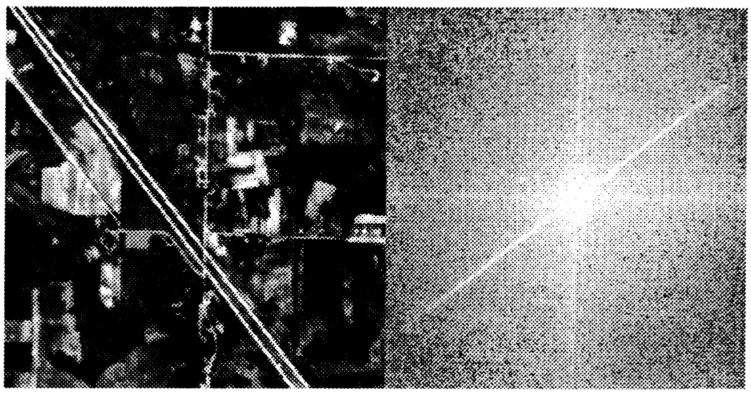


Figure 5-3 An aerial photograph and its Fourier spectrum.

The Fourier spectrum is also displayed in Figure 5-4 as a surface plot.

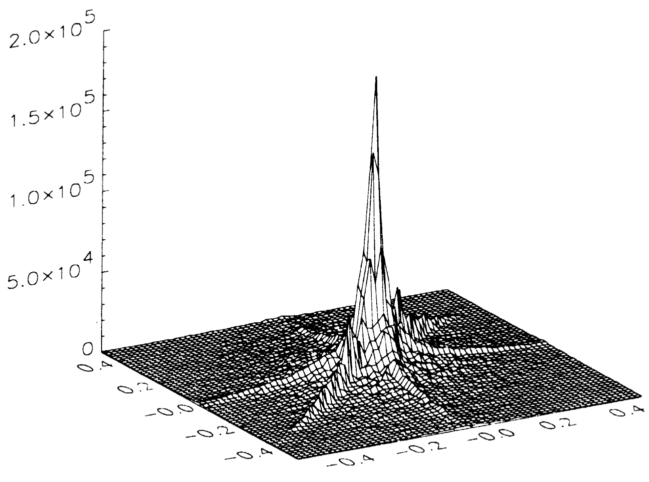


Figure 5-4 Surface plot of the aerial photograph's Fourier spectrum.

It is customary to display the Fourier or power spectrum of images with the DC frequency component in the center of the image, as is done here. This is easily accomplished in PV-WAVE by using the SHIFT function to shift the origin of the 256-by-256 image to the center. The Fourier spectrum in the right side of Figure 5-3 is produced with the statement:

```
TVSCL, SHIFT(ALOG( ABS( FFT(A, -1))),256, 256)
```

This statement performs the following operations:

- The FFT function transforms the image into the frequency domain.
- The ABS function calculates the magnitude of each complex-valued pixel.
- The ALOG function returns the natural logarithm of each pixel.
- The SHIFT function shifts the image so the point with a subscript of (0, 0) is in the center.
- The TVSCL procedure scales and displays the result.

Geometric Transformations

Geometric transformations rearrange the elements of an image. Some examples of commonly used geometric transformations are: magnification, rotation, projection to different coordinate systems, and correction of distortions.

The definition of a geometric transformation may be written:

$$g(x, y) = f(u, v) = f[a(x, y), b(x, y)]$$

where $f(u, v)$ is the input image, $g(x, y)$ is the output image, and the functions a and b specify the spatial transformation that relate the (x, y) coordinate system of the output image to the (u, v) coordinates of the input image.

Rotating and Transposing with the ROTATE Function

The simple and common operations of rotation by multiples of 90 degrees and/or transposition are performed most efficiently by the ROTATE function.

Usage

ROTATE*(image, rotation)*

Parameters

image — The input image.

rotation — An integer specifying one of the eight possible combinations of axis interchange and reversal.

Example

For example, a 90 degree counterclockwise rotation of an *m*-by-*n* image is expressed in the above notation by:

```
rotated_image = ROTATE(image, 1)
```

You can also use the ROT function to rotate an image. It uses POLY_2D (described in the next section) to rotate an image about a specified point with optional magnification or reduction. The rotation angle is not restricted to multiples of 90 degrees as in ROTATE, but ROT is slower. For more information on these functions, see the *PV-WAVE Reference.*

Geometric Transformations with the POLY_2D Function

The POLY_2D function provides an efficient method of performing geometric transformations, assuming the functions *a* and *b* can be expressed as *N*-degree polynomials of *x* and *y*:

$$u = a(x,y) = \sum_{i=0}^{N}\sum_{j=0}^{N} C_{i,j_{ij}} x^j y^i$$

$$v = b(x, y) = \sum_{i=0}^{N} \sum_{j=0}^{N} D_{i,j} x^j y^i$$

Either the nearest neighbor or bilinear interpolation methods may be selected.

Usage

output_image = POLY_2D(*image, c, d* [, *interp* [, *dim$_x$*, *dim$_y$*]])

Parameters

image — The input image.

c and *d* — The arrays containing the polynomial coefficients. Each array must contain $(N + 1)^2$ elements. For example, for a linear transformation C and D contain four elements, and may be a two-by-two array or a four-element vector. $C_{i,j}$ contains the coefficient used to determine u, and is the weight of the term $x^j y^i$. The POLY-WARP procedure may be used to fit (u, v) as a function of (x, y). It returns the coefficient arrays C and D.

Interp — If present and non-zero, selects bilinear interpolation, otherwise the nearest neighbor method is used. For the linear case (i.e. $N = 1$) bilinear interpolation requires approximately twice as much time as does the nearest neighbor method.

Dim$_x$ and *Dim$_y$* — Specify the dimensions of the result. If omitted, the result will have the same dimensions as the original image.

In addition, the *Missing* keyword parameter may be included to specify the output value of pixels whose *u, v* coordinates are outside the input image. If this keyword parameter is not present, missing values are extrapolated from the edges of the input image.

Efficiency and Accuracy of Interpolation

POLY_2D is relatively efficient; however, some of this efficiency is gained at the expense of accuracy. Each output pixel is mapped to the input image and the nearest pixel is used for the result. This method is called the nearest neighbor method. With high magnifications of regular structure, objectionable sawtooth edges result. Bilinear interpolation avoids this effect by determining the value of each output pixel by interpolating from the four neighbors of actual location in the input image at the expense of additional computations.

Correcting Linear Distortion with Control Points

The following example uses POLY_2D to correct a linear distortion using control points. A calibration image containing *n* known points is acquired by a system with linear distortion. Given the original position of each point in the calibration image, *(x, y)* and its measured coordinates in the acquired image, *(u, v)*, it is possible to obtain the polynomial coefficients required to transform the acquired image back to the original.

The value of *n* must be at least four to determine the coefficients of a first degree transformation, as there are $(n + 1)^2$ coefficients in each array, each of which is an unknown to be solved. In this example, four points are measured which describe the pixel coordinates of the corners of a box in the undistorted calibration image: (20, 20), (40, 20), (40, 40), (20, 40). The measured coordinates of the corners of the box, which is distorted into the shape of a trapezoid in the acquired image, are assumed to be: (25, 25), (55, 25), (60, 50), (25, 50). See Figure 5-5.

The equations relating the *(u, v)* coordinates to *(x, y)* are:

$$u_i = c_0 + c_1 y_i + c_2 x_i + c_3 y_i x_i ,\ i = 0, 1, 2, 3$$

$$v_i = d_0 + d_1 y_i + d_2 x_i + d_3 y_i x_i ,\ i = 0, 1, 2, 3$$

We can write the four equations for u_i as:

$$U = ZC$$

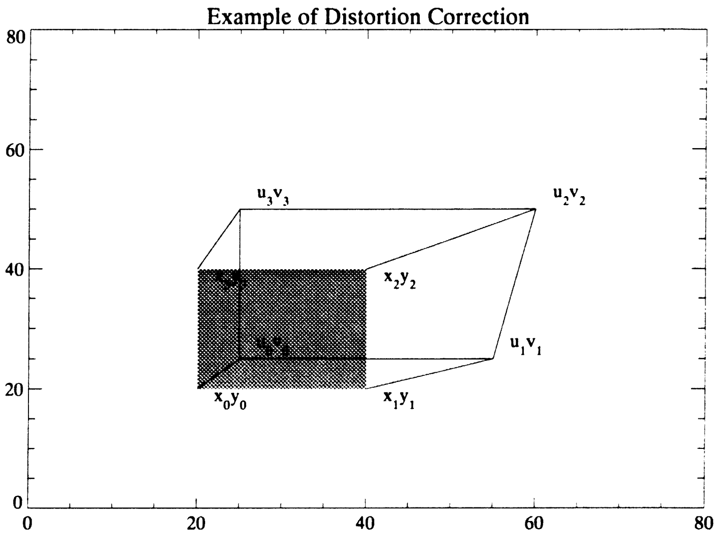


Figure 5-5 Example of geometric distortion.

where $C = [c_0, c_1, c_2, c_3]$, and

$$Z = \begin{vmatrix} 1\ y_0 & x_0 & x_0y_0 \\ 1\ y_1 & x_1 & x_1y_1 \\ 1\ y_2 & x_2 & x_2y_2 \\ 1\ y_3 & x_3 & x_3y_3 \end{vmatrix}$$

Solving for C and D:

$$C = Z^{-1}U \qquad D = Z^{-1}V$$

The PV-WAVE statements implementing this algorithm are:

```
x = [20, 40, 40, 20]
```
Define undistorted x coordinates of box.

```
y = [20, 20, 40, 40]
```
... and y.

```
u = [25, 55, 60, 25]
```
Measured coordinates...

```
v = [25, 25, 50, 50]
```

```
z = FLTARR(4,4)
```

Define the Z matrix.

```
for j=0,1 do for k=0,1 do z(0,j+2*k) = x^k * $
   y^j
```

Fill it, a row at a time.

```
image = BYTARR(100, 100)
```

Create a 100-by-100 image.

```
image(POLYFILLV(u, v, 100, 100)) = 128
```

Simulate the acquired image by filling the pixels inside the (u, v) box with the value 128.

```
q = POLY_2D(image, (c = INVERT(z) # u),  $
   (d = INVERT(z) # v), 1)
```

Solve the equations, using the INVERT function, and apply the geometric transformation, yielding image q, saving coefficients in c and d.

The computed values of c and d are [0.0, –0.25, 1.25, 0.0125], and [0.0, 1.25, 0.0, 0.0].

Figure 5-6 illustrates the application of this geometric transformation to an image. The left side of this figure contains the trapezoid defined by the distorted coordinates of the "acquired" image. The right side is the result of the transformation, as the trapezoid is warped back to the original rectangular shape.

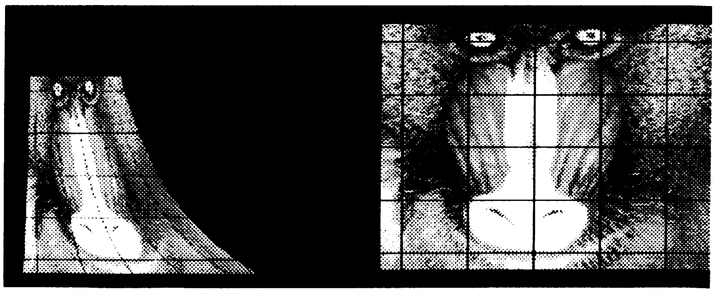


Figure 5-6 Correcting a geometric transformation.

The POLYWARP procedure may be used to obtain the polynomial coefficients in a more general manner. It is not restricted to first-order polynomials, and it computes a least squares fit if there are more than $(n + 1)^2$ control points. For more information on the POLYWARP procedure, see its description in the *PV-WAVE Reference*.

Mathematical Morphology

Mathematical morphology is an approach to image processing that is based on shape. If mathematical morphology is used appropriately, image data can be simplified without losing essential shape characteristics. It plays a particularly important role in those image processing applications that depend on object or feature recognition. For example, some manufacturing defects correlate directly with shape and can be discovered with this approach to image processing.

Mathematical morphology is based on set theory; sets represent the various shapes that are manifested on binary or gray scale images. Dilation is the morphological transformation that combines two sets using vector addition of set elements. It is implemented in PV-WAVE with the DILATE function. The dilation operator is commonly known as the "fill," "expand," or "grow" operator. It is used to fill "holes" in the image that are equal to or smaller in size than a particular structuring element.

Erosion is the morphological opposite of dilation. It is the morphological transformation that combines two sets using the vector subtraction of set elements. Erosion is implemented in PV-WAVE with the ERODE function. The erosion operator is commonly known as the "shrink" or "reduce" operator. It is used to reduce islands smaller than a particular structuring element.

Complete descriptions of the DILATE and ERODE functions are given in the *PV-WAVE Reference*. Additional information on mathematical morphology in general can be found in the article "Image Analysis Using Mathematical Morphology" by Haralick, Sternberg, and Zhuang, found in the *IEEE Transaction on Pattern Analysis and Machine Intelligence*, Vol. PAMI-9, No.4, July, 1987.

6

CHAPTER

Advanced Rendering Techniques

You can render 3D geometric and volumetric data using the advanced rendering capabilities of PV-WAVE. Most of these functions are part of the standard library. The RENDER function is a system routine that performs rendering using the ray tracing technique.

In addition, the standard library contains several utility functions for gridding (2D, 3D, 4D, and spherical) and for conversion of rectangular, polar, cylindrical, and spherical coordinates.

For additional information on the rendering, gridding, and coordinate conversion functions discussed in this chapter, see the *PV-WAVE Reference.*

Advanced rendering in PV-WAVE is performed using a technique called ray tracing. Ray tracing is the process of following the path of light rays from a light source into a scene. It is one of the most powerful techniques in the image synthesis gallery. The PV-WAVE ray-tracing Renderer handles translucency and opacity, and provides the ability to display both geometric (polygonal) and volume data within one image.

For example, it allows you to display the fluid air flow over, around, or through an object, such as an airplane wing together with a general description of the wing.

Using the PV-WAVE's renderer, you can also generate pictures of voxel (volume) data directly, without having to convert to a polygonal iso-surface representation first.

Using the other routines in the Advanced Rendering Library, you can now easily mix the methods you employ for visualizing your data – for example, geometric data with volumetric data. (Volumetric data are 3D entities that have information inside them, instead of using polygons and lines to merely represent geometric surfaces and edges.)

These routines also provide:

- ✓ 3D vector field plots
- ✓ iso-surfaces for polygonal representation. (An iso-surface is a pseudo-surface of constant density within a volumetric data set.)
- ✓ a volume slicer for interactive subsetting and display of volumetric data
- ✓ a view tool to graphically set the X, Y, Z position
- ✓ "rubber sheet" mapping of an image onto a sphere

Demonstration Programs

There are three sources for advanced rendering demonstration programs:

- the directory, `$WAVE_DIR/demo/arl`. (You can run the demonstration programs and look at the code.)

  On VMS, these procedures are located in the directory: `WAVE_DIR:[demo.arl]`.
- the directory, `$WAVE_DIR/demo/render`

  On VMS, these procedures are located in the directory: `WAVE_DIR:[demo.render]`.
- the PV-WAVE Demonstration Gallery.

Demonstration Programs in the Examples Directory

You can find most of the Advanced Rendering Library demonstration programs in the `$WAVE_DIR/demo/arl` directory.

Note Before running these programs on a UNIX system, be sure to read the ARL section of the *PV-WAVE Tips and Technical Notes* for important information.

To run these programs:

- ❑ Change to the `examples` directory and start PV-WAVE.
- ❑ At the `WAVE>` prompt, enter the name of the program you want to run. For example:

  ```
  WAVE> vol_demo1
  ```

- ❑ If you want to run another program, it is recommended that you start a new session (exit and re-enter PV-WAVE) before typing in another program name. Exiting insures that all variables are cleared, and that none of the data or displays from the previous programs interfere with the new demonstration program.

Ray Tracing Demonstration (Render Directory)

Ray tracing demonstrations are located in the `$WAVE_DIR/demo/render` directory. On VMS, these procedures are located in the directory: `WAVE_DIR:[demo.render]`.

These example use the RENDER function to generate images. For a quick preview of these images, try the following.

```
% cd $WAVE_DIR/demo/render
```

The UNIX location of the ray tracing render examples.

```
% wave
WAVE> p = show_anim()
```

Quadric Animation. Shows earth revolving for several revolutions.

```
WAVE> MOVIE,P,ORDER=0
```

Quadric animation with earth continuously revolving. This demo takes approximately five minutes to complete.

```
WAVE> show_iso_head
```
Polygonal Mesh with Many Polygons (Iso-surfaces)

```
WAVE> show_slic_head
```
Slicing a volume.

```
WAVE> show_flat_head
```
Rendering an Iso-surface with Voxel Values.

```
WAVE> show_tran_head
```
Diffuse and Partially Transparent Iso-surfaces

```
WAVE> show_core_head
```
Rendering Iso-surfaces with Transformation Matrices. Renders two volumes in the same scene.

The above routines display the resulting rendered images that have already been created with gen_ routines and then stored in files. To get a feel for how long it actually takes to create these rendered images on your workstation, enter the following commands. (On typical workstations, they each take from five to ten minutes to generate, where the time taken is a function of the speed at which your workstation does floating-point arithmetic.)

```
% cd $WAVE_DIR/demo/render
% wave
WAVE> gen_amin
```
Quadric Animation

```
WAVE> gen_iso_head
```
Polygonal Mesh with many Polygons

```
WAVE> gen_slic_head
```
Slicing a Volume

```
WAVE> gen_flat_head
```
Rendering an Iso-surface with Voxel Values

```
WAVE> gen_tran_head
```
Diffuse and Partially Transparent Iso-surfaces

```
WAVE> gen_core_head
```
Rendering Iso-surfaces with Transformation Matrices. Renders two volumes in the same scene.

SLICE_VOL Function and VIEWER Procedure Demonstrations

These two PV-WAVE routines can be used to manipulate and view portions of volumes.

The SLICE_VOL function returns a two-dimensional array containing a slice from a 3D volumetric array. You can demonstrate the SLICE_VOL function using the Medical Imaging and CFD/Aerospace buttons of the PV-WAVE Demonstration Gallery.

The VIEWER procedure lets you interactively define a 3D view, a slicing plane, and multiple cut-away volumes. You can demonstrate the VIEWER procedure using the 4-D Data, Medical Imaging, Oil/Gas Exploration, and CFD/Aerospace buttons of the PV-WAVE Demonstration Gallery.

Tables of Demonstration Programs

The following tables summarizes the demonstration programs and lists the PV-WAVE rendering routines that are used in these programs:

- Table 6-1, Polygon Rendering
- Table 6-2, Volume Rendering
- Table 6-3, Polygon and Volume Rendering
- Table 6-4, Gridding

Table 6-1: Polygon Rendering

Demonstration Programs	Routines Used
poly_demo1 Displays a perspective view of a surface from a viewpoint within the data. This program is located in $WAVE_DIR/demo/arl.	SET_VIEW3D POLY_SURF POLY_NORM POLY_TRANS POLY_DEV POLY_C_CONV POLY_PLOT

Table 6-1: Polygon Rendering

Demonstration Programs	Routines Used
grid_demo4 Shows 4D gridding and a cut-away view of a block of volume data. This program is located in $WAVE_DIR/demo/arl.	GRID_4D VOL_PAD CENTER_VIEW SHADE_VOLUME POLYSHADE
f_gridemo4 Shows 4D gridding and a cut-away view of a block of volume data. This program is located in $WAVE_DIR/demo/applications/arl/examples.	FAST_GRID4 VOL_PAD CENTER_VIEW SHADE_VOLUME POLYSHADE
cube1 Constructs a polygonal mesh of diffusely shaded polygons. This program is not on the tape.	MESH, RENDER
cube2 Constructs a polygonal mesh of flat-shaded polygons.This program is not on the tape.	MESH, RENDER
gen_iso_head show_iso_head Creates a human head using a polygonal mesh with 52,500 polygons.These programs are located in $WAVE_DIR/demo/render.	SHADE_VOLUME MESH, RENDER
sphere_demo1 Displays an image warped onto a sphere. This program is located in $WAVE_DIR/demo/arl.	POLY_SPHERE CENTER_VIEW POLYSHADE

Table 6-1: Polygon Rendering

Demonstration Programs	Routines Used
sphere_demo2 Displays data warped onto an irregular sphere. This program is located in $WAVE_DIR/demo/arl.	POLY_SPHERE CENTER_VIEW POLYSHADE POLY_COUNT POLY_NORM POLY_TRANS POLY_DEV POLY_C_CONV POLY_PLOT
sphere_demo3 Displays multiple spheres merged together. This program is located in $WAVE_DIR/demo/arl.	GRID_SPHERE POLY_SPHERE POLY_COUNT POLY_TRANS POLY_MERGE CENTER_VIEW POLYSHADE POLY_NORM POLY_DEV POLY_PLOT
grid_demo5 Shows spherical gridding. This program is located in $WAVE_DIR/demo/arl.	GRID_SPHERE POLY_SPHERE CENTER_VIEW POLYSHADE
gen_anim show_anim Constructs a "movie" of an orbit around a sphere. This program takes several minutes to run. These programs are located in $WAVE_DIR/demo/render.	SPHERE RENDER

Table 6-2: Volume Rendering

Demonstration Programs	Routines Used
vec_demo1 Displays a 3D vector field from X-Y-Z data. This program is located in $WAVE_DIR/demo/arl.	VECTOR_FIELD3
vec_demo2 Displays a 3D vector field from the volumetric data with specified starting points for the vectors. This program is located in $WAVE_DIR/demo/arl.	CONV_TO_RECT VECTOR_FIELD3
vol_demo1 Displays a 3D fluid flow vector field with random starting points for the vectors. This program is located in $WAVE_DIR/demo/arl.	CONV_TO_RECT VECTOR_FIELD3
gen_slic_head show_slic_head Demonstrates the rendering of selected slices through some volume data. These programs are located in $WAVE_DIR/demo/render	VOLUME RENDER
gen_flat_head show_flat_head Renders a diffuse iso-surface with voxel values. These programs are located in $WAVE_DIR/demo/render.	VOLUME RENDER
gen_tran_head show_tran_head Renders both a diffuse iso-surface together with a partially transparent iso-surface. These programs are located in $WAVE_DIR/demo/render.	VOLUME RENDER

Table 6-2: Volume Rendering

Demonstration Programs	Routines Used
gen_core_head show_core_head Renders a diffuse iso-surface using actual voxel values Demonstrates the rendering of two volumes into a single image. These programs are located in $WAVE_DIR/demo/render.	VOLUME RENDER

Table 6-3: Polygon and Volume Rendering

Demonstration Programs	Routines Used
vol_demo2 Displays an MRI scan of a human head using three different display techniques. This demonstration takes a while to run. This program is located in $WAVE_DIR/demo/arl.	VOL_PAD CENTER_VIEW VOL_MARKER SHADE_VOLUME POLYSHADE VOL_TRANS VOL_REND
vol_demo3 Displays 3D fluid data using two display techniques. This program is located in $WAVE_DIR/demo/arl.	CENTER_VIEW SHADE_VOLUME POLYSHADE VOL_PAD VOL_TRANS VOL_REND
vol_demo4 Similar to grid_demo3, but also renders the data using POLY_PLOT and VOL_REND. This program is located in $WAVE_DIR/demo/arl.	GRID_4D VOL_PAD CENTER_VIEW SHADE_VOLUME POLYSHADE POLY_NORM POLY_TRANS POLY_DEV POLY_COUNT POLY_PLOT VOL_TRANS VOL_REND

Table 6-4: Gridding

Demonstration Program	Routines Used
f_gridemo2 Shows 2D gridding with dense data input. This program is located in $WAVE_DIR/demo/arl.	FAST_GRID2
f_gridemo3 Shows 3D gridding with dense data input. This program is located in $WAVE_DIR/demo/arl.	FAST_GRID3
f_gridemo4 Shows 4D gridding with dense data input. This program is located in $WAVE_DIR/demo/arl.	FAST_GRID4
grid_demo2 Shows 2D gridding with sparse data input. This program is located in $WAVE_DIR/demo/arl.	GRID_2D
grid_demo3 Shows 3D gridding with sparse data input. This program is located in $WAVE_DIR/demo/arl.	GRID_3D
grid_demo4 Shows 4D gridding with sparse data input. This program is located in $WAVE_DIR/demo/arl.	GRID_4D
grid_demo5 Shows spherical gridding. This program is located in $WAVE_DIR/demo/arl.	GRID_SPHERE

Note The Advanced Rendering Library also contains the demonstration program, `img_demo1`. This program displays a pseudo true-color Landsat image on an 8-bit color system. On some systems, it may be necessary to click in the Wave 0 window to see the proper colors.

The Basic Rendering Process

The five basic steps to rendering are:

- ❑ Import or generate data to be rendered. See the section *Importing and Generating Data for Rendering* on page 185 for details.
- ❑ Manipulate and convert data. This step is optional, depending on the type of data you are using. PV-WAVE provides several functions and procedures for transforming data to be rendered. See the section, *Manipulating and Converting Data* on page 191 for details.
- ❑ Set up your data for viewing. See the section *Setting Up Data for Viewing* on page 194 for details.
- ❑ Use one of the rendering routines to render the image. The PV-WAVE rendering routines are:

  POLY_PLOT

  POLYSHADE

  VECTOR_FIELD3

  VOL_MARKER

  VOL_REND

  RENDER

  See the sections *Rendering with Standard Techniques* on page 195 and *Ray-tracing Rendering* on page 196.
- ❑ Display data. See the section *Displaying Rendered Images* on page 219.

Importing and Generating Data for Rendering

Before you can render data, you must import and/or generate data. There are several ways to render imported or generated data. The demonstration programs illustrate five ways:

- Import the data, manipulate the data, set up the data for viewing, and then render the imported data. Demonstration programs that illustrate this method are:

  `vec_demo2`

  `vol_demo1`

- Import the data, generate polygons or volumes, manipulate the data, set up the data for viewing, and then render the data. Examples are:

  `poly_demo1`

  `vol_demo2`

  `vol_demo3`

- Import the data, generate polygons or volumes, set up the data for viewing, and then render the data. Examples are:

  `sphere_demo1`

  `gen_iso_head`

  `gen_amin`

  `gen_slic_head`

  `gen_flat_head`

  `gen_tran_head`

  `gen_core_head`

  The `gen_` routines import data, generate polygons or volumes, use the RENDER function to render images and then store the rendered images in a file.

- Generate polygons, manipulate the data, set up the data for viewing, and then render the data. Examples are:

  `sphere_demo2`

  `sphere_demo3`

  `f_gridemo4`

  `grid_demo4`

- Generate polygons, set up the data for viewing, and then render the data. Examples are:

  `grid_demo5`

  `vec_demo1`

  `cube1`

  `cube2`

Importing Data

You can render data that is imported from one or more files. Refer to Chapter 8, *Working with Data Files*, in the *PV‑WAVE Programmer's Guide* for details on importing data into PV‑WAVE. Example programs that import data from more than one file are `poly_demo1`, `vec_demo2`, and `vol_demo1`.

Generating Polygons and Volumes

PV‑WAVE provides routines for creating various types of polygons and volumes such as meshes, rectangular surfaces, spherical surfaces, cones, and cylinders.

Some of these routines are only used with the RENDER function (CONE, CYLINDER, MESH, SPHERE and VOLUME). For information on these RENDER-specific functions, see the section *Specifying RENDER Objects* on page 196, as well as the individual function descriptions in *PV‑WAVE Reference*.

Many of the render routines and their utilities require a *vertex_list* and a *polygon_list* as input parameters. Routines that generate a *vertex_list* and a *polygon_list* representation for polygons and volumes are described in this section. These routines include POLY_SPHERE, POLY_SURF, and SHADE_VOLUME.

Vertex Lists and Polygon Lists

PV‑WAVE uses a very simple format for polygonal representation. It consists of an array of vertices and a flat one-dimensional array of polygons, as described below.

- ***vertex_list*** – A (3, *n*) array containing the three-dimensional coordinates of each vertex.
- ***polygon_list*** – An array containing the number of sides for each polygon and the subscripts into the *vertex_list* array.

Here's an example of how to render two adjacent square polygons with a *vertex_list*:

X-axis	Y-axis	Z-axis
0.0	0.0	0.0
1.0	0.0	0.0
2.0	0.0	0.0
2.0	1.0	0.0
1.0	1.0	0.0
0.0	1.0	0.0

As shown in Figure 6-1, there are only six vertices in the resulting *vertex_list* because two vertices are shared by both polygons.

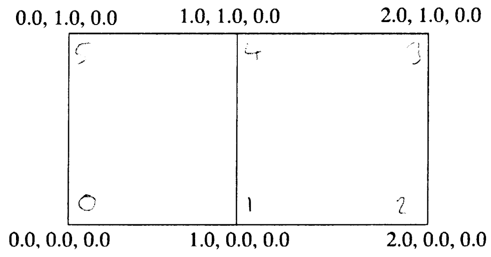


Figure 6-1 Vertices of two square polygons. Six vertices define both polygons.

The *polygon_list* then contains:

4	The first polygon has 4 sides.
0	The first vertex is *vertex_list(*, 0).*
1	The second vertex is *vertex_list(*, 1).*
4	The third vertex is *vertex_list(*, 4).*

5 The fourth vertex is *vertex_list(*, 5)*.

4 The second polygon has 4 sides.

1 The first vertex is *vertex_list(*, 1)*.

2 The second vertex is *vertex_list(*, 2)*.

3 The third vertex is *vertex_list(*, 3)*.

4 The fourth vertex is *vertex_list(*, 4)*.

The rendering procedures POLYSHADE and POLY_PLOT both use a *vertex_list* and *polygon_list* as input parameters. Other routines that use either a *vertex_list* or a *polygon_list* include:

- POLY_C_CONV
- POLY_COUNT
- POLY_DEV
- POLY_NORM
- POLY_MERGE
- POLY_TRANS

The RENDER function also requires a *vertex_list* and a *polygon_list* if it is used to render polygonal meshes with the MESH function. Polygonal meshes representing objects that have been derived outside of PV-WAVE can be imported, converted to the representation used by MESH, and then rendered with the RENDER function.

Examples of the RENDER function that use *vertex_list* and *polygon_list* to create polygonal meshes include *Example 1: Polygonal Mesh (Diffusely-shaded Polygons)* on page 204, *Example 2: Polygonal Mesh (Flat-shaded Polygons)* on page 204, and *Example 3: Polygonal Mesh (Many Polygons)* on page 205.

Rectangular Surfaces

You can generate a *vertex_list* and a *polygon_list* for rectangular surfaces with the POLY_SURF procedure. This procedure generates a three-dimensional *vertex_list* and a *polygon_list* from a two-dimensional array that contains Z values. The example program `poly_demo1` uses this procedure.

Spherical Surfaces

You can use the POLY_SPHERE procedure to generate a *vertex_list* and a *polygon_list* for a sphere. Demonstration programs that use this procedure are:

- `grid_demo5`
- `sphere_demo1`
- `sphere_demo2`
- `sphere_demo3`

Three-Dimensional Volumes

The SHADE_VOLUME procedure generates a *vertex_list* and *polygon_list* describing the contour iso-surface of a given three-dimensional volume. Example programs that use SHADE_VOLUME include:

- `f_gridemo4`
- `grid_demo4`
- `vol_demo2`
- `vol_demo3`
- `vol_demo4`
- `gen_iso_head`

For a complete description of the SHADE_VOLUME procedure and the other procedures mentioned in this section, see the *PV-WAVE Reference*.

Manipulating and Converting Data

PV-WAVE provides routines for manipulating and converting data, as summarized in this section. This step is optional depending on the type of data you are using. For details about each routine, see its description in *PV-WAVE Reference.*

2-, 3-, and 4-dimensional Gridding

Gridding is a method that generates a uniform grid from irregularly spaced data; the method interpolates or extrapolates new data from a given set of data, and then creates a uniform grid that maps this data. PV-WAVE supports 2D, 3D, and 4D gridding.

2D Gridding

The functions FAST_GRID2 and GRID_2D return a gridded one-dimensional array containing Y values for input data with X, Y coordinates. The FAST_GRID2 function works best with dense data points (more than a thousand points to be gridded). The GRID_2D function works best with sparse data points (less than a thousand points to be gridded).

3D Gridding

The functions FAST_GRID3 and GRID_3D return a gridded two-dimensional array containing Z values for input data with X, Y, and Z coordinates. The FAST_GRID3 function works best with dense data points. The GRID_3D function works best with sparse data points.

4D Gridding

The functions FAST_GRID4 and GRID_4D return a gridded three-dimensional array containing intensity values for input data with four-dimensional coordinates. The FAST_GRID4 function works best with dense data points. The GRID_4D function works best with sparse data points.

Spherical Gridding

The GRID_SPHERE function returns a gridded, two-dimensional array containing radii, given random longitude, latitude, and radius values.

Polygon Manipulation

The polygon manipulation routines generate information to be used by the polygon rendering routines. These routines are discussed in detail in the *PV-WAVE Reference.*

- **POLY_C_CONV** — This function returns a list of colors for each polygon. The function requires a *polygon_list* and a list of colors for each vertex. The POLY_PLOT procedure uses data generated by this function.
- **POLY_COUNT** — This function returns the total number of polygons contained in a *polygon_list.* The total number of polygons is required as an input by the POLY_PLOT procedure.
- **POLY_MERGE** — This procedure merges two vertex lists and two polygon lists.
- **POLY_TRANS** — This function returns a list of 3D points transformed by a 4-by-4 transformation matrix.

Volume Manipulation

The two volume manipulation routines, VOL_PAD and VOL_TRANS prepare volumes for rendering.

- VOL_PAD — This function returns a three-dimensional volume of data padded on all six sides with zeroes. For example, if you are transforming a small volume inside a large volume using the VOL_TRANS function, then you should use the VOL_PAD function to pad the small volume with zeros. If you do not pad the small volume with zeros, the data points at the edge of the small volume will be duplicated to fill the space between the outer surfaces of the small volume and the inner surfaces of the large volume. See Figure 6-2. This function is often used to process volumes before using the VOL_TRANS and SLICE_VOL function.

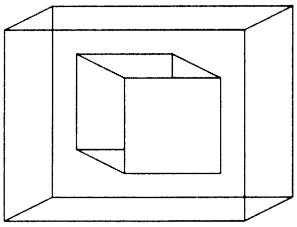


Figure 6-2 The VOL_PAD function pads the space between two volumes with zeros. Without VOL_PAD, data values from the outer edges of the small volume fill the empty space between the two volumes.

- VOL_TRANS — This function returns a three-dimensional volume of data transformed by a 4-by-4 matrix.
- SLICE_VOL — This function returns a two-dimensional array containing a slice from a three-dimensional volumetric array.

Coordinate Conversion

PV-WAVE provides several routines for converting data to various coordinate systems. Some of the rendering functions require that data be mapped to a particular coordinate system. The POLY_PLOT procedure requires a *vertex_list* with device coordinates. The POLYSHADE procedure must be in either data or normalized coordinates.

- CONV_TO_RECT — This function converts polar, cylindrical, or spherical coordinates to rectangular coordinates.
- CONV_FROM_RECT — This function converts rectangular coordinates to polar, cylindrical, or spherical coordinates.

- **POLY_NORM** — This function returns a list of three-dimensional points converted from data coordinates to normal coordinates. This function is often used in conjunction with the POLY_TRANS and POLY_DEV functions to transform a *vertex_list* that is used by some of the render functions.
- **POLY_TRANS** — This function returns a list of three-dimensional points transformed by a 4-by-4 transformation matrix. Like the POLY_NORM function, this function is used to transform a *vertex_list.*
- **POLY_DEV** — This function returns a list of three-dimensional points converted from normal coordinates to device coordinates. This function is often used in conjunction with the POLY_TRANS and POLY_NORM functions to transform a *vertex_list.*

Setting Up Data for Viewing

In some instances, you may need to set up your data for viewing before rendering. Several PV-WAVE routines that set up viewing are:

- **CENTER_VIEW** — This procedure sets system viewing parameters to display data in the center of the current window.
- **SET_VIEW3D** — This procedure generates a three-dimensional view given a view position and a view direction.
- **VIEWER** — This procedure lets you interactively define a three-dimensional view, a slicing plane, and multiple cut-away volumes.
- **T3D** — This is a Standard PV-WAVE library procedure. Refer to Chapter 2 *Procedure and Function Reference,* in the *PV-WAVE Reference.* This procedure is used by `sphere_demo3`, `vec_demo1`, `vec_demo2`, and `vol_demo1`. These demonstration programs are located in

`$WAVE_DIR/demo/arl`. This procedure is also used by all of the `gen_` demonstration programs in the `$WAVE_DIR/demo/render` directory.

Rendering with Standard Techniques

Once you have imported, generated, and set up your data for viewing, you are ready to render it. PV-WAVE provides routines to render both polygons and volumes. This section briefly describes these rendering routines, which are part of the standard library. For additional information on these routines, see the *PV-WAVE Reference*.

Polygon Rendering

The two polygon rendering routines are POLY_PLOT and POLYSHADE:

- POLY_PLOT — This procedure requires a *vertex_list*, a *polygon_list*, and the total number of polygons to plot. The procedure does not render polygons with light-source shading, but it can plot opaque and transparent polygons.
- POLYSHADE — This function constructs a shaded surface representation of one or more solids described by a set of polygons. This function also requires a *vertex_list* and a *polygon_list*.

Volume Rendering

A volume of data consists of intensity values represented at data points located by three-dimensional coordinates. There are three routines for rendering volumes.

- **VECTOR_FIELD3** — This procedure plots a three-dimensional vector field from three 3D arrays.
- **VOL_MARKER** — This procedure displays colored markers scattered throughout a volume.
- **VOL_REND** — This function renders volumetric data translucently.

Ray-tracing Rendering

The RENDER function lets you generate multiple images for a scene from five object types using a ray-tracing technique. For example, you can generate pictures of voxel data directly, without having to convert to a polygonal iso-surface representation. (Voxels are the 3D counterpart of a 2D pixel).

You can also simultaneously render volumes, polygonal meshes, and three kinds of quadric objects: cones, cylinders, and spheres.

- Volumes are applicable to any voxel processing domain, such as the visualization of astronomical, geological, and medical data.
- Polygonal meshes can be used for iso-surfaces, as well as spatial-structural data.
- Cones can be used for caps on axes.
- Cylinders can be used for molecular modeling (symbolizing bonds) as well as axes and 3D line generation.
- Spheres are applicable to spherical inverse ("rubber sheet") mapping as well as molecular modeling (atoms).

This section describes the lighting and color models used by the Renderer. It also explains how you specify objects to be rendered, including setting material properties and view transformations.

Specifying RENDER Objects

The five object types (primitives) supported by RENDER correspond to five functions that define these objects. They are summarized below and detailed in the *PV-WAVE Reference*.

- **CONE** – A conic primitive that is defined by default to be centered at the origin with a height of 1.0, and to have an upper radius of 0.5 and a lower radius of 0. The lower radius can be changed using the *Radius* keyword, while the upper radius can be changed using the *Scale* keyword with the T3D procedure.

The *Radius* keyword corresponds to a scaling factor in the range [0...1] which is multiplied by the upper radius to give the lower radius. For example, *Radius*=0.5 corresponds to a conic object whose lower radius is one-half of the upper radius, while *Radius*=0.0 corresponds to a point whose lower radius is 0 (a conic that ends in a point).

- **CYLINDER** – This is the similar to a CONE, except that the lower radius is the same as the upper radius (a CONE with *Radius*=1.0).
- **MESH** – A polygonal mesh primitive that uses the standard PV-WAVE list of vertices and polygons that are described in *Vertex Lists and Polygon Lists* on page 187.

  Note that any non-coplanar polygons in a mesh will automatically be reduced to triangles by RENDER.
- **SPHERE** – An ellipsoid primitive centered at the origin with a radius of 0.5.
- **VOLUME** – Volume data that uses a three-dimensional byte array. Each byte in the voxel array corresponds to an index into the material properties associated with the volume.

Lighting Model

The RENDER function uses a more complicated lighting model than that used by the other PV-WAVE routines. Under this new paradigm, the intensity value at a pixel is generated using a recursive shading function that is designed to imitate natural light.

Light rays are emitted from lights, bounce, and are then absorbed and possibly re-emitted with respect to objects in the scene; sometimes they reach (are visible to) the viewer (in this case, an image). This technique of rendering is called "ray tracing."

The components that comprise the color at a particular point on an object in a scene are a function of the material properties of the object at that point and the orientation of the object with respect to other objects, light sources, and the viewer.

The Renderer supports Lambertian diffusion, transparency, and ambient material properties for color, as detailed below.

Defining Color and Shading

The color at point P on an object is defined simply as:

$$Color(D + T + A)$$

where D represents the diffuse component, T represents the transmission component, and A represents the ambient component. These three shading components are defined below.

(PV-WAVE allows only a scalar value in the specification of color via the *Color* keyword. Thus, the term "intensity" is technically more accurate. However, the term "color" was chosen to allow for future enhancements.)

Diffuse Component

The diffuse component corresponds to a simple approximation of Lambertian shading where the resulting intensity at some point on an object is a function of the light incident at that point, the position of the associated light source(s), and the surface normal at that point.

The diffuse component is defined as:

$$D = Kdiff\sum_{i=0}^{nvl} I_{Li}(N_P \bullet L_{Pi})$$

where

$Kdiff$ is the diffuse reflectance coefficient.

I_{Li} is the intensity of light source i.

N_P is the unit surface normal at point P.

L_{Pi} is the unit vector from point P to the location of the point light source Li.

$\bullet$ is the vector dot product.

By default, *nvl* is the total number of lights. If the *Shadows* keyword is specified in the call to RENDER, then *nvl* is the number of visible light sources (possibly via transmission through objects) at point *P*.

Transmission Component

The transmission component is simply the light which has passed through the object at a particular point. For example, the color of a point on a glass ball is a combination of both the light striking the surface and the light which passes through it from the opposite side of the point. RENDER currently assumes that the refractive indices of all objects are the same.

The transmission component is defined as:

$$T = (Ktran \cdot T_i)\ (N_P \bullet T_N)$$

where

Ktran is the specular transmission coefficient.

T_i is the intensity of the light that is transmitted from other objects, assuming that all objects have a refractive index of 1 (air).

N_P is the unit surface normal at point *P*.

T_N is the calculated specular transmission microfacet normal from the direction of transmission.

• is the vector dot product.

Ambient Component

The ambient component of the resulting shaded color is completely independent of the position of objects and light sources. It is typically used alone (i.e., *Kdiff* and *Ktran* are 0) for flat shading and for rendering voxel values as intensities that correspond directly to their actual byte values.

The ambient component is defined as:

$$A = Kamb \sum_{i=0}^{nl} I_{Li}$$

where

Kamb is the ambient coefficient.

nl is the total number of light sources.

I_{Li} is the intensity of light source *i*.

Defining Object Material Properties

The following keywords can be used with each RENDER object:

- *Color* — The color (intensity) coefficient of the object.
- *Kamb* — The ambient coefficient (flat shaded).
- *Kdiff* — The diffuse reflectance coefficient.
- *Ktran* — The specular transmission coefficient.

Objects may have up to 256 material properties each; thus, an array of 256 double-precision floating-point values can be assigned to each keyword.

The defaults for these properties vary from object to object:

- For CONE, CYLINDER, SPHERE, and MESH, *Color* and *Kdiff* are all 1, while *Kamb and Ktran* are all 0. (This corresponds to `Color(0:255)=1.0` and `Ktran(0:255)=0.0` in PV-WAVE notation.)
- For VOLUME, *Color* are all 1, *Kdiff* and *Ktran* are all 0, and *Kamb* is an array of 256 linearly increasing values from 0 to 1.

CONE, CYLINDER, and SPHERE also support a *Decal* keyword that allows mapping of a byte image onto the surface of the object. The values in the image correspond to an index into the arrays of material properties defined above; thus, different regions on an object can have different properties.

For polygonal meshes, in addition to specifying a list of polygons, you can also specify a 1D array of bytes, one element for each polygon. This array is an index into the arrays of material properties defined above. This allows you to then use the *Materials* keyword to specify different properties for different polygons.

The actual value in the voxel array of bytes defining a VOLUME is used as an index into the arrays of material properties defined with the *Materials* keyword; thus, a voxel data set can be considered to be made up of as many as 256 voxel types.

Tip For best results, be sure that each *Color(Kamb+Kdiff+Ktran)* setting is in the range [0...1]. Otherwise, you must use the *Scale* keyword in the call to RENDER.

Decals

A decal is a 2D array (image) of bytes whose elements correspond to indices into the arrays of material properties. You can use the *Decal* keyword with the quadric objects.

For example, if a given point on an object is mapped to coordinates `(u,v)` in the decal image, then the material properties used at that point for shading would be `Color(Decal(u,v))`, `Kamb(Decal(u,v))`, `Kdiff(Decal(u,v))` and `Ktran(Decal(u,v))`. An example of applying a decal to a sphere is shown in *Example 4: Quadric Animation* on page 207.

Setting Object and View Transformations

The view that is automatically generated by RENDER is depicted in Figure 6-3. (You can retrieve this view with the *Info* keyword; for details, see the *PV‑WAVE Reference*.)

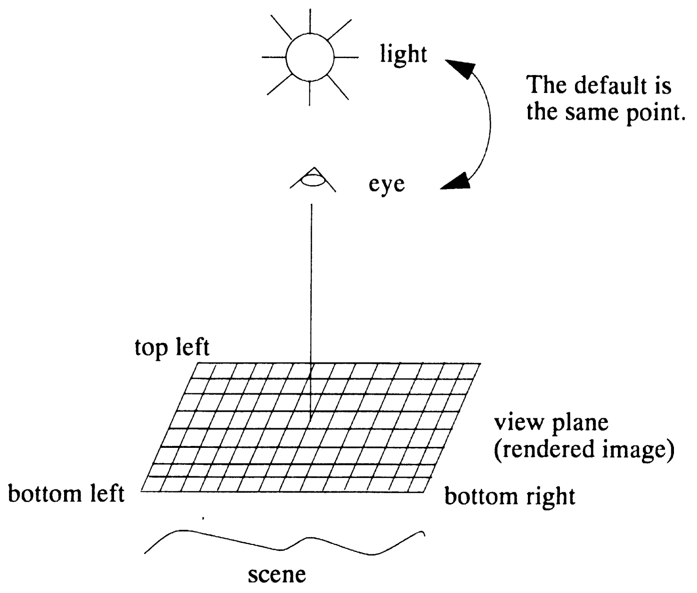


Figure 6-3 The default view used in RENDER positions the observer's eye on the positive Z axis, looking towards the origin into the scene with a slight perspective. All objects are visible in this default view.

You can use the *View* keyword with RENDER to specify a different view. This is especially useful for zooming in or for animations, since changes in scale can result if you use the default view in animations.

You can also use the *Transform* keyword with any object passed into RENDER. This keyword allows individual objects to be transformed (e.g., rotated, scaled, and positioned) separately from other objects in the scene. *Transform* contains the local transformation matrix whose default is the identity matrix:

```
1  0  0  0
0  1  0  0
0  0  1  0
0  0  0  1
```

Typically, you would build the transformation matrix by first using the *T3D* procedure and then using the system variable transformation matrix *!P.T.* Examples of using this method of matrix construction are shown throughout the *RENDER Examples* section starting on page 203.

For more information, see the section *Geometric Transformations* on page 167.

Invoking RENDER

RENDER is the function that generates the image from the objects you have specified. The general format is:

result = RENDER(object$_1$, ..., object$_n$)

where *object$_i$* is any number of objects previously-defined with the RENDER object functions.

RENDER returns a byte image of size X-by-Y, where X and Y each default to 256 unless overridden by the keywords *X* and *Y*. The returned image can then be displayed using either the TV or TVSCL procedure.

As illustrated in Figure 6-3 on page 202, RENDER automatically generates a default view. However, you may choose to use the *View* or *Transform* keywords to alter this default view.

Unless otherwise specified, a single-point light source is defined to coincide with the observer's viewpoint. The *Lights* keyword can be used to pass in an array of locations and intensities of point light sources.

For details on using the other RENDER keywords – *Sample, Scale, Shadow, X, Y,* and *Info* – see the description of this function in the *PV-WAVE Reference.*

RENDER Examples

The following examples were designed to show the capabilities of RENDER, rather than to depict typical applications. You can find most of the examples in this section on the distribution tape, along with the images they produce.

The location is `$WAVE_DIR/demo/render` for procedure files and `$WAVE_DATA` for data and image files.

Example 1: Polygonal Mesh (Diffusely-shaded Polygons)

This example constructs a polygonal mesh (iso-surface) of diffusely-shaded polygons. The default light source is at the eye-point.

Program Listing

```
PRO cube1
   verts = [[-1.0,-1.0,1.0], [-1.0,1.0,1.0], [1.0,1.0,1.0],$
      [1.0,-1.0,1.0], [-1.0,-1.0,-1.0], [-1.0,1.0,-1.0],$
      [1.0,1.0,-1.0], [1.0,-1.0,-1.0]]
   polys=[4,0,1,2,3, 4,4,5,1,0, 4,2,1,5,6, 4,2,6,7,3,$
      4,0,3,7,4, 4,7,6,5,4]
   m = MESH(verts, polys)
   T3D, /Reset, Rotate=[15.0,30.0,45.0]
   i = RENDER(m, x=512, y=512, Transform=!P.T)
   TV, i
END
```

Example 2: Polygonal Mesh (Flat-shaded Polygons)

This example constructs a polygonal mesh of flat-shaded polygons. Each polygon face has a different intensity, independent of the light source or the eye-point (which are the same here.)

Program Listing

```
PRO cube2
   verts = [[-1.0,-1.0,1.0], [-1.0,1.0,1.0], [1.0,1.0,1.0],$
      [1.0,-1.0,1.0], [-1.0,-1.0,-1.0], [-1.0,1.0,-1.0],$
      [1.0,1.0,-1.0], [1.0,-1.0,-1.0]]
   polys=[4,0,1,2,3, 4,4,5,1,0, 4,2,1,5,6, 4,2,6,7,3, $
      4,0,3,7,4, 4,7,6,5,4]
   amb = FLTARR(256)
   amb(0:5)=[.5, .3, .7, .9, .4, .1]
   m = MESH(verts, polys, Materials=[0,1,2,3,4,5], $
      Kdiff=FLTARR(256), Kamb=amb)
   T3D, /Reset, Rotate=[15.0,30.0,45.0]
```

```
    i = RENDER(m, x=512, y=512, Transform=!P.T)
    TV, i
END
```

Example 3: Polygonal Mesh (Many Polygons)

This example is a realistic application of polygonal meshes. It generates 52,500 polygons (approximately 98,000 triangles) as an iso-surface using the SHADE_VOLUME procedure. The polygons are then rendered.

The resulting image, shown in Figure 6-4, is saved to a file and is displayed using `show_iso_head.pro`.

Note, however, that it is not necessary to convert to a polygonal representation prior to rendering volumes; this is shown in Examples 5 through 8.

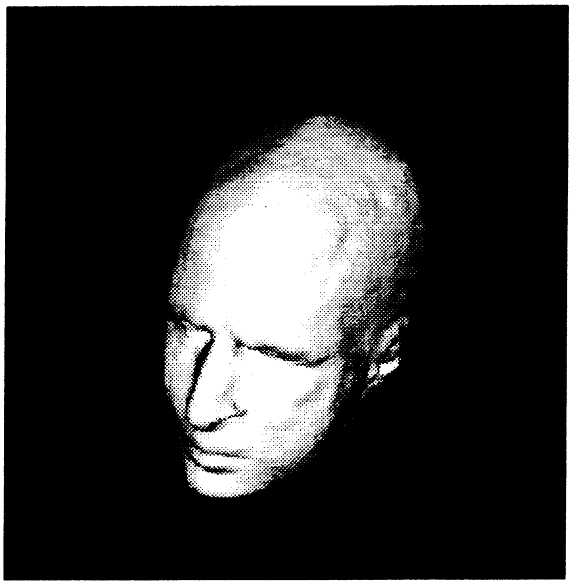


Figure 6-4 An example of polygonal meshes showing 52,000 polygons generated as an iso-surface and ray traced using the RENDER function.

Program Listing

```
PRO gen_iso_head
   volx = 115 & voly = 75 & volz = 105 ; volume data dimensions
   band = 5 ; neighborhood size of average filter

   dat = BYTARR(volx, voly, volz)
   OPENR, 1, !Data_Dir + 'man_head.dat'
   READU, 1, dat
   CLOSE, 1

   head = BYTARR(volx + 2 * band, voly + 2 * band, volz + 2 * band)
   head(band:band + volx - 1, band:band + voly - 1, $
      band:band + volz - 1) = dat
   head = SMOOTH(head, band) ; apply band ^ 3 average filter

   ; generate iso-surface
   SHADE_VOLUME, head, 18, vertex_list, polygon_list, /Low
   m = MESH(vertex_list, polygon_list)

   T3D, /Reset, Rotate=[60.0,0.0,-60.0]
   im = RENDER(m, x=512, y=512, Transform=!P.T)
   TV, im

   OPENW, 1, 'iso_head.img'
   WRITEU, 1, im
   CLOSE, 1
END
```

Program Listing

```
PRO show_iso_head
   im = BYTARR(512, 512)
   OPENR, 1, !Data_Dir + 'iso_head.img'
   READU, 1, im
   CLOSE, 1
   TV, im
END
```

Example 4: Quadric Animation

This example “constructs” a movie of an orbit around a sphere which has ocean temperature mapped on as a decal and a color lookup table applied from PV‑WAVE after generation of the movie.

If you wanted to add the boundaries of countries, you could do so by drawing them directly into the decal prior to calling SPHERE.

Note that the movie is saved to a file and is displayed using `show_anim.pro`.

Program Listing

```
PRO gen_anim

   ; Load the decal to apply.
   decal = BYTARR(720, 360)
   OPENR, 1, !Data_Dir + 'world_map.dat'
   READU, 1, decal
   CLOSE, 1

   ; Set shading to correspond directly to image values.
   dif = FLTARR(256)
   amb = FINDGEN(256)/255.

   T3D, /Reset, Rotate=[-90.0, 90.0, 0.0]
   c = SPHERE(Decal=decal, Kamb=amb, Kdiff=dif, $
      Transform=!P.T)

   ; Create an animation by orbiting view around the sphere.
   mve = BYTARR(256, 256, 72)
   FOR i=0, 71 DO BEGIN
      T3D, /Reset, Rotate=[-20.0, i*5.0, 0.0]
      mve(*, *, i) = RENDER(c, x=256, y=256, Transform=!P.T)
   ENDFOR


   OPENW, 1, !Data_Dir + 'world_anim.img'
   WRITEU, 1, mve
   CLOSE, 1
END
```

Program Listing

```
FUNC show_anim
   Window, 0, XSize=256, YSize=256, Colors=128, XPos=300,$
      YPos=50

   ; Create a color lookup table.
   red  = FLTARR(256)
   grn  = FLTARR(256)
   blu1 = FLTARR(256)
   blu2 = FLTARR(256)
   FOR i=0, 100 DO BEGIN
      fi = FLOAT(i)
      red(i)  = (-((ABS(fi - 100.0)^2.00)))
      grn(i)  = (-((ABS(fi -  50.0)^1.50)))
      blu1(i) = (-((ABS(fi -  25.0)^1.00)))
      blu2(i) = (-((ABS(fi - 100.0)^0.50)))
   ENDFOR
   red = BYTSCL(red)
   grn = BYTSCL(grn)
   blu = BYTSCL(blu1) > BYTSCL(blu2)
   TVLCT, red, grn, blu, 0

          white = 127 & TVLCT, 255, 255, 255, white
   light_yellow = 126 & TVLCT, 255, 255, 127, light_yellow
   light_purple = 125 & TVLCT, 255, 127, 255, light_purple
     light_cyan = 124 & TVLCT, 127, 255, 255, light_cyan
         yellow = 123 & TVLCT, 255, 255, 000, yellow
         purple = 122 & TVLCT, 255, 000, 255, purple
           cyan = 121 & TVLCT, 000, 255, 255, cyan
      light_red = 120 & TVLCT, 255, 127, 127, light_red
    light_green = 119 & TVLCT, 127, 255, 127, light_green
     light_blue = 118 & TVLCT, 127, 127, 255, light_blue
      greenish_red = 117 & TVLCT, 255, 127, 000, greenish_red
      redish_green = 116 & TVLCT, 127, 255, 000, redish_green
       redish_blue = 115 & TVLCT, 127, 000, 255, redish_blue
        bluish_red = 114 & TVLCT, 255, 000, 127, bluish_red
      bluish_green = 113 & TVLCT, 000, 255, 127, bluish_green
     greenish_blue = 112 & TVLCT, 000, 127, 255, greenish_blue
               red = 111 & TVLCT, 255, 000, 000, red
             green = 110 & TVLCT, 000, 255, 000, green
```

```
              blue = 109 & TVLCT, 000, 000, 255, blue
              gray = 108 & TVLCT, 127, 127, 127, gray
       dark_yellow = 107 & TVLCT, 127, 127, 000, dark_yellow
       dark_purple = 106 & TVLCT, 127, 000, 127, dark_purple
         dark_cyan = 105 & TVLCT, 000, 127, 127, dark_cyan
          dark_red = 104 & TVLCT, 127, 000, 000, dark_red
        dark_green = 103 & TVLCT, 000, 127, 000, dark_green
         dark_blue = 102 & TVLCT, 000, 000, 127, dark_blue
            black1 = 101 & TVLCT, 000, 000, 000, black1
             black = 000 & TVLCT, 000, 000, 000, black

   EMPTY

   ; Load the previously generated animation.
   frames = BYTARR(256, 256, 72)
   OPENR, 1, !Data_Dir + 'world_anim.img'
   READU, 1, frames
   CLOSE, 1

   ; Display the animation.
   MOVIE, frames, Order=0
   RETURN, frames
END
```

Example 5: Slicing a Volume

This example renders selected slices from a large amount of volume data. The resulting image, shown in Figure 6-5, is saved to a file and displayed using `show_slic_head`.

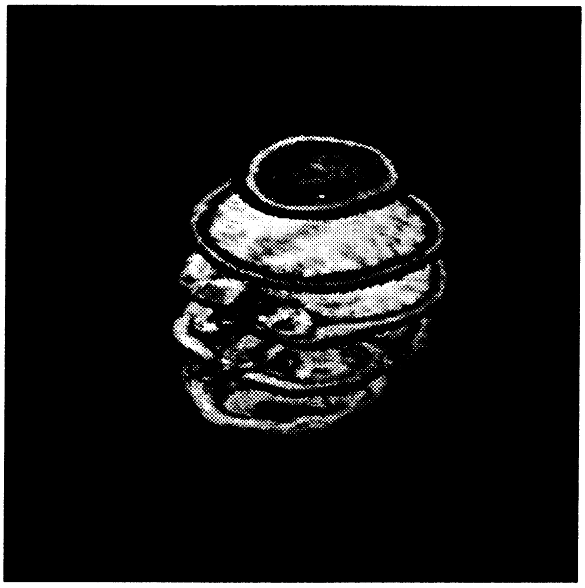


Figure 6-5 After slices have been rendered from a large quantity of volume data for this example, the resulting pixel intensity values show the actual density values of the voxel data.

Program Listing

```
PRO gen_slic_head
   width = 125 & height = 85 & depth = 115

   ; Use the procedure load_seg_head.pro to load the byte
   ; voxel data, set all data outside the head to zero,
   ; return the "segmented head" as HEAD, and return the
   ; thresholded surface of the head as SKULL.
   load_seg_head, head, skull
   ; Generate the slices of segmented data we wish to view.
   vox = BYTARR(width, height, depth)
   FOR i=0,depth-2,20 DO BEGIN
      vox(*,*,i) = head(*,*,i)
      vox(*,*,i+1) = head(*,*,i+1)
   ENDFOR

   v = VOLUME(vox)

   T3D, /Reset, Rotate=[60.0,0.0,-45.0]
   im = RENDER(v, x=512, y=512, Transform=!P.T, /Scale)
   TV, im

   OPENW, 1, !Data_Dir + 'sliced_head.img'
   WRITEU, 1, im
   CLOSE, 1
END
```

Program Listing

```
PRO show_slic_head
   im = BYTARR(512, 512)
   OPENR, 1, !Data_Dir + 'sliced_head.img'
   READU, 1, im
   CLOSE, 1
   TV, im
END
```

Example 6: Rendering an Iso-Surface with Voxel Values

This example renders a diffuse iso-surface using actual voxel values. The results, shown in Figure 6-6, are saved to a file and displayed using `show_flat_head`.

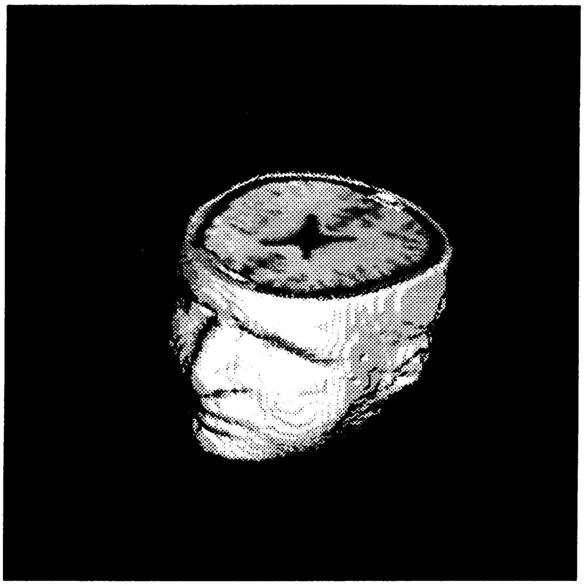


Figure 6-6 This example renders a diffuse iso-surface using actual voxel values. The surface of the head is shaded using diffusion, and the intensity values on top correspond directly to the voxel density values.

Program Listing

```
PRO gen_flat_head
   width = 125 & height = 85 & depth = 115

   ; Use the procedure load_seg_head.pro to load the byte
   ; voxel data, set all data outside the head to zero,
   ; return the 'segmented head' as HEAD, and return the
   ; thresholded surface of the head as SKULL.
   load_seg_head, head, skull
```

```
    ; Remove portion of head that overlaps with skull.
    overlap = skull * head
    overlap(where(overlap GT 0)) = 1
    head = head * (BYTE(1) - overlap)

    ; Generate the slices of smoothed data we wish to view.
    vox = BYTARR(width, height, depth)
    FOR i=0,76 DO vox(*,*,i) = $
       head(*, *, i) + (skull(*, *, i)*BYTE(255))

    ; Voxel value 255 is special, representing the skull surface.
    diff = FLTARR(256) & diff(255) = 0.6
    amb = FINDGEN(256)/255.0 & amb(255) = 0.0

    v = VOLUME(vox, Kdiff=diff, Kamb=amb)

    T3D, /Reset, Rotate=[60.0,0.0,-45.0]
    im = RENDER(v, x=512, y=512, Transform=!P.T, /Scale)
    TV, im

    OPENW, 1, !Data_Dir + 'flat_head.img'
    WRITEU, 1, im
    CLOSE, 1
END
```

Program Listing

```
PRO show_flat_head
   im = BYTARR(512,512)
   OPENR, 1, !Data_Dir + 'flat_head.img'
   READU, 1, im
   CLOSE, 1
   TV, im
END
```

Example 7: Diffuse and Partially Transparent Iso-Surfaces

This example renders a diffuse iso-surface and a partially transparent iso-surface. The results, shown in Figure 6-7, are saved to a file and displayed using `show_tran_head`.

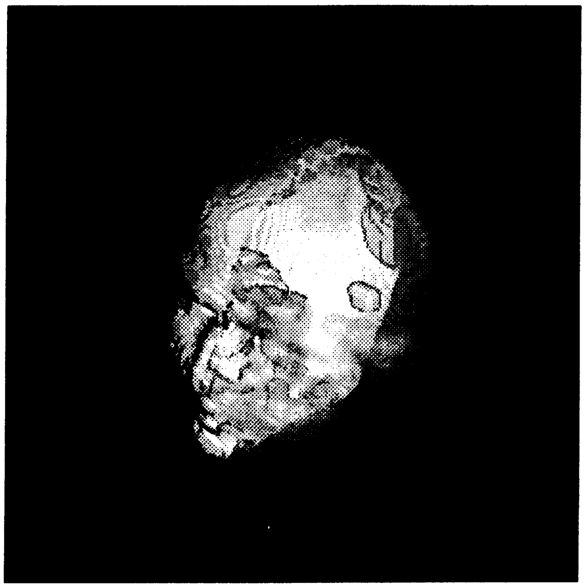


Figure 6-7 The voxel values within the iso-surfaces are completely transparent in this example, which renders a diffuse iso-surface and a partially transparent iso-surface.

Program Listing

```
PRO gen_tran_head
   width = 125 & height = 85 & depth = 115

   ; Use the procedure load_seg_head.pro to load the byte
   ; voxel data, set all data outside the head to zero,
   ; return the 'segmented head' as HEAD, and return the
   ; thresholded surface of the head as SKULL.
   ; See the file $WAVE_DIR/demo/render/load_seg_head.pro
   load_seg_head, head, skull
   ; Generate a mask plane that will split head in half,
   ; allowing half to be diffuse and rest to be transparent.
   mask = BYTARR(width, height, depth)
   mask(*, height/2:*, *) = 1

   ; Half surface = 255, other half = 254
   shell = skull * mask * BYTE(255) + $
      skull * (BYTE(1) - mask) * BYTE(254)

   ; Remove portion of head that overlaps with skull.
   overlap = skull * head
   overlap(WHERE(overlap GT 0)) = 1
   head = head * (BYTE(1) - overlap)

   vox = shell + head

   ; Voxel value 255 is special, corresponding to surface of
   ; half head. Value 254 corresponds to surface of other
   ; half. Remaining values are actual unsmoothed head data
   ; and are not used for this example (i.e., they are
   ; completely transparent).
   diff = FLTARR(256) & diff(255) = 1.0 & diff(254) = 0.05

   tran = FLTARR(256) & tran(*) = 1.0 & tran(255) = 0.0
   tran(254) = 0.95

   v = VOLUME(vox, Ktran=tran, Kamb=FLTARR(256), Kdiff=diff)

   T3D, /Reset, Rotate=[60.0, 0.0, -45.0]
   im = RENDER(v, x=512, y=512, Transform=!P.T)
   TV, im

   OPENW, 1, !Data_Dir + 'trans_head.img'
   WRITEU, 1, im
   CLOSE, 1
END
```

Program Listing

```
PRO show_tran_head
   im = BYTARR(512, 512)
   OPENR, 1, !Data_Dir + 'trans_head.img'
   READU, 1, im
   CLOSE, 1
   TV, im
END
```

Example 8: Rendering Iso-Surfaces with Transformation Matrices

This example renders two diffuse iso-surfaces as well as actual voxel values. The results, shown in Figure 6-8, are saved to a file and displayed using `show_core_head`.

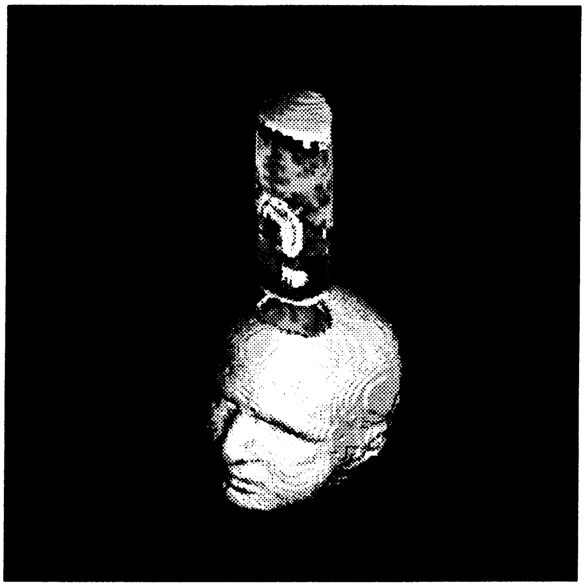


Figure 6-8 Two separate volumes are rendered simultaneously in this example, each using a different transformation matrix.

Program Listing

```
PRO gen_core_head
   width = 125 & height = 85 & depth = 115

   ; Use the procedure load_seg_head.pro to load the byte
   ; voxel data, set all data outside the head to zero,
   ; return the 'segmented head' as HEAD, and return the
   ; thresholded surface of the head as SKULL.
   load_seg_head, head, skull

   ; Remove portion of head that overlaps with skull.
   overlap = skull * head
   overlap(WHERE(overlap GT 0)) = 1
   head = head * (BYTE(1) - overlap)

   vox = head + (skull * BYTE(255))

   ; Create a circle (used for CYLINDER) mask plane.
   circle = BYTARR(width, height)
   radius2 = 16 * 16
   FOR x=0, width-1 DO BEGIN
      dx = x - width / 2
      dx = dx * dx
      FOR y=0, height-1 DO BEGIN
         dy = y - height / 2
         dy = dy * dy
         IF ((dx + dy) LE radius2) THEN BEGIN
            circle(x, y) = 1
         ENDIF
      ENDFOR
   ENDFOR

   ; Mask out the core sample and "subtract" out from slices.
   core = BYTARR(width, height, depth)
   FOR z=0, depth-1 DO BEGIN
      core(*, *, z) = vox(*, *, z) * circle
      vox(*, *, z) = vox(*, *, z) - core(*, *, z)
   ENDFOR
   ; Voxel value 255 is special, representing the skull surface.
   diff = FLTARR(256) & diff(255) = 0.6
   amb = FINDGEN(256)/255.0 & amb(255) = 0.0

   ; Surface and interior of skull.
   v0 = VOLUME(vox, Kdiff=diff, Kamb=amb)

   T3D, /Reset, Translate=[0.0,0.0,1.0]

   ; Core sample.
   v1 = VOLUME(core, Transform=!P.T, Kdiff=diff, Kamb=amb)

   T3D, /Reset, Rotate=[60.0,0.0,-45.0]
   im = RENDER(v0, v1, x=512, y=512, Transform=!P.T, /Scale)
   TV, im

   OPENW, 1, !Data_Dir + 'core_head.img'
   WRITEU, 1, im
   CLOSE, 1
END
```

Program Listing

```
PRO show_core_head
   im = BYTARR(512, 512)
   OPENR, 1, !Data_Dir + 'core_head.img'
   READU, 1, im
   CLOSE, 1
   TV, im
END
```

Displaying Rendered Images

Many of the rendering routines both render and display images. However, three rendering functions – POLYSHADE, VOL_REND and RENDER – use the Standard Library procedures TV and TVSCL to display rendered images.

Example programs that demonstrate this usage are listed below:

- See `sphere_demo3` for an example of using TVSCL with the POLYSHADE function.
- See `sphere_demo2` for an example of using TV with the POLYSHADE function.
- See `vol_demo2` for an example of using TVSCL with the VOL_REND function.
- See the programs in the section *RENDER Examples* on page 203 for examples of using TV and TVSCL with the RENDER function.

7

CHAPTER

Working with Date/Time Data

Data often follows a regular pattern related to the dates and times on which business is conducted or measurements are recorded. This data is often represented in relation to several levels of date/ time information such as seconds, minutes, hours, days, weeks and years. In conjunction with the PLOT and OPLOT procedures, PV-WAVE's Date/Time routines let you generate two-dimensional plots that display multiple levels of labeling for the date/ time axis.

Introduction to Date/Time Data

PV-WAVE's Date/Time feature provides a precise method for creating two-dimensional plots with Date/Time data represented on the X axis. Once you have generated PV-WAVE Date/Time data, you can create plots that reflect various levels of time intervals. The PLOT procedure automatically draws and labels the Date/ Time axis. Figure 7-1 illustrates a plot with two levels of Date/ Time labeling:

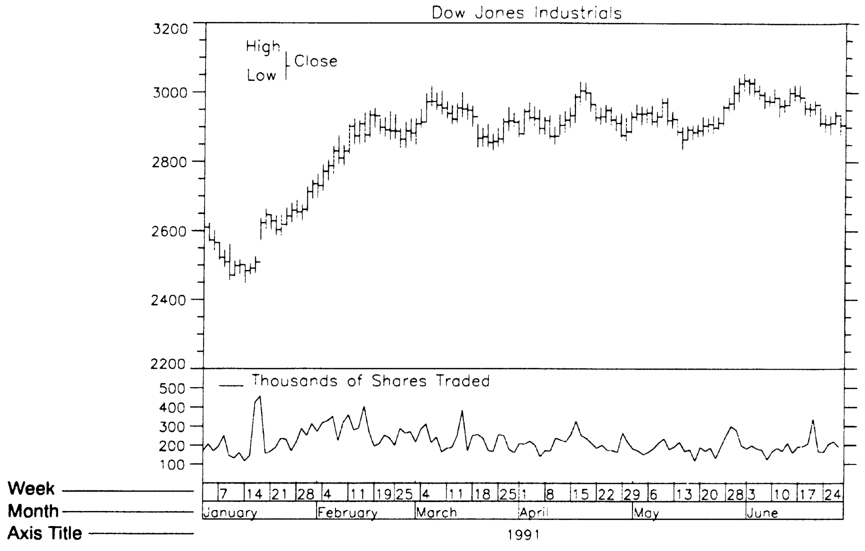


Figure 7-1 PV-WAVE Date/Time Plot

The PV-WAVE Date/Time axis is well-suited for the display of data that follows an hourly, daily, weekly, or monthly pattern; financial and meteorological data are two examples of this type of data. By default, PV-WAVE labels a Date/Time axis with up to six levels of tick labels that show the time frame of the data that is being displayed.

The four basic steps for creating a Date/Time plot are:

- ❑ Read data into PV-WAVE.
- ❑ Convert data representing dates and/or times to PV-WAVE Date/Time data.
- ❑ Manipulate the PV-WAVE Date/Time data (optional).
- ❑ Plot the data.

Reading in Your Data

Read your data from an input file into PV-WAVE using a command such as DC_READ_FREE, DC_READ_FIXED, READF or READU.

The DC_READ_FIXED and DC_READ_FREE functions can be used with the *DT_Template* keyword to read data directly into PV-WAVE Date/Time variables. See the descriptions for the DC_READ_FIXED and DC_READ_FREE functions in the *PV-WAVE Reference* for detailed information on these routines and examples of their use.

The READU and READF procedures can be used to read dates/times into atomic PV-WAVE data types, which must then be converted into PV-WAVE Date/Time variables. For a complete account of these input procedures, see Chapter 8, *Working with Data Files*, in the *PV-WAVE Programmer's Guide.*

Converting the Data to the PV-WAVE Date/Time Format

If you read your data into PV-WAVE with the READU or READF procedures, you must use conversion functions to convert the date/time information into PV-WAVE Date/Time variables.

There are four functions that you can use to convert your Date/Time data: STR_TO_DT, VAR_TO_DT, SEC_TO_DT, and JUL_TO_DT. The function you use depends on the configuration of the data you are reading in. See *Converting Your Data into Date/Time Data* on page 229 for details.

Tip In some instances, your input file may not contain explicit date/time information. You can generate a scalar PV-WAVE Date/Time variable with one of the conversion functions and then use the DTGEN function to create a PV-WAVE Date/Time variable containing an array of Date/Time structures. See *Generating PV-WAVE Date/Time Data* on page 234 for details. Also see Examples 1, 2, and 3 in *Creating Plots With PV-WAVE Date/Time Data* on page 243.

Manipulating the PV-WAVE Date/Time Data

After you have created PV-WAVE Date/Time data, you may want to alter it. PV-WAVE provides two functions, DT_ADD and DT_SUBTRACT, to add to or subtract date/time intervals from a PV-WAVE Date/Time variable. You may also want to eliminate holidays and weekends from your data with the CREATE_HOLIDAYS and CREATE_WEEKENDS procedures. See *Manipulating Date/Time Data* on page 236 for details.

Plotting Your Data

You can plot your Date/Time data with PLOT or OPLOT. PV-WAVE automatically generates labels and tick marks for your Date/Time data. If you want to modify the appearance of the PV-WAVE Date/Time axis, PV-WAVE provides several keywords. See *Creating Plots With PV-WAVE Date/Time Data* on page 243 for details.

The PV-WAVE Date/Time Structure

Date/Time data is stored in a PV-WAVE structure (!DT) containing the following fields:

Table 7-1: Fields of the !DT Structure

Element	Data Type	Valid Range
!DT.Year	integer	0 to 9999
!DT.Month	byte	1 to 12
!DT.Day	byte	1 to 31
!DT.Hour	byte	0 to 23
!DT.Minute	byte	0 to 59
!DT.Second	floating point	0.0000 to 59.9999
!DT.Julian	double precision	The number of days calculated from September 14, 1752. The decimal part contains the time as a fraction of a day.
!DT.Recalc	byte	Recalculation flag: setting this flag to 1 forces the julian day to be recalculated.

For example:

```
date = {!dt, 1992,4,27,7,45,40.0,87519.323,0}
PRINT, date
{ 1992 4 27 7 45 40.0000 87519.323 0}
```

For more information on structures, see Chapter 6, *Working with Structures*, in the *PV-WAVE Programmer's Guide.*

The Julian Field

PV-WAVE uses the Julian field to perform many Date/Time calculations. A Date/Time value is interpreted as a day in a series of days that begins on September 14, 1752. For example, 2 is equated with September 15, 1752. The decimal part of the Julian day indicates the time as a portion of the day. For example, for May 1, 1992 at 8:00 a.m, the Julian day is 84702.333.

The Recalc Field

If you modify a PV-WAVE Date/Time variable directly by assigning a new value to one of its elements, you must also set the Recalc flag (the last element of the Date/Time structure) to 1. This recalculates the Julian day for the new date. For example, for a PV-WAVE Date/Time variable `date` that looks like:

```
date = {!dt, 1992, 4, 27, 7, 45, 40.0,
   87519.323, 0}
```

If you add three days to this variable by assigning a new value to `date.day` directly.

```
date.day = 30
```

The new value of `date` is:

```
PRINT, date
{ 1992 4 30 7 45 40.0000 87519.323 0}
```

Notice that the Julian field 87519.323 has not changed. You must set the recalc flag to 1 for `date` to obtain the correct Julian day:

```
date.recalc = 1
```

The Julian date is then recalculated automatically when the PV-WAVE Date/Time variable is used with any of the Date/Time functions.

Tip Rather than modifying a PV-WAVE Date/Time variable by assigning a new value to one of its elements, you should use the DT_ADD and DT_SUBTRACT functions to create new variables. If you use these functions, the Julian day is automatically recalculated.

Creating Empty PV-WAVE Date/Time Variables

Normally, you create PV-WAVE Date/Time variables using the conversion functions or the DC_READ functions. However, you can also create an "empty" PV-WAVE Date/Time variable by assigning !DT to a variable name. Here are a couple of examples:

```
date = {!DT}
```

Creates a PV-WAVE Date/Time structure filled with zeros.

```
PRINT, date
{ 0 0 0 0 0 0.00000 0.0000000 0}
```

```
date1 = REPLICATE({!DT}, 3)
```

Creates 3 structures filled with zeros.

```
PRINT, date1
{ 0 0 0 0 0 0.00000 0.0000000 0}
{ 0 0 0 0 0 0.00000 0.0000000 0}
{ 0 0 0 0 0 0.00000 0.0000000 0}
```

Note When you use the DC_READ functions with the *DT_Template* keyword to import and convert data, you must use this REPLICATE method to create an "empty" array variable containing Date/Time structures. Once you have created this array variable, you can read Date/Time data from a file into the variable. See *Creating Plots With PV-WAVE Date/Time Data* on page 243 for examples.

Reading in Your Date/Time Data

Before you can generate a Date/Time axis on a plot, your data must be read in and converted to PV-WAVE Date/Time data. There are three methods for generating variables containing PV-WAVE Date/Time data:

- You can read data directly into PV-WAVE Date/Time variables using the DC_READ functions in conjunction with the *DT_Template* keyword. See *Importing Date/Time Data* on page 162 of the *PV-WAVE Programmer's Guide* for an example.

  Refer to *Transferring Date/Time Data* on page 168 of the *PV-WAVE Programmer's Guide*; this section contains an example showing date/time data being transferred using DC_READ_FIXED. You can also refer to the descriptions for the DC_READ_FIXED and DC_READ_FREE procedures in the *PV-WAVE Reference* for other examples.

- You can read date and time data as atomic data types and then use conversion procedures to create PV-WAVE Date/Time variables. These conversion routines are discussed in the next section.

  Refer to *Transferring Date/Time Data* on page 168 of the *PV-WAVE Programmer's Guide*; this section contains an example showing date/time data being transferred using the READF function. You can also refer to the descriptions of the READF and READU functions in the *PV-WAVE Reference* for more examples.

- You can use the DTGEN function to generate PV-WAVE date/time data for an input file that does not contain date/time information such as data generated by a computer time stamp. See *Generating PV-WAVE Date/Time Data* on page 234.

Converting Your Data into Date/Time Data

If you are importing Date/Time data into PV-WAVE, four functions simplify converting this data into PV-WAVE Date/Time data. These functions are:

- **STR_TO_DT** – Converts string data or variables containing string data into PV-WAVE Date/Time variables.
- **VAR_TO_DT** – Converts numeric variables containing Date/Time information into PV-WAVE Date/Time variables.
- **SEC_TO_DT** – Converts seconds into PV-WAVE Date/Time variables.
- **JUL_TO_DT** – Converts the Julian day into a PV-WAVE Date/Time variable.

Error checking is performed by these conversion functions to verify that numbers assigned to the Date/Time structure elements fall within valid ranges. For more information about these functions, see *PV-WAVE Reference*.

Note If you read and converted your data with the DC_READ routines, you do not need to use these functions to convert your data.

The STR_TO_DT Function

This function converts date and time data stored as strings into PV-WAVE Date/Time variables. The function has the form:

result = STR_TO_DT(*date_strings* [, *time_strings*])

The *Date_Fmt* and *Time_Fmt* keywords are used to describe the format of the input string data by specifying a template to use as the data is read. These templates are listed in Table 7-2.

Table 7-2: Valid Date Formats for STR_TO_DT Function

Keyword Value	Template Description	Examples for May 1, 1992
1	MM*DD*[YY]YY	05/01/92
2	DD*MM*[YY]YY	01-05-92
3	ddd*[YY] YY	122,1992
4	DD*mmm[mmmmmm]*[YY]YY	01/May/92
5	[YY]YY*mm*DD	1992-01-01

The abbreviations used in the template descriptions are:

MM – The numerical month. The month does not need to occupy two spaces. For example, you can enter a 1 for the month of January.

DD – The numerical day of the month. The day does not need to occupy two spaces. For example, for May 5, the numerical day can be 5.

[YY]YY – The numerical year. For example, 1992 can be entered as 92 or 1992.

ddd æ The numerical day of the year. The day does not need to occupy three spaces. For example, February 1 is 32.

mmm[mmmmmm] – The full name of the month or its abbreviation depending on how the system variable !Month_Names is set.

* – Represents a delimiter that separates the different fields of data. The delimiter can also be a slash (/), a colon (:), a hyphen (–), period (.), or a comma (,).

Table 7-3: Valid Time Formats for STR_TO_DT function

Keyword Value	Template Description	Examples for 1:30 p.m.
–1	HH*Mn*SS[.SSS]	13:30:35.25
–2	HHMn No separators are allowed between hours and minutes. Both hours and minutes must occupy two spaces.	1330

The abbreviations used in the template descriptions are:

HH – The numerical hour based on a 24-hour clock. For example, 14 is 2 o'clock in the afternoon. For the –1 format, both spaces do not need to be occupied. However, the – 2 format requires that both spaces be occupied. For example, 1:00 in the morning must be entered as 01.

Mn – The number of minutes in the hour. For the –1 format, both spaces do not need to be occupied. However, the –2 format requires that both spaces be occupied. For example, 6 minutes must be entered as 06.

SS[.SSS] – The number of seconds in the minute. A decimal part of a second is optional.

* – Represents a delimiter that separates the different fields of data. The delimiter can also be a slash (/), a colon (:), a hyphen (–), or a comma (,).

Note You do not need both a date and time to use the STR_TO_DT function. You can enter a date only or a time only. For more information, refer to the STR_TO_DT function in the *PV-WAVE Reference*.

Example 1

```
date2 = STR_TO_DT('3-13-92', $
   '14:12:22', Date_Fmt=1, Time_Fmt=-1)
```

The data contained in the strings corresponds to the date format MM DD YY and the time format HH Mn SS.

```
DT_PRINT, date2
03/13/1992 14:12:22
```

Example 2

```
date3 = STR_TO_DT('4-12-92', Date_Fmt=1)
```

You can convert a date without a time.

```
DT_PRINT, date3
04/12/92
```

The VAR_TO_DT Function

If you have read Date/Time elements into numeric PV-WAVE variables, you can use the VAR_TO_DT function to convert these variables into PV-WAVE Date/Time variables. This function is useful for converting time stamp data that does not conform to a format used by the STR_TO_DT function.

This function has the form:

result = VAR_TO_DT(*yyyy, mm, dd, hr, mn, ss*)

Example

This example illustrates how to convert a numeric date/time value into PV-WAVE Date/Time data and verify that a PV-WAVE Date/Time variable has been created using the PRINT procedure.

```
z = VAR_TO_DT(1992, 11, 22, 12, 30)
PRINT, z
{ 1992 11 22 12 30 0.00000 87728.521 0}
```

The SEC_TO_DT Function

In some instances, scientific and engineering data has been collected at regular intervals over long periods of time from a specified start date. Some examples include sun spot activity or seismic data about an active volcano. The SEC_TO_DT function is designed to handle this type of data. It converts any number of seconds into PV-WAVE Date/Time variables. These variables are calculated from a specified base time. The default base, September 14, 1752, is defined by the system variable !DT_Base. You can change the base time by using the keyword *Base*.

This function has the form:

result = SEC_TO_DT(*num_of_seconds*)

Example

The example shows how to convert 20 seconds to a PV-WAVE Date/Time variable. The example uses a base start date of January 1, 1970.

```
date = SEC_TO_DT(20, Base='1-1-70', $
   Date_Fmt=1)
PRINT, date
{ 1970 1 1 0 0 20.0000 79367.000 0}
```

The JUL_TO_DT Function

This function converts a Julian number into a PV-WAVE Date/Time variable. For more information on how to use this function with the PV-WAVE table functions, refer to the examples in the section *Using Date/Time Data in Tables* on page 281.

This function has the form:

result = JUL_TO_DT(*julian_date*)

Example

```
dt = JUL_TO_DT(87507)
```

Converts the Julian day 87507 to a PV-WAVE Date/Time variable.

```
PRINT, dt
{ 1992 4 15 0 0 0.00000 87507.000 0}
```

Generating PV-WAVE Date/Time Data

You can generate PV-WAVE Date/Time data for data files that do not have date and time stamps. There are two steps:

- ❑ Create an initial PV-WAVE Date/Time structure using one of four conversion functions: STR_TO_DT, VAR_TO_DT, SEC_TO_DT, or JUL_TO_DT.
- ❑ Use the DTGEN function to create a variable from the original function that contains an array of PV-WAVE Date/Time structures.

The DTGEN function has the basic form:

result = DTGEN(*dt_start, dimension*)

Example 1

Assume that you have a file that contains seismic data collected on an hourly basis for the month of April, 1992. The file contains the seismic data, but does not have a time stamp appearing with each data entry. The file looks like:

```
Seismic data
1.03
2.04
1.33
4.45
.
.
.
```

The first data entry (1.03) was taken at 1:00 a.m on April 1, 1992. Each successive entry was taken on an hourly basis for the rest of the month. To generate PV-WAVE Date/Time data for all of the hours of the month:

```
datel = VAR_TO_DT(92,4,1,1)
```

Use VAR_TO_DT to create the initial PV-WAVE Date/Time variable for April 1, 1992, 1:00 a.m.

```
PRINT, datel
{ 1992 4 1 1 0 0.00000 87493.042 0}
```

```
dtarray = DTGEN(datel, 720, /hour)
```

Generate a PV-WAVE Date/Time array variable that contains a PV-WAVE Date/Time structure for every hour in the month of April (24 hours * 30 days =720 hrs).

```
PRINT, dtarray
{ 1992 4 1 1 0 0.00000 87493.042 0}
{ 1992 4 1 2 0 0.00000 87493.083 0}
{ 1992 4 1 3 0 0.00000 87493.125 0}
.
.
{ 1992 5 1 0 0 0.00000 87523.000 0}
```

Example 2

You can also use the *Compress* keyword with the DTGEN function. This example creates a PV-WAVE Date/Time variable that contains all of the weekdays for the month of January.

```
datel = VAR_TO_DT(1992,1,1)
```

Creates an initial PV-WAVE Date/Time variable to use with the DTGEN function.

```
CREATE_WEEKENDS, ['sat', 'sun']
```

Defines the weekend days.

```
dates = DTGEN(datel, 23, /Compress)
```

Generates a PV-WAVE Date/Time variable that contains the weekdays for January. The Compress keyword excludes the weekend days.

```
DT_PRINT, dates
01/01/1992
01/02/1992
01/03/1992
01/06/1992
01/07/1992
        .
        .
```

Notice that the 4th and 5th of January have been removed (compressed) from the result. These days fall on Saturday and Sunday.

Manipulating Date/Time Data

PV-WAVE provides several functions for manipulating PV-WAVE Date/Time variables. These functions are:

- DT_ADD
- DT_SUBTRACT
- DT_DURATION
- CREATE_WEEKENDS
- CREATE_HOLIDAYS
- LOAD_HOLIDAYS
- LOAD WEEKENDS
- DT_COMPRESS

Once you have converted your Date/Time data, you may want to alter it. The manipulation functions provide you with the tools for adding or subtracting Date/Times, or removing holidays and weekends from your PV-WAVE Date/Time variables. This section briefly describes each of these functions. For more information about these functions, see *PV-WAVE Reference*.

Adding to a PV-WAVE Date/Time Variable

You may wish to add any number of Date/Time units to one or more existing PV-WAVE Date/Time variables with the DT_ADD function. The form of the function is:

result = DT_ADD(*dt_value*)

Example 1

This example illustrates how to add 30 hours to a single PV-WAVE Date/Time variable to produce a new variable.

```
dtvar = VAR_TO_DT(1992, 12, 31, 15)
```
Create a PV-WAVE Date/Time variable.

```
dtvar1= DT_ADD(dtvar,Hour=30)
```
Create a new PV-WAVE Date/Time variable by adding thirty hours to dtvar, an existing PV-WAVE Date/Time variable.

```
PRINT, dtvar1
{ 1993 1 1 21 0 .0000 87768.875 0}
```

Example 2

The second example shows how to use the DT_ADD function to create a PV-WAVE Date/Time variable that contains all the days of the month of May excluding weekends.

```
dates = REPLICATE({!DT}, 21)
```
Creates a PV-WAVE Date/Time variable to read date/time data into.

```
CREATE_WEEKENDS, ["sun", "sat"]
```
Defines Saturday and Sunday as weekend days.

```
dates(0) = VAR_TO_DT(1992, 5, 1)
```
Creates an initial PV-WAVE Date/Time variable to use with DT_ADD.

```
FOR I = 1,20 DO $
   dates(I)=DT_ADD(dates(I-1), /day, $
   /Compress)
```

Generates PV-WAVE Date/TIme structures for the remaining days of the month. The Compress keyword excludes the weekend days.

Subtracting From a PV-WAVE Date/Time Variable

The function DT_SUBTRACT subtracts a value from a PV-WAVE Date/Time variable or array of variables. (This function is very similar to DT_ADD.) The basic form of the function is:

result = DT_SUBTRACT(*dt_value*)

Example

```
dtvar = VAR_TO_DT(1993, 1, 1, 21)
```

Create a PV-WAVE Date/Time variable.

```
dtvar1= DT_SUBTRACT(dtvar, Hour=30)
```

Create a new PV-WAVE Date/Time variable by subtracting 30 hours from dtvar.

```
PRINT, dtvar1
{ 1992 12 31 15 0 0.0000 87767.625 0}
```

The new Date/Time variable is 30 hours less than dtvar. Notice that for this example the year, month, day and Julian day have changed.

Finding Elapsed Time Between Two Date/Time Variables

The DT_DURATION function determines the elapsed time between two PV-WAVE Date/Time variables. The return units are a double-precision value or array of values expressed in days and fractions of days. The function has the form:

result = DT_DURATION(*dt_var_1*, *dt_var_2*)

Example

Assume two PV-WAVE Date/Time variables, `dtarray` and `dtarray1`, have been created. The contents of `dtarray` are:

```
{ 1992 3 17 6 35 23.0000 87478.275 0}
{ 1993 4 18 7 38 47.0000 87875.319 0}
```

The contents of `dtarray1` are:

```
{ 1989 5 22 9 32 22.0000 86448.397 0}
{ 1995 7 26 10 33 27.0000 88704.440 0}
```

You can use the DT_DURATION function to find the number of days between corresponding elements of the arrays.

```
dtdiff = DT_DURATION(dtarray, dtarray1)
PRINT, dtdiff
 1029.8771 -829.12130
```

Note that the function returns a negative number for the second value since the second element in `dtarray1` is more recent than the second element in `dtarray`.

Excluding Days From PV-WAVE Date/Time Variables

You can exclude holidays and weekend days from PV-WAVE Date/Time plots using the following functions.

CREATE_HOLIDAYS Procedure

If you wish to skip particular days such as holidays in your plots, first you must define them. The form of the procedure is:

CREATE_HOLIDAYS, *dt_list*

Example

Assume that you want to exclude Christmas and New Years from a PV-WAVE Date/Time variable.

```
dates = ['1-1-92', '12-25-92']
```

Create a variable that contains the dates of holidays you wish to exclude.

```
holidays = STR_TO_DT(dates, Date_Fmt=1)
```

Create a variable that contains the PV-WAVE Date/Time structures for Christmas and New Year.

```
CREATE_HOLIDAYS, holidays
```

Use the CREATE_HOLIDAYS procedure to create and store the holidays in the system variable !Holiday_List.

```
PRINT, !Holiday_List
{ 1992 12 25 0 0 0.00000 87761.000 0}
{ 1992 1 1 0 0 0.00000 87402.000 0}
                  .
                  .
{ 0 0 0 0 0 0 0 0 0 0}
```

Note You can create and store up to 50 holidays. To exclude the holidays from PV-WAVE Date/Time variables, you use the keyword *Compress* or the system variable !PDT.Compress. The system variable !PDT.Exclude_Holiday must also be set to a value of 1 (the default value).

LOAD_HOLIDAYS Procedure

This procedure is called by the CREATE_HOLIDAYS procedure. It passes the value of the !Holiday_List system variable to the PV-WAVE conversion functions. You need to run this procedure after restoring any PV-WAVE session in which you used the CREATE_HOLIDAYS function or if you directly changed the value of the !Holiday_List system variable.

CREATE_WEEKENDS Procedure

This function allows you to define certain days of the week to skip when performing Date/Time operations. CREATE_WEEKENDS defines weekend days and makes this definition available to the conversion functions and procedures. The syntax of the procedure is:

CREATE_WEEKENDS, *day_names*

Note Do not set all seven days in the week to be weekend days. This will generate an error message.

Example

```
CREATE_WEEKENDS, ['Saturday', 'Sunday']
```

Makes Saturday and Sunday the weekend days.

```
PRINT, !Weekend_List

1 0 0 0 0 0 1
```

The system variable !Weekend_List is an array of integers where one = weekend and zero = weekday.

LOAD_WEEKENDS Procedure

This procedure is called by the CREATE_WEEKENDS procedure. It passes the value of the !Weekend_List system variable to the conversion functions. You only need to run this procedure after restoring any PV-WAVE session in which you used the CREATE_WEEKENDS function or if you directly changed the value of the !Weekend_List system variable.

Note Do not set all seven days in the week to be weekend days. This will generate an error message.

Example

```
PRINT, !Weekend_List
   0     0     0     0     0     0     1
```

Current contents of !Weekend_List system variable.

```
!Weekend_List = [1, 0, 0, 0, 0, 0, 1]
```

Add Sunday to the weekend list.

```
LOAD_WEEKENDS
```

Run LOAD_WEEKENDS so the new weekend value will take effect.

DT_COMPRESS Function

This function compresses an array of PV-WAVE Date/Time values. The function returns an array of floating point values containing the compressed Julian days, that is, all holidays and weekends are removed from the array.

Note This function is only used for specialized plotting applications, such as bar charts. In most cases, you do not need to use this function. Instead use the *Compress* keyword to remove holidays and weekends from the results of Date/Time functions and plots. For detailed information, see the description of the DT_COMPRESS function in the *PV-WAVE Reference.*

Creating Plots With PV-WAVE Date/Time Data

The plotting procedures, PLOT and OPLOT, in conjunction with keywords can be used to plot multiple Date/Time labels and tick levels on the X axis. The keywords for the PLOT procedure for Date/Time include:

- *XType*
- *Start_Level*
- *Month_Abbr*
- *Box*
- *DT_Range*
- *Max_Levels*
- *Compress*

The keywords for OPLOT are *XType* and *Compress*. You can find a complete description of all these keywords in Chapter 3 *Graphics and Plotting Keywords*, in the *PV-WAVE Reference*.

The following examples show eight different types of plots with PV-WAVE Date/Time axes.

Tip Also see the *PV-*WAVE *Command Language Samples and Applications Guide* for more examples of creating Date/Time plots.

Example 1: Plotting Seconds

This example illustrates how to generate a Date/Time plot for a data file named `datafile.dat` that does not contain explicit date and time information, that is, no time stamp information. The file contains data for every second of the day for April 1, 1992. The one-column file looks like:

```
00.355187
91.9201
00.22395
63.9256
97.4526
      .
      .
      .
```

The following code generates a plot that shows the first seven seconds of data. The Date/Time axis is shown with the maximum of six labels.

```
fday = VAR_TO_DT(1992, 4, 1, 1, 1, 1)
```

Generates an initial PV-WAVE Date/Time variable to use with the DTGEN function.

```
num =7
```

Creates a variable for generating seven seconds of data.

```
x = DTGEN(fday, num, /Second)
```

Generates a PV-WAVE Date/Time variable with Date/Time structures for the first seven seconds of April.

```
status = DC_READ_FREE("datafile.dat", y, /Col)
```

Reads the data from the file datafile.dat and assigns it to the variable y. The values for all of the seconds, 86400, are actually read into y. However, only the first seven seconds are plotted for this example.

```
PLOT, x, y, Psym=-4
```

Plots the first seven seconds of data.

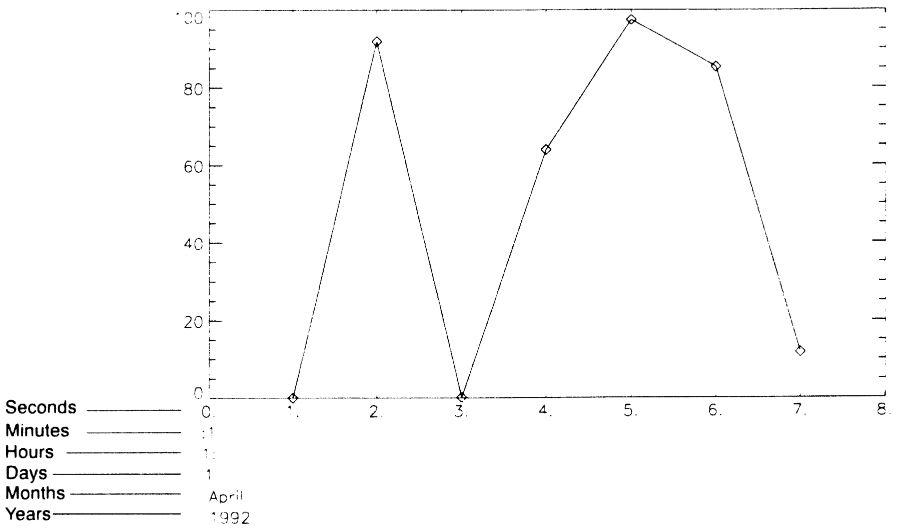


Figure 7-2 A Date/Time plot showing the first seven seconds of data for April 1, 1992. The keyword Psym value of -4 connects the data points with solid lines.

Example 2: Plotting Minutes

The second example uses the same data file as Example 1 (`datafile.dat`). The example shows how you can plot a graph for the data at each minute rather than each second. The *Box* keyword draws boxes around the tick marks and labels of the Date/Time axis.

```
fday = VAR_TO_DT(1992, 4, 1, 1, 1, 1)
```

Generates an initial PV-WAVE Date/Time variable to use with the DTGEN function.

```
num = 20
```

Creates a variable to use with the DTGEN function for generating an array of PV-WAVE Date/Time structures.

```
x = DTGEN(fday, num, /Minute)
```

Generates a PV-WAVE Date/Time variable with Date/Time structures for 20 minutes in April.

```
status = DC_READ_FREE("datafile.dat", y, /Col)
```

Reads the data from the file datafile.dat into the variable y.

```
PLOT, x, y, Psym=-4, /Box
```

Plots the first 20 minutes of data with boxes around the Date/Time axis.

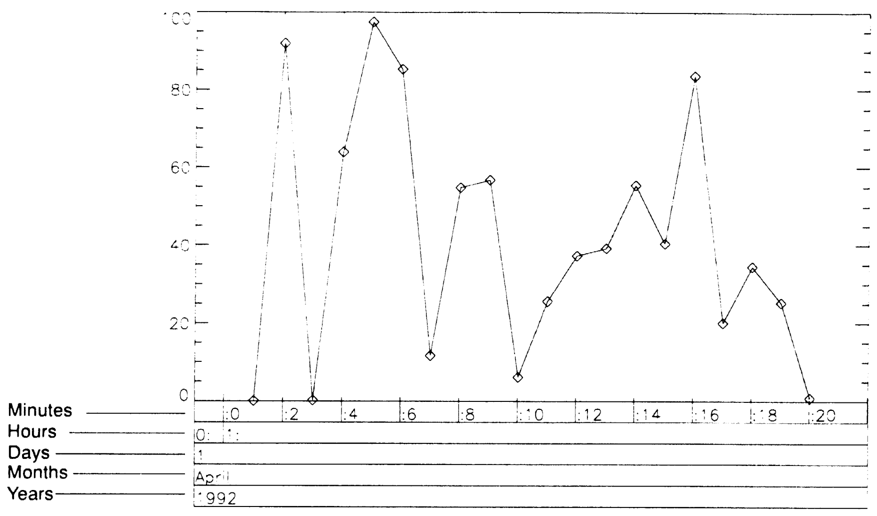


Figure 7-3 A Date/Time plot for the first twenty minutes of April 1. The Box keyword draws the boxes around the Date/Time labels.

If no boxes are drawn for the Date/Time axis, labels are centered with respect to the tick marks for seconds, minutes, hours, and days. Weeks, months, quarters, and years are always left-justified. See Example 1. With boxes, the labels are left-justified in relation to the tick marks.

Example 3: Plotting Hourly Data

The third example uses the same data file as Examples 1 and 2. This example plots data for every hour of the day April 1.

```
fday=VAR_TO_DT(1992, 4, 1, 1, 1, 1)
```

Creates an initial PV-WAVE Date/Time variable to use with the DTGEN function.

```
num =24
```

Creates a variable used with the DTGEN function to create an array of Date/Time structures.

```
x = DTGEN(fday, num, /Hour)
```

Creates 24 Date/Time structures for the hours of the day.

```
status = DC_READ_FREE("datafile.dat", y, /Col)
```

Reads the data into the variable y.

```
Plot, x, y, Psym=-4, /Box
```

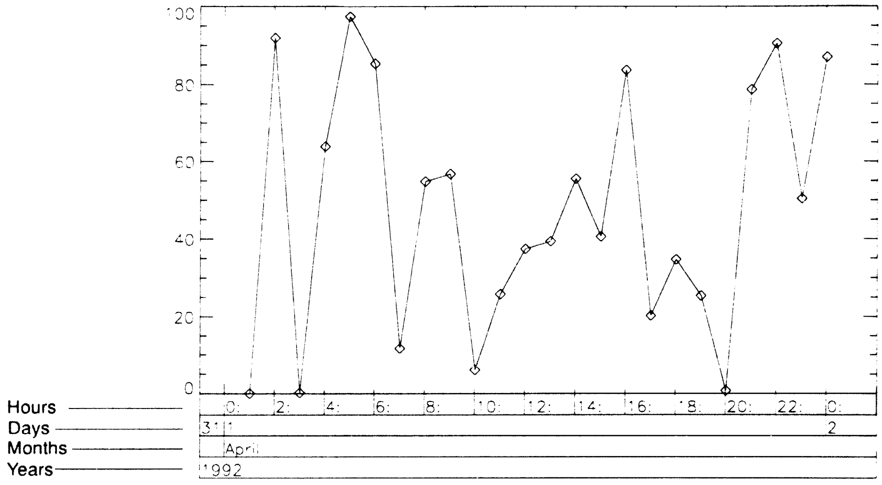


Figure 7-4 A Date/Time example with the data at each hour for April 1 plotted.

Example 4: Plotting Daily Sales Data

Examples 4 through 8 plot Date/Time data for a file named `sales1.dat` that contains Date/Time stamps for product sales. The file has ten columns. Each data set column has an accompanying date stamp column:

Product Sales

Daily	Weekly	Monthly	Quarterly	Yearly
00 1/01/1991	159 1/06/91	1088 1/31/91	3000 910101	5280 85941
91 1/02/1991	152 1/13/91	1085 2/28/91	1942 910401	6581 86307
05 1/03/1991	202 1/20/91	0827 3/31/91	.	.
.	.	.	2345 911001	7621 87037
.	.	1147 12/31/31		
.	150 12/31/91			
57 12/31/1991				

Note This data file is an example file only. It is used to generate plots for various levels of PV-WAVE Date/Time data.

Example 4 plots the daily sales for the month of January. Weekend days are compressed with the keyword *Compress*. The *DT_Range* keyword is used to plot a portion of the Date/Time data read in from the file.

```
dates = REPLICATE({!DT}, 60)
```

Create PV-WAVE Date/Time structures to hold Date/Time data for the days in January and February.

```
CREATE_WEEKENDS, ["sun", "sat"]
```

This procedure defines the weekend days.

```
status = DC_READ_FREE("sales1.dat", $
   amount, dates, /Col, Dt_Template=[1], $
   Delim=[" "], NSkip=2, $
   Get_Columns=[1,2])
```

Reads the data from Column 1 into the variable amount. Reads the data from Column 2 into the variable dates. The date/times from Column 2 are converted to PV-WAVE Date/Time data. The NSkip keyword skips over the first two header lines in the file.

```
sdate = VAR_TO_DT(1991,1,1)
edate = VAR_TO_DT(1991,1,30)
```

Creates variables to be used with the DT_Range keyword. These variables establish a range for plotting each day of the month in January.

```
PLOT, dates, amount, /Compress, $
   Start_Level=3, $
   DT_Range=[sdate.julian,edate.julian]
```

Plots the PV-WAVE Date/Time data on the X-axis and the daily sales for the month of January on the Y-axis. Weekends are compressed. Setting the Start_Level keyword to 3 forces the plot to use days as the first axis level. The DT_Range keyword defines the range of Date/Time data that will be plotted. In this example only the days of January are plotted.

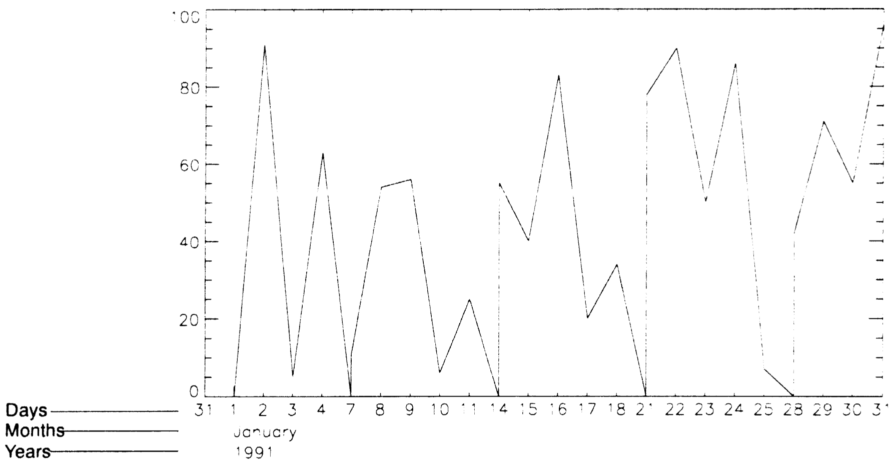


Figure 7-5 A Date/Time plot illustrating daily product sales for January. The DT_Range keyword restricts the days to January only. The Compress keyword eliminates weekend days (January 5, 6, 12, 13, 19, 20, 26, and 27).

Example 5: Plotting Sales Per Week

Example 5 plots the weekly sales from January 1 to May 6. The data and Date/Time are read in from Columns 3 and 4 of `sales1.dat`.

```
dates = REPLICATE({!DT}, 18)
```

Create PV-WAVE Date/Time structure to read Date/Time data into.

```
status = DC_READ_FREE("sales1.dat", amount, $
   dates, /Col, Dt_Template=[1], $
   Delim=[" "], Get_Columns=[3,4], NSkip=2)
```

Read sales data into the amount variable. Read and convert Date/Time data into the dates variable.

```
PLOT, dates, amount, Start_Level=4, PSym=-4
```

The keyword Start_Level selects weeks for plotting.

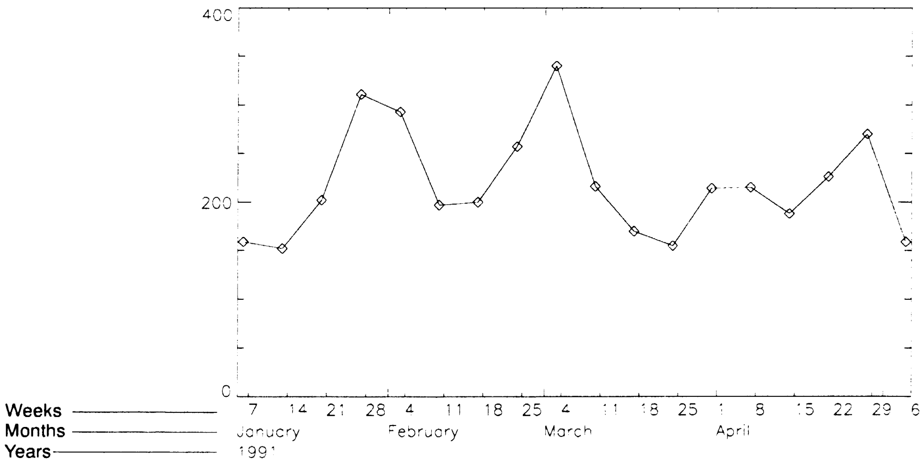


Figure 7-6 Date/Time plot of product sales for each week from January 7 to May 6. The Start_Level keyword value of 4 ensures that the weekly amounts are plotted on the first level.

Example 6: Plotting Monthly Sales

Example 6 plots the total sales for each month. This data is contained in Columns 5 and 6 of `sales1.dat`. The keyword *Month_Abbr* automatically abbreviates some month names to three characters depending on the available space on the axis. In this example, no labels would be shown for the months of February or September without this keyword.

```
dates = REPLICATE({!DT}, 12)
```

Creates PV-WAVE Date/Time structures for the months of the year.

```
status = DC_READ_FREE("sales1.dat", $
   amount, dates, /Col, Dt_Template=[1], $
   Delim=[" "], NSkip=2, Get_Columns=[5,6])
```

Reads monthly sales data into the amount variable. Reads date/time data into the variable dates as PV-WAVE Date/Time data.

```
PLOT, dates, amount, /Month_Abbr
```

Plots data with several months abbreviated to fit labels for all 12 months on the Date/Time axis.

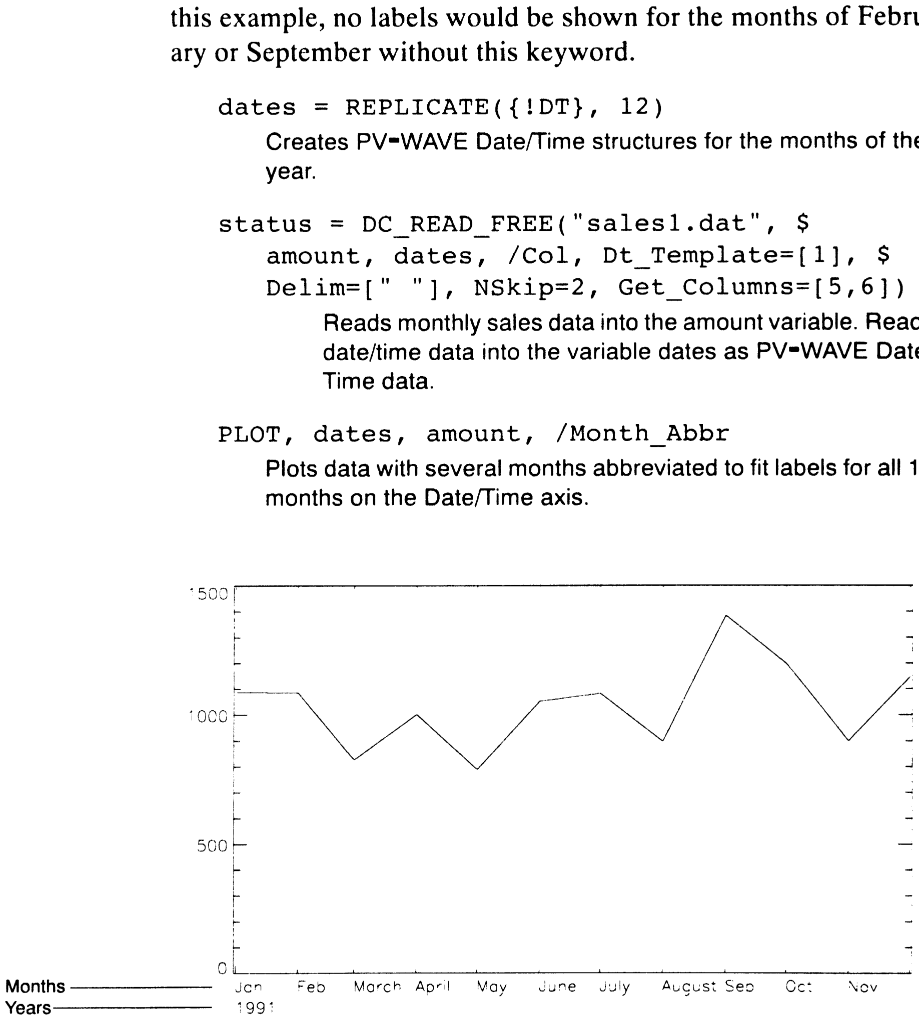


Figure 7-7 The monthly sales for 1991. The Month_Abbr keyword allows all month names to be written on the Date/Time axis. Without this keyword, no labels would be shown for February or September.

Example 7: Plotting Quarterly Sales

This example plots the data from Columns 7 and 8 of the sample file `sales1.dat`. The date information is not in any format that can be used with the *DT_Template* keyword. Therefore, the dates are first read and then converted using the VAR_TO_DT function.

The *Start_Level* keyword insures that the quarterly sales are printed out on the first level. The *Max_Levels* keyword defines one level of axis labels. An OPLOT is also shown for the projected sales for each quarter of the year (the individual diamonds in Figure 7-8). The OPLOT does not generate any tick marks or labels; you can only plot a second set of data on the original Date/Time axis.

```
year = intarr(4) & month = intarr(4)
day = intarr(4)
```

Creates variables to hold the date information for the four quarters of the year.

```
status = DC_READ_FIXED("sales1.dat", $
   amount, year, month, day, /Col, NSkip=2, $
   FORMAT = ('39X, I4, 1X, 3I2)')
```

Reads the sales data into the amount variable. Reads the date information into the variables, year, month, and day.

```
dates = VAR_TO_DT(year, month, day)
```

Converts the date information to a PV-WAVE Date/Time variable.

```
PLOT, dates, amount, Start_Level=6, $
   /Max_Levels
```

Plots the data with only one level of axis labeling. Without the Max_Levels keyword assignment, the labels for years would also be printed out.

```
amount1 = [3507, 2310, 2917, 1807]
```

These are the projected sales for each quarter of 1991.

```
OPLOT, dates, amount1, PSym=4
```

Plots the projected sales of each quarter as individual diamonds.

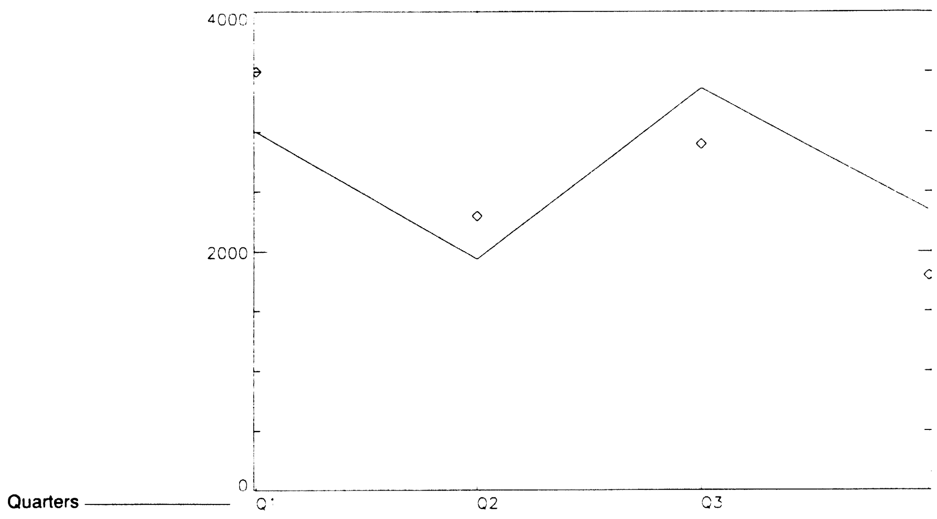


Figure 7-8 The quarterly sales for 1991. The plot also shows the projected sales (individual diamonds) for each quarter plotted with the OPLOT procedure.

Example 8: Plotting Yearly Sales

Example 8 plots the data from Columns 9 and 10 of `sales1.dat`. The Julian dates are supplied in Column 10. Column 9 contains the sales for each of the last four years.

```
d = lonarr(4)
```

Defines the d variable as a long array containing 4 elements. This variable will contain the Julian dates read in from Column 10.

```
status = DC_READ_FREE("sales1.dat", $
   amount, d, /Col, Delim=[" "], NSkip=2, $
   Get_Columns=[9,10])
```

Reads the sales data into the amount variable. Reads the date information into variable d.

```
dates = JUL_TO_DT(d)
```

Converts the date information in variable d to a PV-WAVE Date/Time variable.

```
PLOT, dates, amount, Start_Level=7
```

Plots yearly sales. The Start_Level keyword ensures that the years are labeled on the first level of the X axis. If this keyword were omitted, quarters would be the first level of labeling.

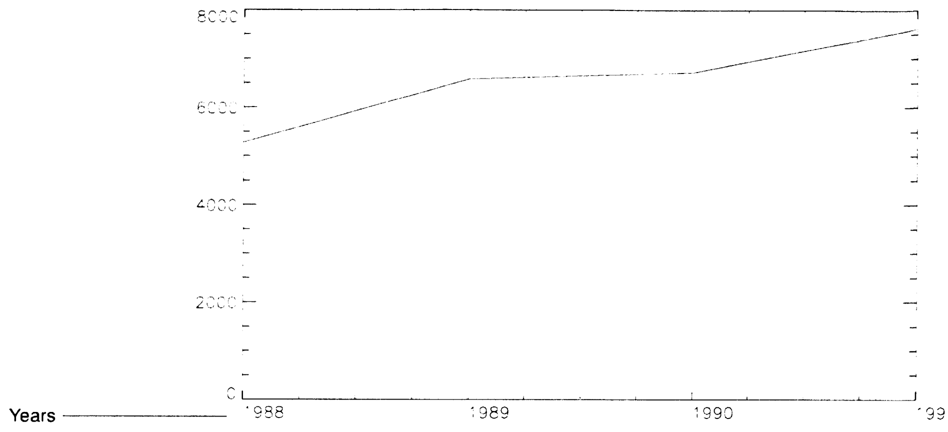


Figure 7-9 The yearly sales for the last four years. This plot can be generated using the JUL_TO_DT function (Example 8) or the XType keyword (Example 9).

Example 9: Plotting Yearly Sales with the XType Keyword

PV-WAVE provides another method for plotting Date/Time data if your file contains Julian days as in Example 8. You can set the *XType* plot keyword to a value of 2 to generate a PV-WAVE Date/Time axis. Since the Julian days are provided in Column 10 for the yearly sales data in Column 9, you can plot these two columns as follows:

```
dates = LONARR(4)
```

Defines the dates variable as a long array containing 4 elements.

```
status = DC_READ_FREE("sales1.dat", $
   amount, dates, /Col, $
   Delim=[" "], Get_Columns=[9,10], NSkip=2)
```
Reads the sales data into the amount variable. Reads the date information into the dates variable.

```
PLOT, dates, amount, XType=2, Start_Level=7
```
Plots the data as shown in Figure 7-9.

See Chapter 3 *Graphics and Plotting Keywords*, in the *PV-WAVE Reference* for more information about the *XType* keyword. Also see the description of the DT_COMPRESS function in the *PV-WAVE Reference.*

Writing PV-WAVE Date/Time Data To a File

There are two methods that you can use to write PV-WAVE Date/ Time data to a file. You can use DC_WRITE functions to both convert and write data or you can first convert the PV-WAVE Date/ Time data and then write it to a file.

Using DC_WRITE Functions

You can use DC_WRITE functions to convert data from the PV-WAVE Date/Time format to another format and then write the new Date/Time data to a file. The DC_WRITE functions are easy to use because they automatically handle many aspects of data transfer, such as opening and closing the data file.

The two DC_WRITE functions that you can use to convert and write data are DC_WRITE_FIXED and DC_WRITE_FREE. For examples and a detailed discussion of these two functions, refer to their descriptions in the *PV-WAVE Reference.*

Tip By default, DC_WRITE_FREE generates CSV (Comma Separated Value) ASCII data files.

Using Conversion Routines

You can also use three conversion routines in conjunction with the WRITE and WRITEU procedures to convert PV-WAVE Date/Time data for output to a file. The conversion routines are:

- DT_TO_SEC
- DT_TO_STR
- DT_TO_VAR

DT_TO_STR Procedure

This procedure converts PV-WAVE Date/Time variables to strings. The procedure has the form:

DT_TO_STR, *dt_var*

Example

Assume you have a PV-WAVE Date/Time variable named `date1` that contains the following Date/Time structures:

```
{ 1992 3 13 1 10 34.0000 87474.049 0}
{ 1983 4 20 16 18 30.0000 84224.680 0}
{ 1964 4 24 5 7 25.0000 77289.213 0}
```

To convert to data, use the DT_TO_STR procedure:

```
DT_TO_STR, date1, d, t, Date_Fmt=1, $
   Time_Fmt=-1
```

Converts PV-WAVE Date/Time data. Stores the date data in d and the time data in t. The Date_Fmt and Time_Fmt keywords define the formats that date1 is using. DT_TO_STR uses the same formats as STR_TO_DT. See *The STR_TO_DT Function* on page 229 for an explanation of valid formats.

```
PRINT, d
03/13/1992 04/20/1983 04/24/1964
PRINT, t
01:10:34 16:18:30 05:07:25
```

DT_TO_VAR Procedure

This procedure converts PV-WAVE Date/Time variables into variables that contain numerical Date/Time information. The procedure has the form:

DT_TO_VAR, *dt_value*

Example

Assume that you have created a PV-WAVE Date/Time variable named `date1` that contains the following Date/Time data:

{ 1992 3 13 10 34 15.0000 87474.440 0}

{ 1983 4 20 12 30 19.0000 84224.521 0}

{ 1964 6 24 16 25 14.0000 77350.684 0}

To convert the data in this PV-WAVE Date/Time variable:

```
DT_TO_VAR, date1, Year=years, $
   Month=months, Day=days
```

This procedure creates several variables containing the Date/Time data.

```
PRINT, "Years =", years
Years = 1992 1983 1964
```

The keyword Year generates an integer array that contains the years.

```
PRINT, "Months =", months
Months = 3 4 6
```

The keyword Month creates a byte array with the months.

```
PRINT, "Days =", days
Days = 13 20 24
```

The keyword Day creates a byte array with the days of the month.

DT_TO_SEC Function

This function converts PV-WAVE Date/Time data into seconds. The function has the form:

result = DT_TO_SEC(*dt_value*)

Example

Assume that you have created the array *dtarray* that contains the following PV-WAVE Date/Time data:

```
{ 1992 4 15 7 29 19.0000 87507.312 0}
{ 1993 4 15 7 29 19.0000 87872.312 0}
{ 1994 4 15 7 29 19.0000 88237.312 0}
```

To find out the number of seconds for each Date/Time from January 1, 1970, use the DT_TO_SEC function:

```
seconds = DT_TO_SEC(dtarray, $
   Base='1-1-70', Date_Fmt=1)
PRINT, seconds
 7.0332296e+08 7.3485896e+08 7.6639496e+08
```

Miscellaneous PV-WAVE Date/Time Utility Functions

PV-WAVE contains several utilities for generating and obtaining information about PV-WAVE Date/Time variables. For more information about each of these functions, see the *PV-WAVE Reference*. These include:

- TODAY
- DAY_NAME
- DAY_OF_WEEK
- MONTH_NAME
- DAY_OF_YEAR
- DT_PRINT

The TODAY Function

This function returns a PV-WAVE Date/Time variable containing the current date and time. The form of the function is:

result = TODAY

Example

```
dttoday = TODAY( )
PRINT, dttoday
{ 1992 3 26 7 11 14.0000 87487.299 0}
```

The DAY_NAME Function

This function returns a PV-WAVE string variable or array of string variables containing the name(s) of the day(s) of the week of the date(s) in the input variable. The form of the function is:

result = DAY_NAME(*dt_var*)

Example

Assume that you have a PV-WAVE Date/Time variable, `date`, for April 13, 1992. To find out which day of the week this date is, enter:

```
day = DAY_NAME(date)
PRINT, day
Monday
```

The day names are accessed from the !Day_Names system variable.

The DAY_OF_WEEK Function

This function returns the day(s) of the week expressed as an integer(s) for a PV-WAVE Date/Time variable. Day 0 is Sunday, 1 is Monday, etc. The syntax of the function is:

result = DAY_OF_WEEK(*dt_var*)

Example

Assume that you have a PV-WAVE Date/Time variable, `date`, for April 13, 1992. To find out which day of the week this date is, enter:

```
day = DAY_OF_WEEK(date)
PRINT, day
    1
```

It is a Monday.

The MONTH_NAME Function

This function returns a PV-WAVE string or array of strings containing the month name of *dt_var*, where *dt_var* is a PV-WAVE Date/Time variable. The function has the form:

result = MONTH_NAME(*dt_var*)

Example

```
dttoday = TODAY()
{ 1992 4 1 6 12 57.0000 87493.259 0}
```

Create a variable that contains PV-WAVE Date/Time data for today's date.

```
m = MONTH_NAME(dttoday)
PRINT, m
April
```

The month is April.

The month names are accessed from the system variable !Month_Names.

The DAY_OF_YEAR Function

This function returns a PV-WAVE integer or array of integers representing the day number of the year for each Date/Time value. Day numbers fall in a range between 1 and 365 (or 366 for a leap year). The syntax of the function is:

result = DAY_OF_YEAR(*dt_var*)

Example

```
dttoday = TODAY()
```

Create a PV-WAVE Date/Time variable.

```
daynumber = DAY_OF_YEAR(dttoday)
PRINT, daynumber
   106
```

The DT_PRINT Procedure

This procedure takes the values in a PV-WAVE Date/Time variable and prints these values in a readable manner. The procedure has the form:

DT_PRINT, *dt_var*

```
dttoday= TODAY()
DT_PRINT, dttoday
4/2/1992 7:7:51.0000
```

8

CHAPTER

Creating and Querying Tables

A table is a natural and easily understood way of organizing data into columns and rows. Many computer systems use the table model to organize large amounts of data. For example, the relational database stores all of its data in a tabular format.

The PV-WAVE table functions let you create tables and subset them in various ways. These functions are both powerful and easy to use. Tables, which you create with the BUILD_TABLE function, can be subsetted and manipulated with the QUERY_TABLE function. QUERY_TABLE, which closely resembles the Structured Query Language (SQL) SELECT command, is an easy to learn and conceptually natural way to access data in tables.

What are the Table Functions?

The PV-WAVE table functions include:

- **BUILD_TABLE** – Creates a new table from PV-WAVE numeric or string vectors (one-dimensional arrays) of equal length.
- **QUERY_TABLE** – Lets you subset, rearrange, group, and sort table data. This function returns a new table containing the query results.

- **UNIQUE** – Removes duplicate elements from any vector (one-dimensional array).

Table Functions and Structured Query Language (SQL)

The syntax of the QUERY_TABLE function closely resembles the Structured Query Language (SQL) SELECT command. SQL is a widely-used language that allows users to access the information in relational databases (databases that are organized as tables). Many SQL statements resemble English sentences, and thus SQL syntax is generally easy to learn and understand. Anybody familiar with SQL will find the QUERY_TABLE function easy to understand and use.

A Quick Overview of the Table Functions

This quick overview is intended to give you a feel for the capabilities of the table functions. Greater detail on all aspects of these functions is provided throughout the rest of this chapter.

Assume that a company-wide telephone system automatically collects data on various aspects of a company's telephone calls. The system collects the date and time of each call, the caller's initials, caller's extension number, area code of call, phone number of call, call duration, and cost. This information is collected and stored in a data file.

After you read this data into PV-WAVE, you can use the BUILD_-TABLE function to create a table. Once the table is created, you can use QUERY_TABLE to subset the data in various ways.

Here are some typical table queries using the QUERY_TABLE function. Assume that the name of the table (which is specified when the table is created) is `phone_data`. The names of the table's columns are just as they appear in the following table. Don't worry now about the details of how the functions work, similar queries are explained in detail later in this chapter.

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901002	093200	21.40	TAC	311	5.78	215	2155554242
901002	094700	1.05	BWD	358	0.0	303	5553869
901002	094700	17.44	EBH	320	4.71	214	2145559893
901002	094800	16.23	TDW	289	0.0	303	5555836
901002	094800	1.31	RLD	248	.35	617	6175551999
901003	091500	2.53	DLH	332	.68	614	6145555553
901003	091600	2.33	JAT	000	0.0	303	555344
901003	091600	.35	CCW	418	.27	303	5555190
901003	091600	1.53	SRB	379	.41	212	2125556618
901004	094700	.80	JAT	000	0.0	303	555320
901004	094900	1.93	SRB	379	.52	818	8185552880
901004	095000	3.77	DJC	331	1.02	512	5125551228

Create a subset of the table that only shows the date, duration, and extension of calls made.

```
tbl = QUERY_TABLE(phone_data, $
   'DATE, DUR, EXT')
```

Show me all of the calls made on October 2, 1990.

```
tbl = QUERY_TABLE(phone_data, $
   '* Where DATE = 901002')
```

Sort the table in descending order, by cost.

```
tbl = QUERY_TABLE(phone_data, $
   '* Order By COST Desc')
```

Sort the table first in ascending order by date, then within each group of dates by cost in descending order.

```
tbl = QUERY_TABLE(phone_data, $
   '* Order By DATE, COST Desc')
```

Show me the total cost incurred from each telephone extension on October 3.

```
tbl = QUERY_TABLE(phone_data, $
   'EXT, Sum(COST) Where DATE = 901003,' + $
   'Group By EXT')
```

Note Note that the second parameter in a QUERY_TABLE call is one string. The plus sign (+) used above is the PV-WAVE string concatenation operator. It is used because it is not legal otherwise to break a string onto multiple lines within a PV-WAVE command.

For each extension, what was the average cost of out-of-state calls from October 3 to October 6?

```
tbl = QUERY_TABLE(phone_data, $
   'EXT, Avg(COST) ' + $
   'Where (DATE >= 901003 AND DATE <= ' + $
   '901006) AND (AREA <> 303), Group By ' + $
   'EXT')
```

Show me the data on all of the calls that cost less than $5.00.

```
tbl = QUERY_TABLE(phone_data, $
   '* Where COST < 5.0')
```

Show me the calls made by the caller with initials TAC.

```
tbl = QUERY_TABLE(phone_data, $
   '* Where INIT = "TAC"')
```

Show me the extension, date, and total duration of all calls made from each extension on each date.

```
tbl = QUERY_TABLE(phone_data, $
   'EXT, DATE, Sum(DUR) Group By EXT, DATE')
```

Creating a Table

To use the QUERY_TABLE function, you have to create a table first with the BUILD_TABLE command. Tables are created from vectors (one-dimensional variables) that contain the same number of elements. Each variable becomes, in effect, a column in the table. Before you attempt to create a table, however, you need to read your data into a set of variables.

For detailed information on reading data into PV-WAVE, see Chapter 8, *Working with Data Files*, in the *PV-WAVE Programmer's Guide.*

For information on creating a table that contains Date/Time data, see *Using Date/Time Data in Tables* on page 281.

Once your data is read into a set of equal-sized variables, use the BUILD_TABLE function to build a table. Each variable becomes, in effect, a separate column in the table. Once the variables are placed into a table, QUERY_TABLE can be used to subset and manipulate the data.

Tip In PV-WAVE, a table is represented as an array of structures. You do not have to understand or use structures to use the table functions. However, you may want to review the chapter on structures, Chapter 6, *Working with Structures*, in the *PV-WAVE Programmer's Guide*, before you proceed to learn about the table functions. Also see the section *Tables and Structures* on page 287.

Note The table columns and the original input variables are separate. The original variables are not removed when the table is created.

Example 1: Building a Table

The following example assumes that you have defined eight variables and read data into them. The data for this example represents information collected from a company-wide telephone system. The variable names are: DATE, TIME, DUR, INIT, EXT, COST, AREA, and NUMBER.

The following command builds an eight-column table from the telephone data variables. Note that BUILD_TABLE takes one parameter, a string containing the names of the variables.

```
phone_data = BUILD_TABLE('DATE, TIME, ' +$
   'DUR, INIT, EXT, COST, AREA, NUMBER')
```

The result is a new table called phone_data, which is illustrated below:

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901002	093200	21.40	TAC	311	5.78	215	2155554242
901002	094700	1.05	BWD	358	0.0	303	5553869
901002	094700	17.44	EBH	320	4.71	214	2145559893

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901002	094800	16.23	TDW	289	0.0	303	5555836
901002	094800	1.31	RLD	248	.35	617	6175551999
901003	091500	2.53	DLH	332	.68	614	6145555553
901003	091600	2.33	JAT	000	0.0	303	555344
901003	091600	.35	CCW	418	.27	303	5555190
901003	091600	1.53	SRB	379	.41	212	2125556618
901004	094700	.80	JAT	000	0.0	303	555320
901004	094900	1.93	SRB	379	.52	818	8185552880
901004	095000	3.77	DJC	331	1.02	512	5125551228
901004	095100	.16	GWP	370	0.0	303	5551245

Tip You can format and print a table so that it appears approximately like the above example. For information on printing table data, see *Formatting and Printing Tables* on page 284.

Using INFO to View the Table Structure

You can use the INFO command to view the table structure. Tables are represented in PV-WAVE as arrays of structures (for more information on this, see *Tables and Structures* on page 287). Thus, the *Structure* keyword is used with the INFO command to obtain information on tables, for example:

```
INFO, /Structure, phone_data
** Structure TABLE_0, 8 tags, 40 length:
DATE            LONG            901002
TIME            LONG            93200
DUR             FLOAT           21.4000
INIT            STRING          'TAC'
EXT             LONG            311
COST            FLOAT           5.78000
AREA            LONG            215
NUMBER          STRING          '2155554242'
```

Only Vectors can be Used in BUILD_TABLE

A table is built from vector (one-dimensional array) variables only. You cannot include expressions in the BUILD_TABLE function. For example, the following BUILD_TABLE call is *not* allowed:

```
result = BUILD_TABLE('EXT(0:5), COST(0:5)')
```

However, you can achieve the desired results by performing the array subsetting operations first, then using the resulting variables in BUILD_TABLE. For example:

```
EXT = EXT(0:5)
COST = COST(0:5)
result = BUILD_TABLE('EXT, COST')
```

In addition, you cannot include scalars or multidimensional-array variables in BUILD_TABLE.

Example 2: Building a Different Table with the Same Data

From any given set of equal-length variables, BUILD_TABLE can use all or some of the variables to build a table, and the table's columns can be placed in any order.

The following table contains just four columns instead of eight. Also, the columns appear in a different order than in the previous example.

```
new_tbl = BUILD_TABLE('DATE,EXT,DUR,COST')
```

Here is a portion of this new table:

DATE	EXT	DUR	COST
901002	311	21.40	5.78
901002	358	1.05	0.0
901002	320	17.44	4.71
901002	289	16.23	0.0
901002	248	1.31	.35
901003	332	2.53	.68
901003	000	2.33	0.0

Example 3: Renaming Columns

By default, BUILD_TABLE uses the original variable names as the names of the table columns. You can rename columns by including the new name or "alias" directly in the BUILD_TABLE command. Place the alias immediately after the original variable name. For example, the previous new_tbl table can be created with different column names:

```
rename_tbl = BUILD_TABLE('DATE Call_Date, '+$
   'EXT Extension, DUR Call_Length,'+$
   'COST Call_Cost')
```

The resulting table is identical to the table created in the previous section, except for the column names. To see the structure of this new table, enter:

```
INFO, /structure, rename_tbl
** Structure TABLE_0, 8 tags, 40 length:
CALL_DATE       LONG         901002
EXTENSION       LONG         311
CALL_LENGTH     FLOAT        21.4000
CALL_COST       FLOAT        5.78000
```

Querying a Table

To query a table usually means to subset the data in it. The QUERY_TABLE function returns a new table containing your query results, usually a subset of the original table.

QUERY_TABLE lets you:

- Rearrange a table and rename columns.
- Remove duplicate rows from a table.
- Summarize related groups of data with functions that add, average, count, and perform other calculations.
- Sort columns of data into ascending or descending order.
- Subset a table using Boolean and relational operators to retrieve specific ranges of data.

Restoring a Sample Table

The phone_data table described in this chapter is available in a save file in the WAVE_DATA directory. To restore this file, use the following RESTORE command:

```
RESTORE, !Dir+'/data/phone_example.sav'
```

If you restore this file, you can practice using most of the commands described in this chapter.

The QUERY_TABLE Function

The complete syntax (usage) of the QUERY_TABLE function is:

result = QUERY_TABLE(*table*,
' [Distinct] * | *col*$_1$ [*alias*] [, ..., *col*$_n$ [*alias*]]
[Where *cond*]
[Group By *colg*$_1$ [,... *colg*$_n$]] |
[Order By *colo*$_1$ [*direction*][,..,*colo*$_n$ [*direction*]]] ')

Note that the second parameter is one long string and must be inclosed in quotes.

For a complete description of the function's syntax, see the *PV-WAVE Reference.*

Rearranging a Table

One of the simplest uses of QUERY_TABLE is to rearrange and/ or rename the columns of an existing table (a table already created with the BUILD_TABLE function). To create a new table from phone_data containing only the phone extensions, area code, and phone number of each call made, you could enter:

```
new_table = QUERY_TABLE(phone_data, $
   'EXT, AREA, NUMBER')
```

Here is a portion of the resulting table:

EXT	AREA	NUMBER
311	215	2155554242
358	303	5553869
320	214	2145559893
289	303	5555836
248	617	6175551999
332	614	6145555553

Tip You can print or plot data from a table. For information on printing table data, see *Formatting and Printing Tables* on page 284. For information on plotting table data, see *Plotting Table Data* on page 286.

Renaming Columns

The following command is similar to the previous one, except that aliases (`Extension` and `Area_Code`) are used to rename two of the columns:

```
new_table = QUERY_TABLE(phone_data, $
   'EXT Extension, AREA Area_Code, NUMBER')
```

You can see that these new names are in effect with the INFO command:

```
INFO, /Structure, new_table
** Structure TABLE_QT_2, 3 tags, 16 length:
 EXTENSION      LONG          311
 AREA_CODE      LONG          215
 NUMBER         STRING        '2155554242'
```

Using the Distinct Qualifier

The Distinct qualifier removes duplicate rows from the columns specified in the QUERY_TABLE command. For example, the following command returns the unique dates appearing in the table:

```
dates = QUERY_TABLE(phone_data, $
   'Distinct DATE')
```

The result is a one-column table containing the unique dates on which data were gathered. All duplicate dates have been filtered out of the result.

```
PRINT, dates
   {901002}   {901003}   {901004}
```

Tip The same basic result can be accomplished with the UNIQUE function, described in the *PV-WAVE Reference*. UNIQUE returns the unique elements of any one-dimensional array. When used to find unique elements of a table column, PV-WAVE data-structure notation must be used to specify the column (for more information, see *Tables and Structures* on page 287). For example:

```
dates = UNIQUE(phone_data.DATE)
```

Summarizing Data with Group By

The Group By clause sorts the table into rows grouped by common values in specified columns. Used with calculation functions, Group By lets you produce summaries of data associated with each grouping. For example, you can find the total cost of all calls made from each extension:

```
new_tbl = QUERY_TABLE(phone_data, $
   'EXT, Sum(COST) Group By EXT')
```

Or, you can find the number of calls made on each date:

```
new_tbl = QUERY_TABLE(phone_data, $
   'DATE, Count(NUMBER) Group By DATE')
```

Or, you can obtain the total duration from each extension on each date (a multiple grouping):

```
tbl = QUERY_TABLE(phone_data, $
   'EXT, DATE, Sum(DUR) Group By EXT, DATE')
```

Calculation Functions Used with Group By

Group By is always used in conjunction with one or more calculation functions, such as Sum and Count. These functions operate on the lowest-level grouping to produce the desired result:

Table 8-1: Calculation Functions

Function	Description	Phone_Data Applications
Sum( )	Returns the total of the values in the group.	total duration: Sum(DUR) total cost: Sum(COST)
Count( )	Returns the number of items in the group.	how many calls made: Count(NUMBER)
Min( )	Returns the smallest element in the group.	first date: Min(DATE)
Max( )	Returns the largest element in the group.	last day: Max(DATE)
Avg( )	Returns the average of the values in the group.	average cost: Avg(COST) average duration: Avg(DUR)

Tip These functions are described further in the description the QUERY_TABLE function in the *PV-WAVE Reference.*

Using More than One Calculation Function

More than one calculation function can be placed in a single QUERY_TABLE command. For example, you can create a table showing the total cost *and* total duration of calls made from each phone extension for the period of time the data were collected.

```
cost_sum = QUERY_TABLE(phone_data, $
   'EXT, Sum(COST), Sum(DUR) Group By EXT')
```

This produces the new table, called `cost_sum` containing the columns EXT, SUM_COST, and SUM_DUR. The cost and duration columns are renamed, by default, with the prefix SUM_. This prevents confusion with the existing table columns that are already named COST and DUR.

A portion of the resulting table is shown below. The values in the SUM_COST and SUM_DUR columns represent the *total cost* and *total duration* of calls made from each extension.

EXT	SUM_COST	SUM_DUR
0	0.00000	4.49000
248	0.350000	1.31000
289	0.00000	16.2300
311	5.78000	21.4000
320	4.71000	17.4400
331	1.02000	3.77000

The INFO command shows the basic structure of this new table:

```
INFO, /structure, cost_sum
** Structure TABLE_GB_2, 3 tags, 12 length:
 EXT LONG 0
 SUM_COST FLOAT 0.370000
 SUM_DUR FLOAT 592.140
```

Tip You could rename the columns in the previous command by adding an alias after the column names. For example, Total_Cost and Total_Time are aliases in the following function:

```
cost_sum = QUERY_TABLE(phone_data, $
   'EXT, Sum(COST) TOTAL_COST, Sum(DUR) '+$
   'TOTAL_TIME Group By EXT')
```

Multiple Groupings

Finally, you can specify more than one column in the Group By clause. For example, you can obtain a grouping by extension *and* by date. The result is a “group within a group”.

The following command produces such a table:

```
tbl = QUERY_TABLE(phone_data, $
   'EXT, DATE, Sum(DUR) Group By EXT, DATE')
```

For more information on producing multiple groupings, see the description of QUERY_TABLE in the *PV-WAVE Reference.*

Sorting Data with Order By

The Order By clause is used to sort a table. Order By sorts columns into ascending or descending order.

Suppose you want to rearrange the phone data table so that it is sorted by extension, in ascending order (ascending order is the default). You can do this with the following command:

```
ext_sort = QUERY_TABLE(phone_data, $
   '* Order By EXT')
```

Note The asterisk (*) before Order By is a wildcard character that pulls all the columns in `phone_data` into the resulting table.

A portion of the resulting table is shown below. Note that the `EXT` column is sorted in ascending order.

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901004	95300	1.36	JAT	0	0.00	303	480320
901004	94700	0.80	JAT	0	0.00	303	480320
901002	91600	2.33	JAT	0	0.00	303	480344
901002	94800	1.31	RLD	248	0.35	617	6174941999
901002	94800	16.2	TDW	289	0.00	303	2955836

Sorting in Descending Order

Use the Desc qualifier to sort a column in descending order. For example, the previous table can be further refined by sorting the `COST` field in descending order:

```
cost_sort = QUERY_TABLE(phone_data, $
   'EXT, COST, DATE Order By EXT, COST Desc')
```

This command produces a subsetted table with the `COST` column sorted in *descending* order (as specified with the Desc qualifier) within each group of extensions. The following table illustrates part of the new table organization, where extensions are sorted first, and then cost is sorted within each primary grouping of extensions:

EXT	COST	DATE
370	0.12	901003
370	0.00	901004
379	0.52	901004
379	0.41	901003
418	0.27	901003

Subsetting a Table with the Where Clause

To produce a subset of data in a table, use the QUERY_TABLE function in conjunction with a Where clause. A Where clause begins with the word Where and is followed by Boolean (AND, OR, NOT) and/or relational operators (<, >, <>, =, >=, <=) that describe how the data is to be subsetted. See the *PV-WAVE Reference* for more information on these operators.

Note You can use PV-WAVE relational operators (EQ, GE, GT, LE, LT, and NE in a Where clause instead of the SQL-style operators listed above.

For example, to create a subset of the phone_data table that only contains calls made on one particular day:

```
new_table = QUERY_TABLE(phone_data, $
   '* Where DATE = 901002')
```

Note The asterisk (*) before Where is a wildcard character that pulls all the columns in phone_data into the resulting table.

Here is a portion of the resulting table – only rows with date 901002 are included:

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901002	093200	21.40	TAC	311	5.78	215	2155554242
901002	094700	1.05	2	358	0.0	303	5553869
901002	094700	17.44	1	320	4.71	214	2145559893
901002	094800	16.23	2	289	0.0	303	5555836
901002	094800	1.31	1	248	.35	617	6175551999

To find the calls made on `901002` with a duration of greater than 10 minutes, enter:

```
new_table = QUERY_TABLE(phone_data, $
   '* Where DATE = 901002 AND DUR > 10.0')
```

The resulting subset is illustrated in the following table. All rows contain dates 90102 *and* durations greater than 10.0.

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901002	093200	21.40	TAC	311	5.78	215	2155554242
901002	094700	17.44	EBH	320	4.71	214	2145559893
901002	094800	16.23	TDW	289	0.0	303	5555836

Tip If you are familiar with SQL, you will see that this Where clause is similar to the Where clause in the SQL SELECT command.

Using Strings in Where Clauses

The Where clause lets you filter strings in a number of different ways. In the simplest case, you want to find information related to a single string, such as a set of initials. For example, to find the calls made by the person with the initials TAC, you can enter:

```
res = QUERY_TABLE(phone_data, $
   '* Where INIT = "TAC" ')
```

Note that the string must be enclosed in quotes inside the function call. Also note that double quotation marks are used to delimit `TAC`. This is because apostrophes were used to delimit the entire QUERY_TABLE string parameter.

Note If the string is passed into the function as a variable parameter, as explained in the section *Passing Variable Parameters into Table Functions* on page 279, then the quotes are unnecessary.

In a more complex case, you can use relational and Boolean operators to filter the strings in a column to find a particular subset of strings. For example, the following command uses relational and Boolean operators to filter the `INIT` column, which contains the initials of callers:

```
res = QUERY_TABLE(phone_data, $
   '* Where (INIT >= "B") AND (INIT < "D") ')
```

The result of this query is a new table containing information on the calls made by people whose initials begin with the letter B and C.

Passing Variable Parameters into Table Functions

Any string or numeric constant used in the QUERY_TABLE function can be passed in as a variable parameter. This means that you can use variables for numeric and string values that are used in the QUERY_TABLE function. For example, you can create a string variable called `name` and use it in the QUERY_TABLE function:

```
name = 'TAC'

tbl = QUERY_TABLE(phone_data, $
   '* Where INIT = name')
```

Because `name` is a variable and not an actual string, you do not have to enclose it in double quotes inside the function call.

The command shown on page 278 that finds the calls made on 901002 with a duration of greater than 10 minutes can also be written with variable parameters in place of actual values:

```
day = 901002
calldur = 10.0

new_table = QUERY_TABLE(phone_data, $
   '* Where DATE = day AND DUR > calldur')
```

Caution If the variable name and the column name in a comparison are the same, the result of the comparison simply returns "true" for all cases, and the desired comparison may not be made. The following example is similar to the previous example, except the `day` variable is changed to `date`, which is also a column name.

```
date = 901002
calldur = 10.0

new_table = QUERY_TABLE(phone_data, $
   ' * Where DATE = date AND DUR > calldur')
```

In this QUERY_TABLE call, `DATE = date` returns “true” for all cases, rather than only for cases where the date is 901002. The comparison `DATE = 901002` is not made as might be expected. Thus, try to choose column names that are different from the variable names.

Using the In Operator

The In operator provides another means of filtering data in a table. This operator tests for membership in a set (one-dimensional array) of values. For example, the following array contains a subset of the initials found in the `INIT` column of the `phone_data` table:

```
nameset = ['TAC', 'BWD', 'TDW', 'RLD']
```

The following QUERY_TABLE call produces a new table that contains information only on the members of `nameset`:

```
res = QUERY_TABLE(phone_data, $
   '* Where INIT In nameset')
```

Combining Multiple Clauses in a Query

You can place more than one clause in a QUERY_TABLE call to produce more complicated and specific queries. Once you understand the basic parts of QUERY_TABLE, combining these parts into more complex queries is a straightforward process.

Note Within the QUERY_TABLE function, the Group By and Order By functions are mutually exclusive. That is, you cannot place both Group By and Order By in the same QUERY_TABLE call.

Example

The following command produces a table that:

- includes only calls with a duration of more than one minute.
- includes only calls with an area code not equal to 303 (out-of-state calls only).
- sorts the table by phone extension, in ascending order.

- sorts the table, within extension subgroups, by date in descending order.
- sorts the table, within date subgroups, by duration in ascending order.

```
result = QUERY_TABLE(phone_data, $
   '* Where (DUR > 1.0) And (AREA <> 303) '+$
   'Order By EXT, DATE Desc, DUR Desc')
```

A portion of the table is shown below:

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901002	094800	1.31	RLD	248	.35	617	6175551999
901002	093200	21.40	TAC	311	5.78	215	2155554242
901002	094700	17.44	EBH	320	4.71	214	2145559893
901004	095000	3.77	DJC	331	1.02	512	5125551228
901003	091500	2.53	DLH	332	.68	614	6145555553
901004	094900	1.93	SRB	379	.52	818	8185552880
901003	091600	1.53	SRB	379	.41	212	2125556618

Using Date/Time Data in Tables

In the previous examples, the DATE column contains long integer values that represent the dates of calls. Instead of using long integers to represent the dates, you may be able to read the date data into a PV-WAVE Date/Time variable. Once in the Date/Time format, the dates can be converted to strings, placed in a table, and manipulated with QUERY_TABLE. In addition, query results can be converted back into Date/Time form and plotted on with a Date/Time axis.

Read the Date Data into a Date/Time Variable

Instead of reading the date data (901002, 901003, etc.) into a long integer, read it into an array of Date/Time variables. For detailed information on reading date data, see *Reading in Your Date/Time*

Data on page 228. This section contrasts the various alternatives you have available for reading date/time data.

Two Methods of Handling Date/Time Data in Tables

This section discusses two ways to handle Date/Time data in a table. It assumes that data has been read into a Date/Time variable. The first method discussed involves converting the Date/Time variable to a string variable, which you can use to build and subset a table. The second method involves manipulating the Date/Time data directly as Julian day values.

Method 1: Convert the Date/Time Data to Strings

Convert the Date/Time variable to a String using the DT_TO_STR procedure. For example:

```
DT_TO_STR, dtdata, dates, Date_Fmt=5
```

This converts the Date/Time values into strings of the format [YY]YY*MM*DD. The advantage of this format is that it allows dates to be compared directly as strings. For example:

```
"1992-02-01"
```

precedes

```
"1993-03-02"
```

Subsetting the Table

Once you have created string variables from the original Date/Time data, you can build a table using these string variables, and use the strings in query commands:

```
this_date=QUERY_TABLE(phone_data, $
   '* Where DATE = "1990-10-03"')
```

Plotting the Table with a Date/Time Axis

To plot the table with a Date/Time axis, you have to first convert the dates back into Date/Time data. To do this, use the STR_TO_DT function. For example:

```
PLOT, STR_TO_DT(phone_data.DATES), $
   phone_data.COST
```

Method 2: Create a Table that Includes the Date/Time Variable

This method deals directly with the Julian day part of the Date/Time structure. Assuming that the Date/Time variable is called DATE, the following commands create a new table containing three columns:

```
JDATE=DATE.Julian
```

Create a new variable JDATE that contains the Julian date equivalents for each date. This is necessary because you cannot place a Date/Time structure directly in a table; tables must consist of vector (one-dimensional array) variables only.

```
new_ph_tbl = BUILD_TABLE("EXT, COST, JDATE")
```

Create the table.

Subsetting the Table

The following query picks out all rows where DATE is less than or equal to October 3, 1990:

```
TDate = VAR_TO_DT(90,10,03)
END_DATE = TDate.Julian
```

Create a Date/Time variable called END_DATE, and set the variable equal to the Julian equivalent of October 3, 1990.

```
New_Table = QUERY_TABLE(new_ph_tbl, $
   '* where JDATE <= END_DATE')
```

Produce a subset of the table.

Plotting the Table with a Date/Time Axis

To plot the resulting table data with a Date/Time axis, the date data must be converted back to a Date/Time variable. The following command performs the conversion:

```
New_Dates = JUL_TO_DT(new_ph_tbl.JDATE)
```

When the data is plotted, PV-WAVE determines that `New_Dates` is a Date/Time variable, and plots a Date/Time axis automatically. For example:

```
PLOT, New_Dates, new_ph_tbl.COST
```

Plots the dates on the X axis and the cost on the Y axis.

For more information on plotting table data, see *Plotting Table Data* on page 286.

Formatting and Printing Tables

The simplest way to print a table is with the command:

```
PRINT, table_name
```

Unfortunately, the output from such a statement is not formatted in a readable, presentable manner.

Printing the Table without Column Titles

To print the `phone_data` table without column headings, simply enter:

```
WAVE> for i=0,N_ELEMENTS(phone_data)-1 do $
   begin PRINT, phone_data(i)
```

This prints a readable, neatly organized representation of the table. The PRINT statement accesses each column of the table directly, using the basic structure notation, which is:

Variable_Name.Tag_Name

For more information on the relationship between structures and tables, see *Tables and Structures* on page 287.

Printing the Table with Column Titles

To achieve a presentable format with column titles requires a slightly more complicated approach. For example, the following procedure prints a formatted version of the `phone_data` table to the screen and places titles above each column. The *Format* keyword in the PRINT statement uses FORTRAN-style format specifiers to format the rows. For detailed information on format specifiers, see Appendix A, *FORTRAN and C Format Strings*, in the *PV-WAVE Programmer's Guide.* You can also refer to the description of the PRINT function in Chapter 2 *Procedure and Function Reference*, in the *PV-WAVE Reference.*

```
PRO pr_table, t_name

PRINT, ' DATE   TIME    DUR    INIT    EXT '+$
   '  COST    AREA    NUMBER '

for i=0, N_ELEMENTS(t_name)-1 do begin

PRINT, Format='(I6,1X,I6,3X,F5.2,3X,' + $
   'A3,3X,I3,2X,F5.2,3X,I3,3X,A10)', $
   t_name(i).DATE, t_name(i).TIME, $
   t_name(i).DUR, t_name(i).INIT, $
   t_name(i).EXT, t_name(i).COST, $
   t_name(i).AREA, t_name(i).NUMBER
ENDFOR
END
```

After the procedure is compiled with .RUN, the following command prints the formatted `phone_table` to the screen:

```
WAVE> pr_table, phone_data
```

```
DATE     TIME     DUR      INIT   EXT    COST    AREA    NUMBER
901002   93200    21.40    TAC    311    5.78    215     2155554242
901002   94700     1.05    BWD    358    0.0     303     5553869
901002   94700    17.44    EBH    320    4.71    214     2145559893
901002   94800    16.23    TDW    289    0.0     303     5555836
901002   94800     1.31    RLD    248    0.35    617     6175551999
901003   91500     2.53    DLH    332    0.68    614     6145555553
901003   91600     2.33    JAT    000    0.0     303     555344
```

```
DATE     TIME    DUR    INIT  EXT   COST   AREA   NUMBER
901003   91600    .35   CCW   418   0.27   303    5555190
901003   91600   1.53   SRB   379   0.41   212    2125556618
901003   91600    .45   MLK   370   0.12   212    2125557956
901004   94700    .80   JAT   000   0.0    303    555320
901004   94900   1.93   SRB   379   0.52   818    8185552880
901004   95000   3.77   DJC   331   1.02   512    5125551228
901004   95100    .16   GWP   370   0.0    303    5551245
901004   95300   1.36   JAT   000   0.0    303    555320
```

Plotting Table Data

You can plot table data easily using the plot procedures. For example, the following example plots the call duration vs. the cost. The PLOT statement accesses the columns of the table directly, using the basic structure notation, which is:

Variable_Name.Tag_Name

For more information on the relationship between structures and tables, see *Tables and Structures* on page 287.

This command produces a scattergram that plots the call duration on the X axis against the cost along the Y axis:

```
PLOT, phone_data.DUR, phone_data.COST, $
   Psym=4, Title='Duration vs. Cost', $
   XTitle='Duration', YTitle='Cost'
```

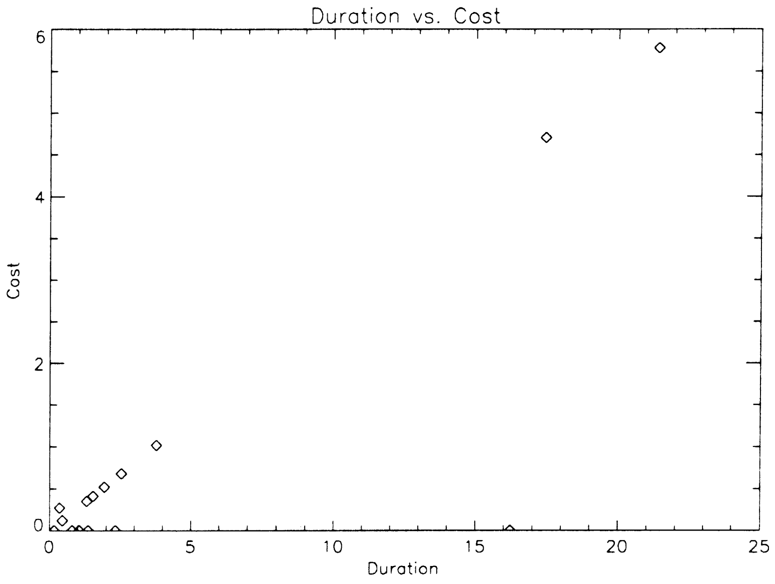


Figure 8-1 Plot of data from a table.

Tables and Structures

As noted previously, a table is represented in PV-WAVE as an array of structures. Although it is not necessary to understand or use structures to use table functions, this section gives a brief overview of their relationship. For more information on structures, see Chapter 6, *Working with Structures*, in the *PV-WAVE Programmer's Guide.*

The basic syntax of PV-WAVE structures is:

{Structure_name, Tag_Name$_1$ *: Tag_Def*$_1$ *, ..., Tag_Name*$_n$ *: Tag_Def*$_n$*}*

The simplest way to refer to a field in a structure is:

Variable_Name.Tag_Name

When you create a table with BUILD_TABLE, the name of the table becomes the *Variable_Name*, and the columns are *Tag_Names* for the underlying structure. The actual name of the structure, *Structure_Name*, is assigned by the system. You can see this name when you list the table's structure with the INFO command. (In the example shown in *Using INFO to View the Table Structure* on page 268, this name is `TABLE_0`.)

You could print the values of one column of `phone_data` with the command:

```
PRINT, phone_data.EXT
```

To print the first fifteen phone extensions, you could enter the command:

```
PRINT, phone_data(0:14).EXT
```

Note Column names *must* be expressed in structure notation when used in the UNIQUE function. For example:

```
dates = UNIQUE(phone_data.DATE)
```

The UNIQUE function is described in the *PV-WAVE Reference.*

Returning Indices of a Subsetted Table

In some situations you might want to build a table and associate index numbers with each row in the result. These index numbers can be useful, particularly when the result of a query generates a very large table that requires a large amount of memory to store. One way to save memory in such a situation is to create a query statement that generates a result containing only the indices of the rows that you are interested in. Then, a print statement allows you to print the rows of interest without first storing them in a variable, which, in some cases, might be too large to hold in memory along with the original table.

The following example demonstrates this technique. In this example, a new table is built from `phone_data` with an extra column called INDEX. This extra column is simply a 1D array of integers in the range {0...14} created with the INDGEN function.

```
INDEX = INDGEN(15)
```

Now, a new table is created from the original table of telephone data, with INDEX included as one of the table's columns.

```
newtbl = BUILD_TABLE(phone_data,
   'INDEX, EXT, DUR, COST')
```

Next, this new table can be subsetted with QUERY_TABLE so that the result contains only the indices of the rows in which you are interested. Because the resulting table contains only the indices of the desired rows, much less memory is required to store the result than if all of the data in the desired rows were stored.

```
result = QUERY_TABLE(newtbl, $
   'INDEX Where COST > .50')
```

Finally, the following statements perform a more meaningful sort, where the indices stored in result are used to locate the desired rows in the newtbl table.

```
FOR i=0, N_ELEMENTS(result) - 1 DO BEGIN $
   PRINT, newtbl(result(i).index)
```

Note This method of subsetting tables based on row indices does not work if a Group By clause is used in the QUERY_TABLE command. The reason for this is that Group By clauses typically return the results of calculations, and these results usually have no counterpart in the original table.

Other Methods of Subsetting and Sorting Variables

PV-WAVE provides other functions for subsetting and sorting the values in one-dimensional arrays. The SORT function sorts the subscripts of an array into ascending order. For example:

```
array = [4, 3, 7, 1, 2]
index = SORT(array)
PRINT, index
   3   4   1   0   2
```

This results because: $A_3 < A_4 < A_1 < A_0 < A_2$. To see the sorted array, enter:

```
PRINT, array(index)
   1   2   3   4   7
```

The WHERE function allows the use of Boolean expressions to select ranges of subscripts in an array. For example:

```
index = WHERE((array GT 50) AND (array LT $
   100))
result = array(index)
```

For more information on these functions, see the *PV-WAVE Reference.*

9

CHAPTER

Software Fonts

PV-WAVE can produce text output using either *software* or *hardware* fonts. Software fonts, sometimes called vector-drawn fonts or Hershey fonts, are internal to PV-WAVE and are drawn with line vectors. Hardware fonts are built into specific output devices, such as PostScript printers and window systems such as X Windows. PV-WAVE simply sends the characters to the graphics device, which displays them using these built-in fonts.

This chapter discusses PV-WAVE's software fonts. See Appendix A, *Output Devices and Window Systems* for detailed information on using the hardware fonts supplied with the devices that PV-WAVE supports.

Software vs. Hardware Fonts: How to Choose

The following sections briefly discuss the things to consider when deciding whether to use software or hardware fonts.

Appearance of Text

Software characters are of medium quality, suitable for most uses. The appearance of hardware-generated characters varies from mediocre (such as the characters found in many terminals) to publication quality (such as provided by PostScript).

3D Transformations

Software characters go through the same 3D transformations as the rest of the plot, yielding a better looking plot. See Chapter 4, *Displaying 3D Data* for examples of software characters used with three-dimensional graphics.

Portability of Text

The appearance and availability of hardware fonts varies greatly from device to device, and thus are not be as portable as software fonts.

In general, the software fonts work the same way on any graphics device and look the same, within the limitations of device resolution. Thus, it is possible to produce graphics on one device and send it to another without worrying about character output. Note, however, that software fonts are scaled relative to the size of the active hardware font. Changing the size of the hardware font will rescale the size of the software font.

Note You may notice that under X Windows the size of the software fonts varies from device to device. When you start PV-WAVE, the PV-WAVE hardware font is set to the current hardware font of the X server. Not all X servers will have the same default font size because users can reconfigure the default font and the default font can differ between X servers. Therefore, you may discover that the hardware font size, and therefore the software font size, may vary across different workstations. You can avoid this by explicitly setting the X font using the DEVICE procedure. For example:

```
DEVICE, font='-adobe-courier-medium-r-normal-
   -14-*'
```

Speed of Plotting

It takes more computer time to draw characters with line vectors, and generally results in more input/output. This is not an important issue, however, unless the plot contains a large number of characters or the transmission link to the device is slow.

Variety

Software fonts provide a great deal of flexibility. There are many different typefaces available, and software fonts can be arbitrarily scaled, rotated, and transformed. See Chapter 5 *Software Character Sets*, in the *PV-WAVE Reference* for a complete listing of the 17 software fonts that are available.

Using Software Fonts

This section explains how to format software text and select different fonts.

You can embed formatting and font commands in the string arguments of plotting keywords such as *Title*, *Subtitle*, *XTitle*, and *YTitle* and in the *string* parameter of the XYOUTS procedure.

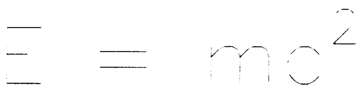
Note To use PV-WAVE software fonts, the value of the system variable !P.Font must be –1. This is the default condition. (To use hardware fonts, set !P.Font to 0.)

Software Font Formatting Commands

You can accomplish a wide variety of text formatting effects, such as subscripting, superscripting, and equation formatting, by embedding formatting commands directly in text strings. For example, the *Title* keyword definition:

```
Title = 'E = mc!U2'
```

produces the following title when plotted:



This example uses the `!U` formatting command, which shifts the 2 up into a superscript. More examples of text formatting appear later in this chapter.

Table 9-1 describes all of the available formatting commands.

Note If you break a line of text using !C, you may have to increase the !X and/or !Y margin fields to allow room for the extra line(s) of text.

Table 9-1: Text Formatting Commands

Format Command	Description
`!A`	Shift above the division line.
`!B`	Shift below the division line.

`!C`	Create a multiple-line annotation. For example:
	`plot, x, y, title= $`
	`'First Line!CSecond Line'`
	(See the Note on the previous page.)

`!D`	Shift down to the first level subscript and decrease the character size by a factor of 0.62.
`!E`	Shift up to the exponent level and decrease the character size by a factor of 0.44.
`!I`	Shift down to the index level and decrease the character size by a factor of 0.44.
`!L`	Shift down to the second level subscript. Decrease the character size by a factor of 0.62.

Table 9-1: Text Formatting Commands

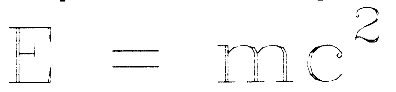
Format Command	Description
!N	Shift back to the normal level and original character size.
!R	Restore position. The current position is set from the top of the saved positions stack.
!S	Save position. The current position is saved on the top of the saved positions stack.
!U	Shift to upper subscript level. Decrease the character size by a factor of 0.62.
!!	Print the ! symbol.

Changing Software Fonts

You can change software fonts by embedding a font selection command directly in a text string. The default font is called Simplex Roman, and its font command !3. The following statement changes the font from the default to Complex Roman (!6):

```
Title = '!6E = mc!U2'
```

This produces the following title when plotted:



You can change the font anyplace in a string by embedding a font command where you want the font change to occur. However, note that the selected font remains in effect until explicitly changed with another embedded font command.

Note Plot titles, subtitles, and axis titles are drawn in a particular order in PV-WAVE. You need to keep this order in mind when you mix the fonts used to annotate plots, because subsequently drawn items "inherit" their font from previously drawn items. The order is:

1. Main title
2. Subtitle

3. X axis numbers
4. X axis title
5. Y axis numbers
6. Y axis title
7. Z axis numbers
8. Z axis title

Tip To achieve some kinds of font combinations in a single plot, you may need to use the OPLOT procedure to overplot some of the text.

There are 17 different software fonts to choose from. They are illustrated in Chapter 5 *Software Character Sets*, in the *PV-WAVE Reference*.

More examples showing font selection appear later in this chapter. Table 9-2 lists the font selection commands.

Table 9-2: Font Selection Commands

Font Command	Description
`!3`	Simplex Roman (default)
`!4`	Simplex Greek
`!5`	Duplex Roman
`!6`	Complex Roman
`!7`	Complex Greek
`!8`	Complex Italic
`!9 (!M)`	Math and special characters
`!10`	Special characters
`!11 (!G)`	Gothic English
`!12 (!W)`	Simplex Script
`!13`	Complex Script
`!14`	Gothic Italian
`!15`	Gothic German

Table 9-2: Font Selection Commands

Font Command	Description
`!16`	Cyrillic
`!17`	Triplex Roman
`!18`	Triplex Italic
`!20`	Miscellaneous

Formatting Commands and Hardware Fonts

The commands listed in Table 9-1 and Table 9-2 have no significance for hardware-generated characters. However, some hardware devices, such as PostScript printers, do have similar types of formatting commands. Therefore, consult the description of the device in Appendix A, *Output Devices and Window Systems* before trying to use these formatting and font commands with hardware characters.

Text Formatting Examples

The following sections demonstrate various ways in which text strings can be formatted. See *Using Software Fonts* on page 293 for details on the formatting and font commands used in these examples.

Example 1: Basic Text Formatting

This example demonstrates the effects of the text formatting commands, where `!N` indicates the normal text level and the original character size. It displays the text using the XYOUTS procedure. The following code produced the text shown in Figure 9-1. In this example, the default font is used.

```
b = '!LLower !NNormal!s!UUp!R!DDown' + $
   '!N!s!AAbove!R!BBelow'

XYOUTS,.02,.2,b,size=3,/Normal
```

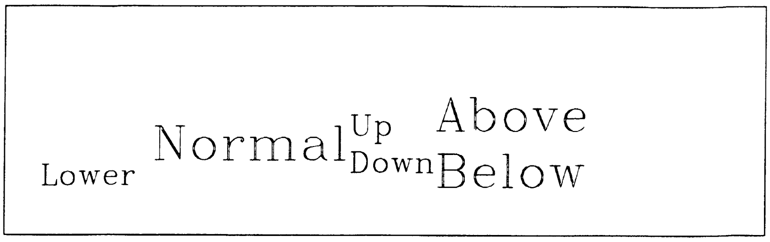


Figure 9-1 Formatted text

Example 2: Changing the Position of Text

This example demonstrates further the use of formatting commands to change the relative position of text. In this example, the font is changed from the default to the Complex Roman font (!6). The result is shown in Figure 9-2:

```
A = '!6!L!!L!S!E!!Exponent!R!I!!Index' + $
    '!N!!N!S!E!!E!R!I!!I!N' + $
    '!S!U!!U!S!I!!I!R!E!!E!R!D!!D!S' + $
    '!E!!E!R!I!!I!N !S!A!!A!S!E!!E!R!I!' + $
    'I!R!B!!B!S!E!!E!R!I!!I'

XYOUTS,.02,.5,A,size=5,/Normal
```

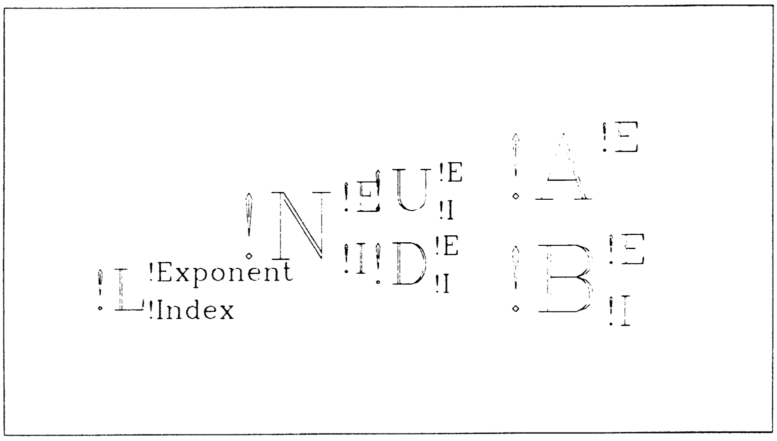


Figure 9-2 Formatted text

Example 3: Multiple Fonts within a Single String

The third example illustrates the effects of changing the font, and illustrates how complex mathematical symbols can be represented. The code used to produce each line is shown in the figure. The *Detailed Discussion* section below explains specifically how the integral term shown at the bottom of Figure 9-3 was produced.

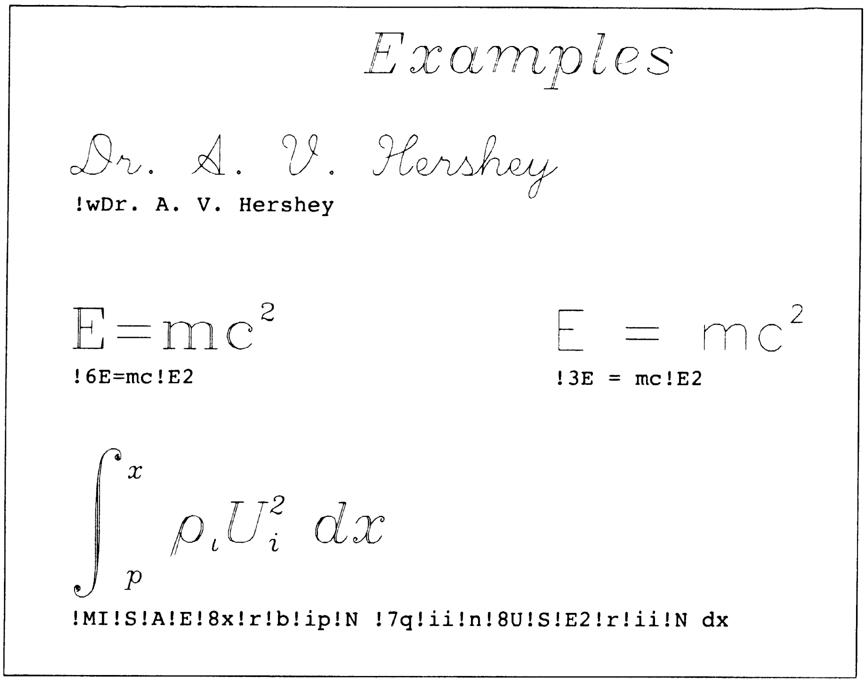


Figure 9-3 Changing text font and formatting mathematical expressions

Detailed Discussion

The bottom integral term shown in Figure 9-3 was formed by the PV-WAVE procedure call:

```
XYOUTS,0,.2,'!MI!S!A!E!8x!R!B!Ip!N!7q' + $
   '!Ii!N!8U!S!E2!R!Ii!N dx',SIZE=3,/NORMAL
```

Table 9-3 summarizes the formatting commands used to produce Figure 9-3.

Table 9-3: Formatting Commands Used in Example 3

Format Command	Description
`!MI`	Changes to math set and draws the integral sign, uppercase I.
`!S`	Saves the current position on the position stack.
`!A!E!8x`	Shifts above the division line and to the exponent level, switches to font 8 the Complex Italic font, and draws the "x".
`!R!B!Ip`	Restores the position to the position immediately after the integral sign, shifts below the division line to the index level and draws the "p".
`!N! 7q`	Returns to the normal level, advances one space, shifts to the Complex Greek font (number 7), and draws the greek letter "rho" which is designated by "q" in this set.
`!Ii!N`	Shifts to the index level and draws the "i" at the index level. Returns to the normal level.
`!8U`	Shifts to the Complex Italic set (number 8), and outputs the uppercase "U".
`!S!E2`	Saves the position and draws the exponent "2".
`!R!Ii`	Restores the position and draws the index "i".
`!N dx`	Returns to the normal level and outputs "dx".

Example 4: Annotating a Plot

This example shows a 2D plot that uses formatted software characters for annotation. The following statements were used to produce Figure 9-4.

```
X = FLTARR(128)
```

Define an array.

```
X(30:40) = 1.
```

Make a step function.

```
X = ABS(FFT(X,1))
```

Take FFT and magnitude.

```
PLOT_OI, X(0:64), Xtitle = '!17Frequency',$
   Ytitle = '!5Power', Title = $
   '!18Example of Vector-Drawn Plot', $
   position = [.2, .2, .9, .6]
```

Produce a Log-Linear plot. Use the Triplex Roman font for the X title (!17), Duplex Roman for the Y title (!5), and Triplex Italic for the main title (!18). The position keyword is used to shrink the plotting "window".

```
ss = '!6F(s) = (2!4p)!e-1/2!N !MI!S!A!E!' + $
   'M!R!B!I!M!!NF(x)e !e-i2!4p!3xs!n!MDx'
```

String to produce equation.

```
XYOUTS,0.1,0.75, ss, Size = 3,/Normal, /Noclip
```

Output string over plot. The NOCLIP keyword is needed because the previous plot caused the clipping region to shrink.

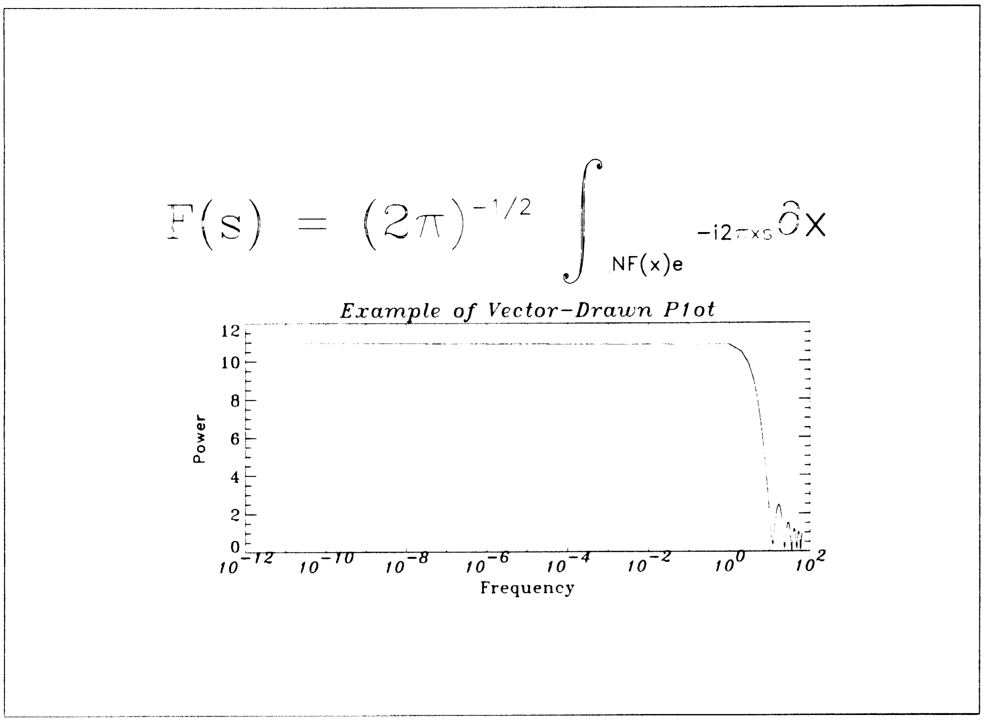


Figure 9-4 Example of a plot drawn with software text

10

CHAPTER

Using Color in Graphics Windows

There are numerous systems for measuring and specifying color; these systems typically have three components. PV-WAVE accepts color specifications in the RGB, HLS, or HSV color systems.

Understanding Color Systems

A *color system* is an algorithm for defining color. Color values are defined in the specified color system and then used by the color table to control the colors on the screen. Different systems use different combinations of values to describe the same color.

Most devices capable of displaying color use the RGB (red, green, blue) color system. Other common color systems include the Munsell, the HSV (Hue, Saturation, Value), the HLS (Hue, Lightness, Saturation), and the CMY (Cyan, Magenta, Yellow) color systems. Many algorithms have been written to convert colors from one system to another, and PV-WAVE has the conversion routines you will need to successfully use color with PV-WAVE graphics. From the command line or from within a program or compiled procedure, you can use the COLOR_CONVERT procedure to convert vector or scalar color table values from one system to another.

Color System Overview

The color systems available in PV-WAVE include:

- **RGB** – Red, Green, and Blue
- **HLS** – Hue, Lightness, and Saturation
- **HSV** – Hue, Saturation, and Value

Note For either 8-bit or 24-bit color, RGB is the default color system for PV-WAVE.

For a more complete discussion of color systems, refer to either of these sources:

- *Fundamentals of Interactive Computer Graphics*, J.D. Foley and A. Van Dam, Addison-Wesley Publishing Company, Reading, MA, 1982.
- *Computer Graphics: Principles and Practice*, by Foley, Van Dam, Feiner, and Hughes, Second Edition, Addison Wesley Publishing Company, Reading, MA, 1990.

Parts of this discussion are taken from these books.

The RGB Color System

The RGB color system uses a three-dimensional Cartesian coordinate system with the value of each color ranging from 0 to 255. Each displayable color is a point within this cube, as shown in Figure 10-1. The origin, (0, 0, 0), where each color coordinate is 0, is black. The point at (255, 255, 255) is white and represents an additive mixture of the full intensity of each of the three colors. Points along the main diagonal are shades of gray, because the intensity of each of the three primaries is equal.

All primary and secondary colors are found on corners of the cube. For example, refer to Figure 10-1 on page 307, and notice that the color yellow is represented by the coordinate (255, 255, 0), or a mixture of 100% red plus 100% green plus 0% blue.

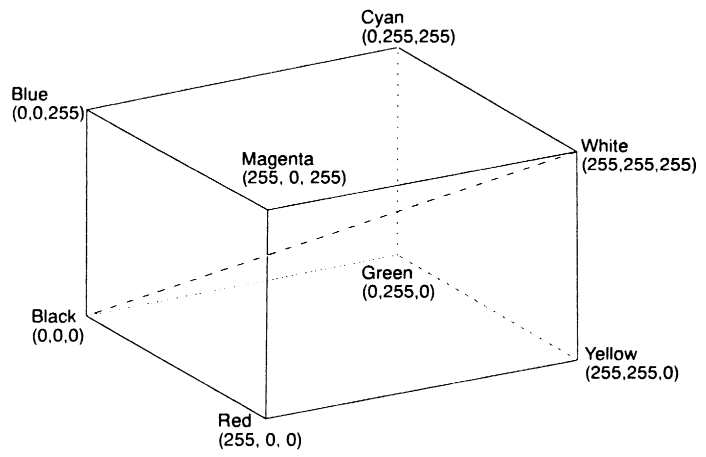


Figure 10-1 RGB color cube. Primary and secondary colors are located at the corners of the cube; grays are along the main diagonal. (After Foley and Van Dam).

How RGB Color Triples Map into Pixels

Typically, digital display devices represent each component of an RGB color coordinate as an n-bit integer in the range of 0 to $2^n - 1$. Each displayable color is an RGB coordinate triple of n-bit numbers — this yields 2^{3n} total colors. For the common example of 8-bit colors, each color coordinate may range from 0 to 255, and the number of color combinations to choose from would be 2^{24} or 16,777,216 colors.

A display with an m-bit pixel can represent 2^m colors simultaneously, as long as the display actually possesses that many pixels. For the increasingly common case where the red, green, and blue components of the color are *each* represented with an 8-bit value, 24-bit pixels are required to present as many colors on screen as there are pixels. Another common case is a display with only 8 bits per pixel, which permits the simultaneous display of $2^8 = 256$ colors selected from the much larger set of 2^{24} colors.

For a more thorough comparison of 8-bit and 24-bit displays, refer to *Colormapped Graphics* on page A-78.

If there are not enough bits in a pixel to represent all colors, or in other words, $m < 2^{3n}$, a *color translation table* (also known as a *color lookup table* or simply a *color table*) is used to associate the value of a pixel with a color triple. This table is an array of color triples with an element for each possible pixel value. Given 8-bit pixels, a color table containing $2^8 = 256$ elements is required. The color table element with an index of i specifies the color for pixels with a value of i.

To summarize, given a display with an n-bit color representation and an m-bit pixel, the color translation table, C, is a 2^m long array of RGB triples:

$$C_i = \{r_i, g_i, b_i\},\ 0 \le i < 2^m$$

$$0 \le r_i, g_i, b_i < 2^n$$

Objects containing a value, or color index, of i are displayed with a color of C_i.

The HSV and HLS Color Systems

The HSV and HLS color systems can be represented as a color solid; HSV uses an ordinary cone, while HLS uses a two-pointed cone. Any cross section through the solid represents a particular color wheel, in which saturation increases radially from the center. As the cross sections progress from the base of the cone to the top point of the cone, the resulting color wheels increase in lightness.

Note In both the HLS and the HSV color systems, hue can vary through a range of 0 degrees (red) to 120 degrees (green) to 240 degrees (blue) to 360 degrees (red).

The HLS Color System

The HLS system is based on the Ostwald color system, which uses hue, lightness, and saturation values, as defined below:

- *Hue* is a term used to distinguish between colors. It is usually represented as a 360-degree color wheel, with red at 0 degrees, green at 120 degrees and blue at 240 degrees. Complementary colors are 180 degrees apart on the wheel.
- *Lightness* corresponds to what is intuitively known as the brightness or intensity of a color.
- *Saturation* refers to how pure (or conversely, how diluted with white) a color is. For example, saturation is what distinguishes lavender from purple, or sky blue from royal blue.

In other words, with the HLS color system, each color index (color table color) comprises values of hue, lightness, and saturation. Hue represents a gradation of color ranging through all the colors. When you select a color in the color table and change its hue, you get a different color. Lightness defines the color on a scale from dark to light, with zero being black and 100 white.

Modifying the saturation produces colors that are more or less gray. A zero value for saturation produces a gray color, while a saturation of 100 produces a pure color with no gray. Saturation has no effect at the extreme ends of the cone (i.e., when lightness equals either 0 or 1).

The HSV Color System

HSV is based on hue, saturation, and value elements, as defined below:

- *Hue* is a term used to distinguish between colors. It is usually represented as a 360-degree color wheel, with red at 0 degrees, green at 120 degrees and blue at 240 degrees. Complementary colors are 180 degrees apart on the wheel.
- *Saturation* refers to how pure (or conversely, how diluted with white) a color is. For example, saturation is what distinguishes red from pink, or meadow green from hunter green.

- *Value* corresponds to what is intuitively known as the brightness or intensity of a color.

In other words, with the HSV color system, each color index (color table color) comprises values of hue, saturation, and value. Hue represents a gradation of color ranging through all the colors. When you select a color in the color table and change its hue, you get a different color. Saturation represents a range of the color from white (zero) through the fully saturated color (100). Value is a range from black (zero) through the pure color (100).

Using Color to Enhance Visual Data Analysis

Color is a valuable aid in the visual analysis process, because it can be used to take advantage of the human brain's capability to distinguish fine gradations of shade and intensity. Color can also be used to simply draw one's attention to a certain part of the screen, or to a certain region of a plot or image.

This section discusses three aspects of PV-WAVE's powerful color capabilities:

✔ loading predefined and custom color tables

✔ modifying color tables to create special effects

✔ plotting colors used for the elements of "simple" plots

Note To use the color capabilities of PV-WAVE, you must use a workstation that is capable of utilizing a display with color. However, no information is lost if you open a saved session on a monochrome or gray-scale workstation that was originally saved on a color workstation.

Experimenting with Different Color Tables

Most color workstations cannot display more than a certain number of colors (usually 256) at once. For this reason, color tables are used to map red, green, and blue values into the available colors on the workstation.

PV-WAVE includes an assortment of 16 predefined color tables with enough variety to produce visually pleasing results for many applications, or you can define your own color table.

You can use either the TVLCT or the LOADCT procedures to load the color table on the current device:

- **LOADCT** – This procedure loads predefined color tables stored in the file `colors.tbl` (found in `$WAVE_DIR/bin`, one portion of the main PV-WAVE install area).
- **TVLCT** – This procedure loads color tables stored in user-defined variables. Once the variables are loaded into the color table, it is used like any other color table.

PV-WAVE's color table functions let you modify the colors used to display images, shaded surfaces, and vector graphics inside graphics windows. Vector graphics colors (also called *plot colors*) let you control the colors assigned to elements of line plots, scatter plots, contour plots, and unshaded surfaces. For more information on how to manipulate PV-WAVE's plot colors, see *Controlling PV-WAVE's Plot Colors* on page 321.

For color and gray-scale devices, the default is to display 8-bit graphics using the PV-WAVE color table B–W Linear (standard color table number 0). On a monochrome display, by default, color graphics are dithered. For more information about dithering, see *Displaying Images on Monochrome Devices* on page 148.

Number of Colors in the Color Table

The color table will allocate as many colors as it can, but the number of colors it can actually use is affected by the type of system you have and its configuration:

- **Graphics output to "simple" graphics devices** (for example, Tektronix terminals or emulators) – The color table defines all 256 colors, even though the device can probably only uniquely display a much smaller number of colors, such as 16, 32, or 64. The device will automatically begin to reuse colors whenever it reaches its limit.

- **Graphics output in a multi-tasking windowing environment** (for example, the X Window System) – By default, the color table defines (and allocates) every color that has not been previously allocated by the window manager or some other application.

For more information about how to reserve colors for the window manager in an X environment, refer to *Reserving Colors for Other Applications' Use* on page A-84.

Loading a Predefined Color Table: LOADCT

Use the LOADCT procedure to load one of the predefined color tables. There are 16 color tables, ranging from 0 to 15, in the file `colors.tbl` (found in `$WAVE_DIR/bin`, one portion of the main PV-WAVE install area). The standard color tables are:

Table 10-1: Standard Color Tables

No.	Color Table Name	No.	Color Table Name
0	B–W Linear	8	Green/White Linear
1	Blue/White	9	Green/White Exponential
2	Green/Red/Blue/White	10	Green/Pink
3	Red Temperature	11	Blue/Red
4	Blue/Green/Red/Yellow	12	16 Level
5	Standard Gamma-II	13	16 Level II
6	Prism	14	Steps
7	Red/Purple	15	PV-WAVE Special

LOADCT has one parameter – the index of the predefined color table to be loaded.

Tip To obtain a menu listing of the available color tables, call LOADCT with no parameters.

Loading Your Own Color Tables: TVLCT

Use the TVLCT procedure to load a color table using data stored in variables. When calling TVLCT, you supply three vectors containing the intensity or value of each color (red, green, and blue) for each index. Given an n-bit color representation, each element must be in the range of 0 to $2^n - 1$. These vectors may contain up to 2^m elements, assuming the display contains m-bit pixels. You can also supply an index pointing into the color translation table, but this is optional. If not specified, a value of 0 is used, causing the tables to be loaded starting at the first element of the translation table vectors.

The TVLCT procedure can also use optional keyword parameters. For information on the keywords, see the TVLCT description in the *PV‑WAVE Reference.*

Example — Modifying Color Tables from the Command Line

The INDGEN function is well-suited for creating larger color tables in which each color's intensity can be expressed as a function of its index. In this example, INDGEN is used to create a linear 256-element color table that is then used to display images in a variety of ways:

```
A = INDGEN(256)
```

Create a "straight line" variable, A(I)=I.

```
TVLCT, A, A*0, A*0
```

Display image with a linear red scale; disable green and blue.

```
TVLCT, A, A, A
```

Display image with linear black and white scale.

```
TVLCT, A, 2 * (A-128) > 64, 4 * (A-192) > 0
```

Display image with a warm-body temperature scale. Red is linear (starting at 0), green starts at 128, and blue starts at 192.

Modifying the Color Tables

PV-WAVE provides many commands and widget-based utilities that you can use to modify existing color tables and to create new ones. Many of the possibilities are described in the following sections.

Note The color table modifications discussed in this section only affect the contents of PV-WAVE graphics windows. If you need to control the colors used in the background, foreground, and border of your window-managed GUI (graphical user interface), you must use different techniques than those described in this section. For more information about selecting GUI colors, refer to *Setting Colors and Fonts* on page 463 of the *PV-WAVE Programmer's Guide.*

Modifying Color Tables from the PV-WAVE Prompt

The MODIFYCT procedure is used to update the file `colors.tbl` (found in `$WAVE_DIR/bin`, one portion of the main PV-WAVE install area) with a new color table (i.e., a new named color table that will take the place of one of the color tables in `colors.tbl`). This procedure should only be used by persons authorized to change the predefined color tables supplied with PV-WAVE. In other words, you may need to contact your System Administrator for assistance.

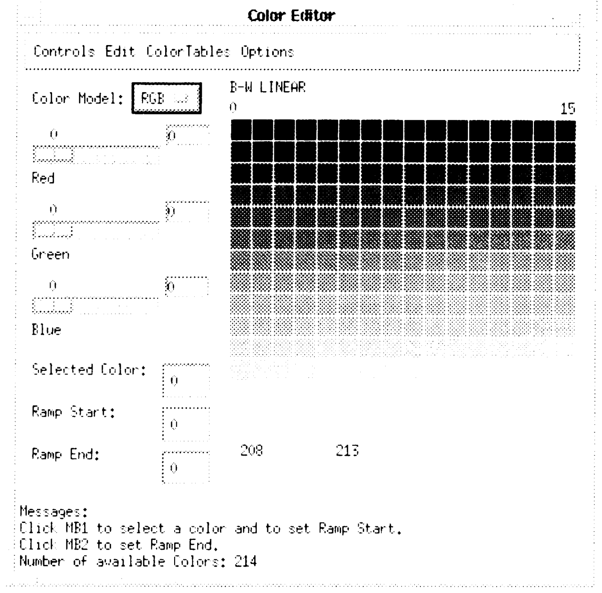
Note Except for editing `colors.tbl` directly, the MODIFYCT command is the only way to modify the predefined standard color tables. For detailed information about using the MODIFYCT command, refer to the MODIFYCT description in the *PV-WAVE Reference.*

Modifying Color Tables Using Widget-based Utility Tools

PV-WAVE provides several widget-based tools that you can use to interactively modify the color tables.

- **WgCeditTool** – A full-featured set of menus and widgets enclosed in a window; this window allows you to edit the values in PV-WAVE color tables in many different ways. WgCeditTool also provides a way to save your color table changes using a name that you choose. The WgCeditTool window is shown in Figure 10-2.
- **WgCbarTool** – A simple vertical or horizontal color bar that can be used to interactively shift a PV-WAVE color table. WgCbarTool can easily be included inside larger container widgets. The WgCbarTool window is shown in Figure 10-3 on page 318.
- **WgCtTool** – A simple widget that can be used interactively to modify a PV-WAVE color table. WgCtTool provides widgets that you can use to stretch, rotate, and reverse the current color table. The WgCtTool window is shown in Figure 10-4 on page 320.

WgCeditTool, WgCbarTool, and WgCtTool can only be used if you are also running a window manager. If you are an experienced programmer, consider providing access to these widget-based utility tools via your PV-WAVE application, so that people using your application can interactively modify and create new color tables without entering commands at the WAVE> prompt.



Messages:
Click MB1 to select a color and to set Ramp Start.
Click MB2 to set Ramp End.
Number of available Colors: 214

Figure 10-2 PV-WAVE provides several widget-based tools that you can use to interactively modify the color tables. Here, the WgCeditTool window displays color table number 0, B–W Linear. The WgCeditTool window lets you use the mouse to create a new color table based on either the HLS, HSV, or RGB color systems.

Shifting the Color Table to the Left or Right

You can "shift" the color table to the left or right to change the color indices that are associated with each color value. All the color table values are still present in the color table, but the color table uses different colors for the start and the end.

Shifting Colors from the PV-WAVE Prompt

This section discusses how the color table's red, green, and blue components can be altered with the SHIFT function to produce a modified color table. Shifting the color table in this manner produces the same basic effect as shifting it with WgCbarTool, except you can precisely control the amount of shifting that occurs and use different amounts of shifting for the red, green, and blue components. For information about WgCbarTool, see the next section.

Tip Following the same basic procedure, you can experiment with other PV-WAVE functions and procedures to produce different effects in the color table, such as producing a nonlinearly interpolated color ramp.

To shift the color table, access the three color variables, as described in *Retrieving Information About the Current Color Table* on page 321. Then process each of the three variables (red, green, and blue) by shifting them some amount, such as:

```
shr = SHIFT(r_curr, 28)
shg = SHIFT(g_curr, 56)
shb = SHIFT(b_curr, 84)
```

The amount of shifting can be any integer amount up to 255 (if you are using all 256 colors). Remember that a shift of zero (0) is equivalent to no shifting of the variable.

After the variables are processed, use them to load the current color table using the TVLCT command.

Shifting Colors Using the Utility Widget WgCbarTool

WgCbarTool creates a simple color bar that can be used to view and interactively shift a PV-WAVE color table. The horizontal form of the color bar is shown in Figure 10-3.

To rotate the color table using the color bar, press and drag the left mouse button inside the color bar. As you “slide” colors into different color table indices, the colors that “scroll off” the end of the color table are added to the opposite end.

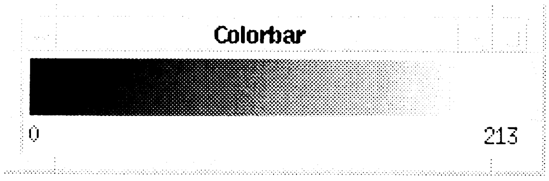


Figure 10-3 WgCbarTool creates an array of colors that match the colors in the current color table; the color array can be shifted to the right or left using the mouse. This color bar widget has been created using the /Horizontal option; the default is for the color bar to be displayed in a vertical orientation.

Smoothing the Color Transitions in a Color Table

This section describes a technique for smoothing out any place in a color table where there is a sharp transition from one color to the next color. This technique involves the SMOOTH function.

This technique helps a color table seem less harsh and helps reduce the banding or "contouring" artifact evident in color tables that have rapid transitions between colors. Otherwise, you may see an artificially pronounced transition in data that actually has no rapid transitions.

To smooth the color table, access the three color variables, as described in *Retrieving Information About the Current Color Table* on page 321. Then process each of the three variables (red, green, and blue) by smoothing them by some amount, such as:

```
smr = SMOOTH(r_curr, 5)
smg = SMOOTH(g_curr, 5)
smb = SMOOTH(b_curr, 5)
```

Initially, start out with a width of 5 (this is the width of the "boxcar" smoothing window), and adjust that width up or down as needed to get the most pleasing results. Additionally, you may want to broaden the boxcar smoothing width if you are using all 256 colors.

After the variables are processed, use them to load the current color table using the TVLCT command.

Stretching the Color Table

This section describes how to linearly expand a range within a color table to provide more detail for that range of pixel values.

Stretching Colors from the PV-WAVE Prompt

The color table's red, green, and blue components can be stretched to emphasize a certain range of values in the image. For example, you could enter this command:

```
STRETCH, 15, 90
```

STRETCH linearly interpolates new red, green, and blue color vectors between the low number and the high number. In other words, the low number (15) is the pixel value that is displayed with color index 0, and the high number (90) is the pixel value that is displayed with the highest color index available. Pixel values between 15 and 90 are displayed proportionately, and pixels outside the range are displayed with either the "low color" or the "high color".

Note The STRETCH procedure does not affect the data or the current color table stored in the `Colors` common block; it only affects the way the data is displayed. For information about the `Colors` common block, see *Retrieving Information About the Current Color Table* on page 321.

Stretching Colors Using the Utility Widget WgCtTool

WgCtTool allows you to interactively modify system color tables by stretching, rotating, and reversing them. A range of color table indices, as defined by the Stretch Bottom and Stretch Top sliders, can be linearly stretched.

The Stretch Bottom number is used for the first parameter to the STRETCH command, and the Stretch Top number is subtracted from the number of colors available in the color table to determine the second parameter to the STRETCH command. For more information about the STRETCH command, refer to the previous section or to the description for STRETCH in the *PV-WAVE Reference*.

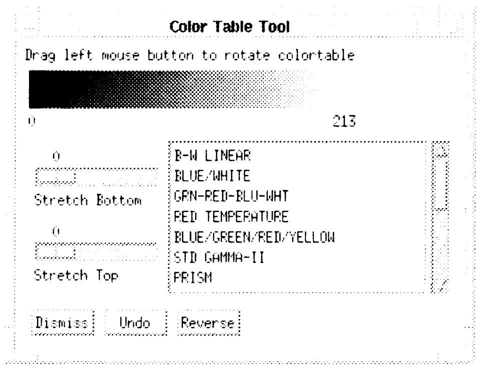


Figure 10-4 The WgCtTool window lets you interactively modify system color tables by stretching, rotating, and reversing them. The color bar in WgCtTool is the same one shown separately in Figure 10-3.

Note Because system color tables are "read-only", no system color table will be permanently altered by any changes you make with the WgCtTool window.

Retrieving Information About the Current Color Table

Most color table procedures maintain the current color table in a common block called Colors, defined as follows:

```
COMMON Colors, r_orig, g_orig, b_orig, $
   r_curr, g_curr, b_curr
```

The variables are integer vectors of length equal to the number of color indices. Your program can access these variables by declaring the common block. The convention is that routines that modify the current color table should read it from `r_orig`, `g_orig`, and `b_orig`, and then load and leave the resulting color table in `r_curr`, `g_curr`, and `b_curr`.

Controlling PV-WAVE's Plot Colors

The currently loaded color table determines PV-WAVE's plot colors (see the following discussion entitled *Default Plot Colors*). *Plot colors* are those colors used to display data, axes, axis titles, and other elements of PV-WAVE line plots, scatter plots, contour plots, and unshaded surfaces. But it is possible that the current color table (or any other standard color table) does not provide the colors you wish to use for plotting your data.

For example, in many of the color tables, there are only subtle differences between adjacent colors in the middle range of the color table. This makes many of the standard color tables better suited for the display of images than they are for the display of data inside a line plot or bar chart.

To customize your plot colors, load new red, green, and blue color vectors defining the colors you want. To load new color vectors, use the technique demonstrated in the example in this section, or use the TVLCT procedure. The TVLCT procedure is discussed in *Loading Your Own Color Tables: TVLCT* on page 313.

Another quick way to obtain a nice set of plot colors is using the TEK_COLOR procedure. The TEK_COLOR procedure is handy because it loads a color table that mimics the 32 distinct plot colors of the Tektronix 4115 display device. The TEK_COLOR procedure is discussed in *Using the TEK_COLOR Command to Control Plot Colors* on page 326.

For more information about PV-WAVE's standard color tables, refer to *Loading a Predefined Color Table: LOADCT* on page 312.

Default Plot Colors

By default, when drawing vector graphics on the screen using the default color table, B–W Linear, PV-WAVE draws a white line on a black background. However, you can use the *Color* keyword (with the PLOT command) to choose any other available color. For example:

```
PLOT, x_data, y_data, Color=144, ... ...
```

Unless you supply the *Color* keyword, color index 0 (often a dark color) is used for the background, and the highest color index (often a light color) is used for the lines. This means that by default, a light color is used for plotting data, axes, titles, and so forth. The highest color index is stored in a PV-WAVE system variable, !D.N_Colors. For more information about !D.N_Colors, see the section entitled *Determining the Number of Available Plot Colors.*

Note Some hardcopy devices that print on white paper automatically swap the foreground and background colors so that on paper, the lines are drawn with a dark color instead of a light color. Otherwise, the hardcopy would be drawn with white on white, and the paper would appear blank. (Please be aware that some color output devices do not adhere to this convention, although most monochrome output devices do.)

Determining the Number of Available Plot Colors

Use one of PV-WAVE's system variables, !D.N_Colors, to find out the number of simultaneously available colors for a particular device. In the case of devices with windows, this field of the device (!D) system variable is set after the window is initialized. For monochrome systems, !D.N_Colors is 2, and for color systems it is normally 256.

If you are using PV-WAVE in a multi-tasking environment under the control of a window manager, some color indices may be reserved for the window manager and for other applications that are running simultaneously with PV-WAVE, and this in turn will affect the value of D.N_Colors.

For more information about why colors are reserved for the window manager in an X environment, refer to *Reserving Colors for Other Applications' Use* on page A-84.

Tip If you are seated at a workstation using a window manager, an easy, interactive way to inquire the current value of !D.N_Colors is to enter the COLOR_PALETTE command. This procedure displays an array of color cells and the color table index associated with each one, as shown in Figure 10-2. The largest number you see displayed in the array of color cells reflects the current value of !D.N_Colors.

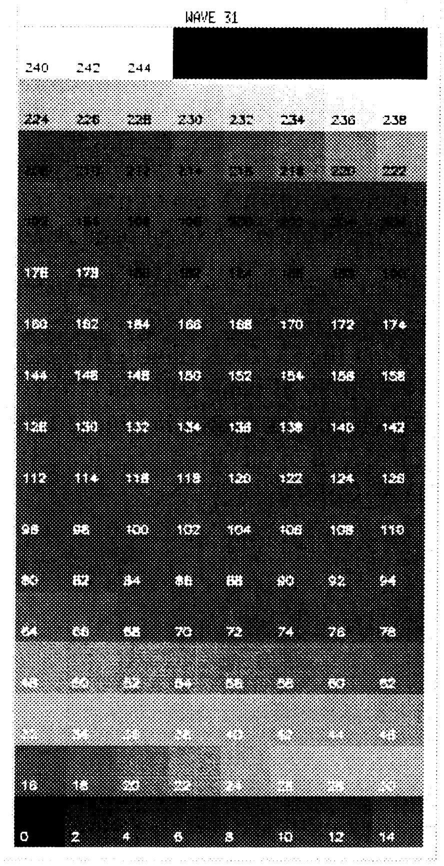


Figure 10-5 PV-WAVE provides sixteen standard color tables, and users can easily modify these color tables or define their own. Here, the COLOR_PALETTE procedure is displaying every other color in the current color table, along with its color table index. The largest number displayed, 244, reflects the current value of !D.N_Colors. The five black cells in the upper-right corner of the window represent colors that are not available to PV-WAVE because they have been reserved by some other application.

Example — Creating a Simple Color Table to Control Plot Colors

This example creates a graph with the axes drawn in white (on a black background), then successively adds red, green, blue, and yellow lines. Because five distinct colors are needed, plus one color for the background, a six-element color table is created. In this color table, color index 0 represents black (0, 0, 0), color index 1 is white (1, 1, 1), 2 is red (1, 0, 0), 3 is green (0, 1, 0), 4 is blue (0, 0, 1), and 5 is yellow (1, 1, 0).

```
red = [0,1,1,0,0,1]
```

Specify the red component of each color (1 = full intensity, 0 = no intensity)...

```
green = [0,1,0,1,0,1]
```

... green component.

```
blue = [0,1,0,0,1,0]
```

... blue component.

```
TVLCT, 255*red, 255*green, 255*blue
```

With a single command, multiply each element in the red, green, and blue vectors, and load the first six elements of the color table. The remaining colors in the color table are not affected.

```
PLOT, Color=1, /Nodata, ... ...
```

Draw the axes in white, color index 1.

```
OPLOT, Color=2, ... ...
```

Draw in red.

```
OPLOT, Color=3, ... ...
```

Draw in green.

```
OPLOT, Color=4, ... ...
```

Draw in blue.

```
OPLOT, Color=5, ... ...
```

Draw in yellow.

Note For this example to work on your display, your display must have at least three bits per pixel so it can simultaneously represent six colors. An 8-bit color table is assumed.

Using the TEK_COLOR Command to Control Plot Colors

Another easy way to change the colors in the lower end of the color table is to enter the TEK_COLOR command; this command loads 32 predefined, unique, highly saturated colors into the bottom 32 indices of the color table. When the TEK_COLOR color table is in effect, you will be able to easily differentiate the different data sets in a line plot.

Example

The TEK_COLOR procedure has no keywords or parameters, so it is simple to use. This example shows how the TEK_COLOR colors can be used to produce bright, vibrant fill colors.

```
b = FINDGEN(37)
x = b * 10
y = SIN(x * !Dtor)
```

Create an array containing the values for a sine function from 0 to 360 degrees.

```
PLOT, x, y, XRange=[0,360], XStyle=1, YStyle=1
```

Plot data and set the range to be exactly 0 to 360.

```
COLOR_PALETTE
```

Put up a window containing a display of the current color table and its associated color indices.

```
TEK_COLOR
```

Load a predefined color table that contains 32 distinct colors.

```
POLYFILL, x, y, Color=6
POLYFILL, x, y/2, Color=3
POLYFILL, x, y/6, Color=4
```

Fill in areas under the curve with different colors.

```
z = COS(x * !Dtor)
```

Create an array containing the values for a COS function from 0 to 360 degrees.

```
OPLOT, x, z/8, Linestyle=2, Color=5
```

Plot the cosine data on top of the sine data.

Note The color table indices specified with the *Color* keyword must be in the range {0 ... 31} to take advantage of the bright colors created by the TEK_COLOR procedure. Color table indices above 31 are not affected by the TEK_COLOR procedure, and will remain as defined by the previously loaded color table.

Specifying Plot Colors on a 24-bit Display

For your convenience, PV-WAVE allows 24-bit colors to be specified in hexadecimal notation. You may want to enter hexadecimal colors if you are controlling plot colors on a 24-bit display. Because this is most frequently done while operating under the control of a window manager in an X Window System environment, refer to *Specifying 24-bit Plot Colors* on page A-92 for more details.

Device-specific Methods for Using Color

Use the SET_PLOT procedure to direct the graphics output to different devices. A scalar string you provide with the command identifies the device to which you wish to send graphics output.

Color Tables — Switching Between Devices

Because devices have differing capabilities, and not all are capable of representing the same number of colors, the treatment of color tables when switching devices is somewhat tricky. See Appendix A, *Output Devices and Window Systems*, for details on each supported device.

After selecting a new graphics output device, SET_PLOT will perform one of the following color table actions depending upon which keyword parameters are specified:

- **Do nothing** – This is the default action. The problem with this treatment is that PV-WAVE's internal color table incorrectly reflects the state of the device's color table until TVLCT is called.
- **Copy the device color table** – If the *Copy* keyword parameter is set, the internal color table is copied into the device.

This is the preferred method if you are displaying graphics and each color index is explicitly loaded.

The color table copying is straightforward if both devices have the same number of color indices. If the new device has more colors than the old device, some color indices will be invalid. If the new device has less colors than the old, not all the colors are saved.

Note When the *Interpolate* keyword is set, the new device color table is loaded by interpolating the old color table to span the new number of color indices. This method works best when displaying images with continuous color ranges.

Combining Colors to Create Special Effects

If you are using an X-compatible window manager, you can use the write mask to specify one or more color planes whose bits you wish to manipulate or the plane you want to use to create the special effects. The way the special effects are rendered also depends on the value you provide for the graphics function. For more details, refer to *Using the Write Mask and Graphics Functions to Manipulate Color* on page A-93.

The write mask is used to control how one graphics pattern interacts with another graphics pattern when plotting to a PV-WAVE graphics window. The write mask allows you to create special effects when overlaying or superimposing images and patterns. For example, some 24-bit displays allow the screen to be treated as two separate 12-bit images. This allows for "double-buffering", a technique useful for animation, or for storing distance data to simplify hidden line and plane calculations in 3D applications.

Another possible application for the write mask is to simultaneously manage two 4-bit-deep images in a single PV-WAVE graphics window instead of a single 8-bit-deep image. You could use the write mask to control whether the current graphics operation operates on the "top" image or the "bottom" image.

Summary of Color Table Procedures

The Standard Library procedures listed in this section are used to manipulate color tables. Some of the procedures are basic procedures that you use programmatically to change color tables, and others are window-based procedures that facilitate interactive modifications. Some of the procedures are "built" using WAVE Widgets and consequently can be incorporated into applications that run under either the Motif or OPEN LOOK window managers. For detailed information on any of these routines, see the *PV-WAVE Reference.*

Basic Color Table Procedures

These commands can always be entered at the WAVE> prompt, even if you are not running PV-WAVE under the control of a window manager:

- **COLOR_CONVERT** – This procedure converts vector or scalar color table values from one color system to another. The supported color systems are HSV, HLS, and RGB.
- **HIST_EQUAL_CT** – This procedure uses an input image parameter, or the region of the display you mark, to obtain a pixel distribution histogram. The cumulative integral is taken and scaled, and the result is applied to the current color table.
- **HLS** – This procedure makes and loads color tables based on the HLS color system. This system is based on the Ostwald color system. As with the HSV procedure, spirals are interpolated in a three-dimensional color space.
- **HSV** – This procedure makes and loads color tables based on the HSV color system. A spiral through the single-ended HSV cone is traced. The color representation of pixel values is linearly interpolated from beginning and ending values of hue, saturation, and value.
- **LOADCT** – This procedure loads predefined color tables. To obtain a menu listing of the available color tables, call LOADCT with no parameters.

- **MODIFYCT** – This procedure is used to update the file (`colors.tbl`) that lists the system color tables. This procedure should only be used by persons authorized to change the predefined color tables supplied with PV-WAVE.
- **PSEUDO** – This procedure generates and loads a pseudo color table based on the HLS color system. The colors it generates are theoretically "a near maximal entropy mapping" for the eye. The parameters are similar to those used with the HLS and HSV procedures.
- **STRETCH** – This procedure linearly expands the entire range of the last color table loaded by a PV-WAVE procedure to cover a given range of pixel values.
- **TEK_COLOR** – This procedure loads a color table that mimics the 32 distinct plot colors of the Tektronix 4115 display device. These colors ensure that the various datasets in a line plot or bar chart are easy to differentiate.
- **TVLCT** – This procedure loads color tables stored in variables. Once the variables are loaded into the color table, it is used like any other color table.

Interactive Color Table Procedures

The procedures listed in this section create windows of varying complexity that can be used to interactively make modifications to PV-WAVE color tables.

Note The windows in the second list ("generic") can operate with either the Motif, OPEN LOOK, or SunView window managers, but if you are using the X Window System in conjunction with either Motif or OPEN LOOK, you will probably prefer to use the procedures from the first list (in section *Interactive (Wave Widgets) Color Table Procedures*).

Interactive (Wave Widgets) Color Table Procedures

The procedures listed in this section are WAVE Widgets applications, and thus are available using either the Motif or OPEN LOOK look-and-feel. For more information on WAVE Widgets, refer to Chapter 15, *Using WAVE Widgets*, in the *PV-WAVE Programmer's Guide*.

- **WgCbarTool** – This procedure creates a simple color bar that can be used to view and interactively shift a PV-WAVE color table.
- **WgCeditTool** – This procedure creates a full-featured set of menus and widgets enclosed in a window; this window allows you to edit the values in PV-WAVE color tables in many different ways.
- **WgCtTool** – This procedure creates a simple widget that can be used interactively to modify a PV-WAVE color table.

Note The window-oriented procedures listed in this section will not work unless you are using an X-compatible window manager, such as Motif or OPEN LOOK. All procedures are written in the PV-WAVE language and they all use the TVLCT procedure to load the color tables.

Interactive (Generic) Color Table Procedures

The procedures listed in this section create windows that have a "generic" look-and-feel:

- **ADJCT** – This procedure lets you interactively control the range and lower limit of the color table with the mouse, on those image displays that allow it. Directions for using ADJCT are written to the screen when the procedure is called.
- **C_EDIT** – This procedure allows the interactive creation of color tables based on the HLS or HSV color system. C_EDIT is similar to the COLOR_EDIT procedure, except that this implementation provides better control of HSV colors near zero percent saturation.
- **COLOR_EDIT** – This procedure creates color tables interactively using the HLS or the HSV color system. A temporary window is created containing a color wheel and bars for intensity and pixel value. The mouse is used to select the three color parameters and the corresponding pixel value. Color values are interpolated between selected pixel values. Graphs show-

ing the three color parameters as a function of value are displayed in the right half of the window.

- **COLOR_PALETTE** – This procedure displays the current color table in a separate window with color indices overwritten on the display. This is a handy procedure for finding out what color in the current color table is associated with a particular color index.
- **PALETTE** – This procedure displays an interactive window that lets you create color tables with RGB slider bars and allows good selection and control of each color index. It can interpolate in RGB space between color indices or edit a single color index.

Note All procedures are written in the PV-WAVE language and they all use the TVLCT procedure to load the color tables.

A

APPENDIX

Output Devices and Window Systems

This appendix discusses the output devices and window systems that are supported by PV-WAVE. The device and window system descriptions are arranged alphabetically in this appendix, beginning with *CGM Output* on page A-9. For a summary of the supported devices and window systems, see Table A-1 and Table A-2 on the next page.

In addition to describing the supported window systems and output devices, this appendix discusses the following topics:

- The general procedure for producing hardcopy output (see *Producing Hardcopy Output* on page A-3).
- Features common to window systems supported by PV-WAVE (see *Window System Features* on page A-6).
- Displaying images on monochrome devices (see *Displaying Images on Monochrome Devices* on page 148).

Table A-1: Supported Output Devices

Device Name	See Page	Description
NULL	N/A	No graphic output
CGM	A-10	Computer Graphics Metafile generator
HP	A-13	Hewlett-Packard Graphics Language (HPGL) plotters
LJ	A-20	Hewlett-Packard LaserJet 250 printer (VMS only)
PCL	A-23	Hewlett-Packard Printer Control Language (PCL)
PICT	A-27	PICT format for the Macintosh
PS	A-31	PostScript devices
QMS	A-48	QMS QUIC printer devices
REGIS	A-54	Regis graphics output devices
SIXEL	A-58	DEC printers
4510	A-61	Tektronix 4510 Rasterizer
TEK	A-65	Tektronix or compatible terminals
Z	A-99	Z-buffer device

Table A-2: Supported Window Systems

Device Name	See Page	Description
X	A-69	X Window System

Producing Hardcopy Output

The five steps you take to produce graphics output are the same no matter which hardcopy device driver you select. The steps are:

- ❑ Select the graphics output device. (This opens a file.)
- ❑ Configure the output device to your specifications.
- ❑ Issue the commands that will display your graphic output.
- ❑ Close the output file.
- ❑ Send the output file to the appropriate printer or plotter.

Each step is described in the following subsections. See the *PV-WAVE Command Language Overview* for additional examples of producing hardcopy output.

Selecting the Output Device with SET_PLOT

You select a graphics output device in PV-WAVE with the SET_-PLOT command. The command is:

SET_PLOT, '*string*'

where *string* can be any one of the following letter codes:

Code	Output Device
CGM	Computer Graphics Metafile format
HP	HPGL device
LJ	DEC LJ250 device
PCL	PCL device
PICT	PICT format for the Macintosh
PS	PostScript device
QMS	QMS QUIC device
SIXEL	SIXEL device
4510	Tektronix 4510 rasterizer device
TEK	Tektronix terminal

For example, this command selects the PostScript device:

```
SET_PLOT, 'ps'
```

Configuring the Output Device with DEVICE

Once the graphics output device has been selected, it is controlled or configured with the DEVICE command. The DEVICE command uses keywords to control the specific functions of each output device. Since each output device is unique, the number and names of keywords that are valid with the DEVICE command are different depending upon the device selected. For example, the DEVICE command for the PostScript device has 34 valid keywords, whereas the same DEVICE command for the Tektronix 4510 rasterizer has only 10 valid keywords.

For a list of keywords used with a particular device, see the device's description in this appendix.

If no DEVICE command is issued after the SET_PLOT command, then the device is configured with default values. To see the current configuration of any output device, issue the SET_PLOT command to select the device and then use the INFO command to obtain information about the device. For example, to learn the current configuration of the PostScript device, you would type the following:

```
SET_PLOT, 'ps'
INFO, /Device
```

Entering Graphics Commands for Output

After you have configured the output device to your specifications, you now enter appropriate graphics commands for the output you wish to produce. These are the same graphics commands you would issue if you were displaying output on a display screen. For example, any of the following graphics commands would be appropriate:

```
PLOT, mydata, Title='Available Light ' + $
   'Measurement'
```

```
TVSCL, my—image

PLOTS, x, y, /Normal

SHADE_SURF, peak, Shades=peak_colors

XYOUTS, 300, 450, 'Lost acreage', /Device

SURFACE, peak, Bottom=35, Color=248
```

Closing the Output File

Before the graphics output file can be sent to the printer or plotter it must be closed. For example, the following commands do not print a file, as you might expect:

```
SET_PLOT, 'ps'
PLOT, x, y
SPAWN, 'lpr wave.ps'
```

This attempt to print the file is premature. It fails because the file is still open within PV-WAVE.

Files are closed automatically when you exit PV-WAVE, but the best way to close an output file is to close it explicitly with the DEVICE command. After you enter the graphics commands for your desired graphics output, enter the following command to close the output file:

```
DEVICE, /Close
```

Sending the Output File to the Printer or Plotter

Once an output file has been closed, it can be sent to a printer or plotter in the normal way (e.g. with an `lpr` command in a UNIX environment or a `print` command in a VMS environment). But it is often more convenient to send a file to a printer or plotter without exiting PV-WAVE. The best way to do this is to use the "$" shortcut method for spawning an external process. For example, you could issue one or the other of the following two commands at the PV-WAVE prompt to send a file named `peak.ps` to a PostScript printer:

In UNIX:

```
WAVE> $lpr peak.ps
```

In VMS:

```
WAVE> $print/queue=post_q peak.ps
```

Note If your PostScript printer looks like it is printing something, but nothing comes out, you may have forgotten to close the file before you sent it to the printer.

Window System Features

PV-WAVE utilizes the X window system by creating and using one or more largely independent windows, each of which can be used for the display of graphics and/or images. One color table is shared among these windows. Up to 32 separate windows can be active at any time. Windows are referenced using their index, which is an integer value between 0 and 31.

"Dithering" or halftoning techniques are used to display images with multiple shades of gray on monochrome displays – displays that can only display white or black. This topic is discussed in the section, *Displaying Images on Monochrome Devices* on page 148.

Graphic and image output is always directed to the current window. When a window system is selected as the current PV-WAVE graphics device, the index number of the current window is found in the !D.Window system variable. This variable equals –1 if no window is open or selected. The WSET procedure is used to change the current window. WSHOW hides or displays a window. WDELETE deletes a window.

The WINDOW procedure creates a new window with a given index. If a window already exists with the same index, it is first deleted. The size, position, title, and number of colors may also be specified. If you access the display before creating the first window, PV-WAVE automatically creates a window with an index number of 0 and with the default attributes.

How Is Backing Store Handled?

One of the features that distinguishes various window systems is how they handle the issue of backing store. When part of a window that was previously not visible is exposed, there are two basic approaches that a window system can take. Some keep track of the current contents of all windows and automatically repair any damage to their visible regions (retained windows). This saved information is known as the backing store. Others simply report the damage to the program that created the window and leave repairing the visible region to the program (non-retained windows). There are convincing arguments for and against both approaches. It is generally more convenient for PV-WAVE if the window system handles this problem automatically, but this often comes at a performance penalty. The actual cost of retained windows varies between systems and depends partially on the application.

The X Window system does not by default keep track of window contents. Therefore, when a window on the display is obscured by another window, the contents of its obscured portion is lost. Re-exposing the window causes the X server to fill the missing data with the default background color for that window, and request the application to redraw the missing data. Applications can request a backing store for their windows, but servers are not required to provide it. Most current X servers do not provide backing store, and even those that do cannot necessarily provide it for all requesting windows. Therefore, requesting backing store from the server might help, but there is no certainty.

The PV-WAVE window system drivers allow you to control backing store using the *Retain* keyword to the DEVICE and WINDOW procedures. Using *Retain* with DEVICE allows you to set the default action for all windows, while using it with WINDOW lets you override the default for the new window. The possible values for this keyword are summarized in Table A-3, and are described in greater detail following the table.

Table A-3: Possible Values for the Retain Keyword

Value	Description
0	No backing store.
1	(The Default) The server or window system is requested to retain the window.
2	PV-WAVE should provide a backing pixmap and handle the backing store directly (X Window System only).

0 – A value of 0 specifies that no backing store is kept. In this case, exposing a previously obscured window leaves the missing portion of the window blank. Although this behavior can be inconvenient, it usually has the highest performance because there is no need to keep a copy of the window contents.

1 – (The Default) Setting the *Retain* keyword to 1 causes PV-WAVE to request that a backing store be maintained. If the window system decides to accept the request, it automatically repairs the missing portions when the window is exposed. X Windows may or may not, depending on the capabilities of the server and the resources available to it.

2 – Specifies that PV-WAVE should keep a backing store for the window itself, and repair any window damage when the window system requests it. This option exists mainly for the X Window System. Under X, a pixmap (off-screen display memory) the same size as the window is created at the same time the window is created, and all graphics operations sent to the window are also sent to the pixmap. When the server requests PV-WAVE to repair freshly exposed windows, this pixmap is used to fill in the missing contents. Pixmaps are a precious resource in the X server, so backing pixmaps should only be requested for windows with contents that must absolutely be preserved.

If the type of backing store to use is not explicitly specified using the *Retain* keyword, PV-WAVE assumes Option 1 and requests the window system to keep a backing store.

CGM Output

Computer Graphics Metafile, or CGM, is a standard for storing graphics output. To direct graphics output from PV-WAVE to a CGM file, enter the command:

```
SET_PLOT, 'CGM'
```

This causes PV-WAVE to use the CGM driver for producing graphical output, including line plots, contour plots, surface plots, and raster images. Once the CGM driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in *Controlling CGM Output with DEVICE Keywords* on page A-10. The default settings for the CGM driver are listed in Table A-4.

Table A-4: Default CGM Driver Settings

Setting	Default Value
Output file name	wave.cgm
File status	Closed
Metafile type	Clear Text
Number of colors in color table	254
Color table offset	1
Clip to VDC range	ON
Horizontal offset	0 Coordinates
Vertical offset	0 Coordinates
Scale factor	None
Standard Cell Array	Standard

The CGM driver supports clear text and binary CGM output. To see the driver's current settings, enter:

```
INFO, /Device
```

Tip If you want to send a CGM file to a hardcopy printer, and you want to modify the color table, it is a good idea to modify the color table before sending CGM graphics output to the file. For more information about modifying colortables, see *Experimenting with Different Color Tables* on page 310.

Controlling CGM Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the CGM driver:

Clip – Specifies that all coordinates will be reduced to fit within the VDC coordinate range. This keyword is primarily used with the *Xoffset* and *Yoffset* keywords to protect the metafile from containing coordinates which are outside the range of the VDC system.

Close – Closes the current CGM output file.

Colors – Specifies the number of colors in the color table. Be sure to specify the number of colors in the color table before any graphics output is written to the file. The default number of colors is 254, since the CGM standard "reserves" color indices 0 and 255.

Ct_Offset – Specifies the location (or offset) of the first element in the color table. As with the *Colors* keyword, this value should be specified before any graphics output is written to the file. The default value is 1, since the CGM standard "reserves" color index 0.

Filename – Specifies the name of the CGM output file to be opened. If the keyword is not specified, the default name `wave.cgm` is used.

Metafile_Type – Specifies the metafile type. Options are `clear_text` and `binary`. The default value is `clear_text`. The `binary` type is machine-specific and is more difficult to transfer.

Scale_Factor – Includes a floating-point value in the metafile for the Scale Mode Metric command. This allows for CGM interpreters to scale the graphics output appropriately for "exact sizing". The metric scale-factor represents the distance (in millimeters) in the displayed picture – this corresponds to one VDC unit.

Std_Cell_Array – Specifies that images created with the binary output (CELL_ARRAY commands) are compliant with the CGM standard. *Std_Cell_Array* is enabled by default. Disabling this keyword may result in some non-standard cell array commands for odd-sized images. The non-standard cell array option is provided for downward compatibility.

Xoffset – Specifies the number of horizontal coordinate units to offset the graphics output. *Xoffset* is specified by a positive normalized coordinate in the range {0.0...1.0}.

Yoffset – Specifies the number of vertical coordinate units to offset the graphics output. *Yoffset* is specified by a positive normalized coordinate in the range {0.0...1.0}.

Using the CGM Driver

The CGM output file is automatically opened when the CGM driver is selected and PV-WAVE commands are issued that result in graphics output; there is no explicit OPEN command. If more than one CGM file is to be created, use the following command sequence:

```
SET_PLOT, 'CGM'
```
Select the CGM driver for graphics output.

```
DEVICE, Filename='filename1'
```
Open the first file.

```
...
```
Graphics output commands here.

```
DEVICE, /Close
```
Close the first file.

```
DEVICE, Filename='filename2'
```
Open another file.

```
...
```
Graphics output commands here.

```
DEVICE, /Close
```
Close the second file, etc.

Note also that color table changes should be made every time a new CGM file is opened. In the absence of changes, the new CGM file contains the default color table, rather than the current color table.

Using Color with CGM Output

The CGM standard reserves two color indices (0 and 255) from a possible 256 colors. Therefore, PV-WAVE is limited to 254 colors. Since color index 0 is reserved by CGM, the PV-WAVE color tables are offset by one.

For example, to create and load a color table with four elements, black, red, green, and blue:

```
TVLCT, [0,255,0,0], [0,0,255,0], [0,0,0,255]
```

Drawing text or graphics with a color index of 1 results in black, 2 in red, 3 in green, and 4 in blue.

Changing the Image Background Color

Images that are displayed with a black background on a monitor frequently look better in hardcopy form if the background is changed to white. This is easily accomplished with CGM output by issuing the statement, where `A` is the image variable:

```
A(WHERE(A EQ 0B)) = 255B
```

Changing the CGM Background Color

To set the background color for a CGM metafile, use the ERASE command with a color index. The index is used to define the background color. For example:

```
ERASE, 150
```

Binary CGM Output for VAX/VMS Machines

Binary CGM output is written to a fixed-length file with a record length of 512 bytes. To display the CGM metafile with Visual Numerics' VAX/VMS Extended Metafile System, provide the CGM metafile filename in quotes followed by format 2, record 512.

For example,

```
CGM> set metafile "cgmfile" format 2 record
   512

CGM> interpret
```

HPGL Output

HPGL (Hewlett-Packard Graphics Language) is a plotter control language used to produce graphics on a wide family of pen plotters.

To use HPGL as the current graphics device, issue the PV-WAVE command:

```
SET_PLOT, 'HP'
```

This causes PV-WAVE to use HPGL for producing graphical output. Once the HPGL driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in *Controlling HPGL Output with DEVICE Keywords* on page A-16. The default settings for the HPGL driver are given in Table A-5.

Table A-5: Default HPGL Driver Settings

Setting	Default Value
Output file name	wave.hp
Orientation	portrait
Erase	no action
Polygon fill	software

Table A-5: Default HPGL Driver Settings

Setting	Default Value
Turn plotter logically on/off	no
Specify xon/xoff flow control	yes
Horizontal offset	.3175 cm (.125 in.)
Vertical offset	11.43 cm (4.5 in.)
Width	17.78 cm (7 in.)
Height	12.7 cm (5 in.)

Use the statement:

```
INFO, /Device
```

to view the current HPGL driver settings.

Supported Features of HPGL

PV-WAVE is able to produce a wide variety of graphical output using HPGL.

Here is a list of what is supported:

- All types of vector graphics can be generated, including line plots, contours, surfaces, etc.
- HPGL plotters can draw lines in different colors selected from the pen carousel. It should be noted that color tables are not used with HPGL. Instead, each color index refers directly to one of the pens in the carousel.
- Some HPGL plotters can do polygon filling in hardware. Others can rely on the software polygon filling provided by PV-WAVE.
- It is possible to generate graphics using the hardware-generated text characters, although such characters do not give much improvement over the standard software fonts. To use hardware characters, set the !P.Font system variable to zero, or set the *Font* keyword to the plotting routines to zero. For more information on fonts, see Chapter 9, *Software Fonts*.

Here is a list of what is not supported:

- Since HPGL is designed to drive pen plotters, it does not support the output of raster images. Therefore the TV and TVSCL procedures do not work with HPGL.
- Since pen plotters are not interactive devices, they cannot support such operations as cursors and windows.

Specifying Linestyles in HPGL Output

The *Linestyle* graphics keyword allows you to specify any of 6 linestyles. This keyword is documented Chapter 3 *Graphics and Plotting Keywords*, in the *PV-WAVE Reference.*

HPGL is not able to support all of the standard PV-WAVE linestyles. Table A-6 summarizes the differences between the normal PV-WAVE linestyles and those supported by HPGL.

Table A-6: HPGL Supported Linestyles

Index	Normal Line Style	HPGL Style
0	Solid	same
1	Dotted	same
2	Dashed	same
3	Dash Dot	The relative size of the dash and dot are different.
4	Dash Dot Dot Dot	Dash Dot Dot
5	Long Dashes	same

Tip If your HPGL plotter is connected to an HP-IB interface, you must run PV-WAVE's HPGL output through a filter before you can plot it. The following UNIX command accomplishes this task:

```
tr -d '/012' <wave.hp  >newwave.hp
```

Controlling HPGL Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the HPGL driver:

Close_File – PV-WAVE creates, opens and writes a file containing the generated graphics output. The *Close_File* keyword outputs any buffered commands and closes the file.

Caution Under UNIX, if you close the output file with the *Close_File* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

See the discussion of printing output files in the section, *Producing Hardcopy Output* on page A-3, for more information on this topic.

Eject – In order to perform an erase operation on a plotter, it is necessary to remove the current sheet of paper and load a fresh sheet. The ability of various plotters to do this varies, so the *Eject* keyword allows you to specify what should be done. Table A-7 gives the possible values.

Table A-7: Possible Values for the Eject Keyword

Value	Meaning
0	(Default) Do nothing. Note that this is likely to cause one page to plot over the previous one, so you should limit yourself to one page of output per file.
1	Use the sheet feeder to load the next page.
2	Put the plotter off-line at the beginning of each page, except the first.

Many HPGL plotters lack a sheet feeder, and require you to load the next page manually. Therefore, the default action is for PV-WAVE to not issue any page eject instructions. In this case, you must restrict yourself to generating only a single plot at a time.

If your plotter has a sheet feeder, you need to issue the command:

```
DEVICE, /Eject
```

to tell PV-WAVE that it should use the sheet feeder instead of placing the plotter off-line. If your plotter does not have a sheet feeder, but it does understand the HPGL NR command, use the command:

```
DEVICE, Eject=2
```

to place the plotter off-line at the start of every plot except the first one. This causes the plotter to wait between plots for you to replace the paper. When you put the plotter back on-line, the graphics commands for the new page are executed by the plotter. Consult the programming manual for your plotter to determine if this instruction is provided.

Filename – By default all generated output is sent to a file named `wave.hp`. The *Filename* keyword can be used to change this default. If you specify a filename, the following occurs:

- If the file is already open (as happens if plotting commands have been directed to the file since the call to SET_PLOT), then the file is completed and closed as if *Close_File* had been specified.
- The specified file is opened for subsequent graphics output.

Inches – By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, these keywords are taken to be in inches instead.

Landscape – PV-WAVE normally generates plots with portrait orientation (the X axis is along the short dimension of the page). If *Landscape* is present, landscape orientation (the X axis is along the long dimension of the page) is used instead.

Output – Specifies a scalar string that is sent directly to the graphics output file without any processing, allowing you to send arbitrary commands to the file. Since PV-WAVE does not examine the string, it is your responsibility to ensure that the string is correct for the target device.

Polyfill – Some plotters (e.g., HP7550A) can perform polygon filling in hardware, while others (e.g., HP7475) cannot. PV-WAVE therefore assumes that the plotter cannot, and generates all polygon operations in software using line drawing. Specifying a nonzero value for the *Polyfill* keyword causes PV-WAVE to use the hardware polygon filling. Setting it to zero reverts to software filling.

Different implementations of HPGL plotters may have different limits for the number of vertices that can be specified for a polygon region before the plotter runs out of internal memory. Since this limit can vary, the HPGL driver cannot check for calls to *Polyfill* that specify too many points. Therefore, it is possible for you to produce HPGL output that causes an error when sent to the plotter. To avoid this situation, minimize the number of points used. On the HP7550A, the limit is about 127 points. If you do generate output that exceeds the limit imposed by your plotter, you have to break that polygon filling operation into multiple smaller operations.

Plotter_On_Off – There are some configurations in which an HPGL plotter is connected between the computer and a terminal. In this mode (known as eavesdrop mode), the plotter ignores everything it is sent and passes it through to the terminal – the plotter is logically off. This state continues until an escape sequence is sent that turns the plotter logically on. At this point the plotter interprets and executes all input as HPGL commands. Another escape sequence is sent at the end of the HPGL commands to return the plotter to the logically off state.

Most configurations do not use eavesdrop mode, and the plotter is always logically on. However, if you are using this style of connection, you must use *Plotter_On_Off* to instruct PV-WAVE to generate the necessary on/off commands. If present and nonzero, *Plotter_On_Off* causes each output page to be bracketed by device control commands that turn the plotter logically on and off. Specifying a value of zero stops the issuing of such commands. You should only use this keyword before any output has been generated.

Portrait – If *Portrait* is present, PV-WAVE generates plots using portrait orientation, the default.

Xoffset – Specifies the X position on the page of the lower-left corner of output generated by PV-WAVE. *Xoffset* is specified in centimeters unless *Inches* is specified.

Xon_Xoff – If present and nonzero, *Xon_Xoff* causes each output page to start with device control commands that instruct the plotter to obey xon/xoff (^S/^Q) style flow control. Specifying a value of zero stops the issuing of such commands. You should only use this keyword before any output has been generated. Such handshaking is the default. To turn it off, use the command:

```
DEVICE, Xon_Xoff=0
```

Often, it is not necessary to tell the plotter to obey flow control because the printing facilities on the system handle such details for you, but it is usually harmless.

Xsize – Specifies the width of output PV-WAVE generates. *Xsize* is specified in centimeters unless *Inches* is specified.

Yoffset – Specifies the Y position on the page of the lower-left corner of output generated by PV-WAVE. *Yoffset* is specified in centimeters unless *Inches* is specified.

Ysize – Specifies the height of output generated by PV-WAVE. *Ysize* is specified in centimeters unless *Inches* is specified.

LJ-250 Output

The LJ-250 driver is used to produce output on a Hewlett-Packard LaserJet 250 printer. The LJ-250 printer option is only available for PV-WAVE running on VAX/VMS systems.

To direct graphics output to an LJ-250 printer, issue the command:

```
SET_PLOT, 'LJ'
```

This causes PV-WAVE to use the LJ-250 driver for producing graphical output. Once the LJ-250 driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in *Controlling LJ-250 Output with DEVICE Keywords* on page A-21. The default settings for the LJ-250 driver are given in Table A-8.

Table A-8: Default LJ-250 Driver Settings

Setting	Default Value
Output file name	wave.lj
Mode	Portrait
Dither Method	Floyd-Steinberg
Resolution	180 dpi
Horizontal Offset	1.27 cm (1/2 in.)
Vertical Offset	2.54 cm (1 in.)
Width	17.78 cm (7 in.)
Height	12.7 cm (5 in.)
# of planes	1 (2 color)

Use `INFO, /Device` to view the driver's current settings.

Tip When you print files to an LJ-250 printer from a UNIX system, you may need to specify the -v option in the print command. This option indicates that a raster image is being transmitted. For example:

```
lpr -Ppcl_series_q -v wave.lj
```

From a VMS system, use the /PASSALL parameter. For example:

```
PRINT /QUEUE=PCL_SERIES_Q /PASSALL WAVE.LJ
```

Controlling LJ-250 Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the LJ-250 driver:

Close_File – PV-WAVE creates, opens and writes a file containing the generated graphics output. The *Close_File* keyword outputs any buffered commands and closes the file.

See the discussion of printing output files in the section, *Producing Hardcopy Output* on page A-3, for more information on this topic.

Depth – Selects the number of color planes available for use. A value of 1 (the default) sets the LJ-250 up for 2 colors, a value of 2 sets up for 4 colors, a value of 3 sets up for 8 colors, and a value of 4 sets up for 16 colors.

Filename – By default all generated output is sent to a file named `wave.lj`. The *Filename* keyword can be used to change this default. If you specify a filename, the following occurs:

- If the file is already open (as happens if graphics have been directed to the file since the call to SET_PLOT), then the file is completed and closed as if *Close_File* had been specified.
- The specified file is opened for subsequent graphics output.

Floyd – If present and nonzero, selects the Floyd-Steinberg method of dithering. For information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Inches – By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, they are taken to be in inches instead.

Landscape – PV-WAVE normally generates plots with portrait orientation (the X axis is along the short dimension of the page).

If *Landscape* is present, landscape orientation (the X axis is along the long dimension of the page) is used instead.

Ordered – Selects the Ordered method of dithering when displaying images on a monochrome display. For information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Pixels – By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Pixels* is present and nonzero, they are taken to be in pixels instead. Note that the selected resolution will determine how large a region is actually written on the page.

Portrait – If *Portrait* is present, PV-WAVE will generate plots using portrait orientation, the default.

Resolution – The resolution at which the LJ-250 printer will print. LJ-250 supports resolutions of 90 and 180 dots per inch. The default is 180 dpi. Lower resolution gives smaller output files, while higher resolution gives superior quality.

Threshold – Changes the dithering method to the threshold dithering algorithm. For information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Xoffset – Specifies the X position on the page of the lower-left corner of output generated by PV-WAVE. *Xoffset* is specified in centimeters unless *Inches* is specified.

Xsize – Specifies the width of output PV-WAVE generates. *Xsize* is specified in centimeters unless *Inches* is specified.

Yoffset – Specifies the Y position on the page of the lower-left corner of output generated by PV-WAVE. *Yoffset* is specified in centimeters unless *Inches* is specified.

Ysize – Specifies the height of output that PV-WAVE generates. *Ysize* is specified in centimeters unless *Inches* is specified.

LJ-250 Image Background Color

Images that are displayed with a black background on a monitor frequently look better if the background is changed to white when printed with the LJ-250. This is easily done with the following statement, where A is the image variable:

```
A(WHERE(A EQ 0B)) = 255B
```

PCL Output

PCL (Printer Control Language) is used by Hewlett-Packard laser and ink jet printers to produce graphics output. This driver does not support the use of color.

To direct graphics output to a PCL file, issue the command:

```
SET_PLOT, 'PCL'
```

This causes PV-WAVE to use the PCL driver for producing graphical output. Once the PCL driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in *Controlling PCL Output with DEVICE Keywords* on page A-24. The default settings for the PCL driver are given in Table A-9.

Table A-9: Default PCL Driver Settings

Setting	Default Value
Output file name	wave.pcl
Mode	Portrait
Optimization Level	0 (None)
Dither Method	Floyd-Steinberg
Resolution	300 dpi
Horizontal Offset	1.27 cm (1/2 in.)
Vertical Offset	2.54 cm (1 in.)
Width	17.78 cm (7 in.)
Height	12.7 cm (5 in.)

Note By default, PCL writes black on white paper. The default color index when PCL is selected is 0. If the color index is set to 255 when PCL is selected, the result is white writing on white, which is invisible on white paper. Color tables are not used with PCL.

Use `INFO, /Device` to view the driver's current settings.

Tip When you print files to an PCL device from a UNIX system, you may need to specify the `-v` option in the print command. This option indicates that a raster image is being transmitted. For example:

```
lpr -Plj250_q -v wave.pcl
```

From a VMS system, use the `/PASSALL` parameter. For example:

```
PRINT /QUEUE=LJ250_Q /PASSALL WAVE.PCL
```

Controlling PCL Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the PCL driver:

Close_File – PV-WAVE creates, opens and writes a file containing the generated graphics output. The *Close_File* keyword outputs any buffered commands and closes the file.

Caution Under UNIX, if you close the output file with the *Close_File* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

See the discussion on printing output files in the section, *Producing Hardcopy Output* on page A-3, for more information on this topic.

Filename – By default all generated output is sent to a file named `wave.pcl`. The *Filename* keyword can be used to change this default. If you specify a filename, the following occurs:

- If the file is already open (as happens if graphics have been directed to the file since the call to SET_PLOT), then the file is completed and closed as if *Close_File* had been specified.
- The specified file is opened for subsequent graphics output.

Floyd – If present and nonzero, selects the Floyd-Steinberg method of dithering. For information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Inches – By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, they are taken to be in inches instead.

Landscape – PV-WAVE normally generates plots with portrait orientation (the X axis is along the short dimension of the page). If *Landscape* is present, landscape orientation (the X axis is along the long dimension of the page) is used instead.

Optimize – It is desirable, though not always possible, to compress the size of the output file. Such optimization reduces the size of the output file, and improves I/O speed to the printer. There are three levels of optimization:

- **0** – No optimization is performed. This is the default because it will work with any PCL device. However, users of devices which can support optimization should use one of the other optimization levels.
- **1** – Optimization is performed using PCL optimization primitives. This gives the best output compression and printing speed. Unfortunately, not all PCL devices support it. On those that can't, the result will be garbage printed on the page.

  The required sequences are: `<ESC>*b0M` (Select full graphics mode), `<ESC>*b1M` (Select compacted graphics mode 1), and `<ESC>*b2M` (Select compacted graphics mode 2). To determine if your printer supports the required escape sequences, consult the programmers manual for the device.

  The HP LaserJet II does not support this optimization level. The DeskJet PLUS does.

- 2 – PV-WAVE attempts to optimize the output by explicitly moving the left margin and then outputting non-blank sections of the page. This is primarily intended for use with the LaserJet II, which does not support optimization level 1.

Note Level 2 optimization can be very slow on some devices (such as the DeskJet PLUS). On such devices, it is best to avoid this optimization level.

Ordered – Selects the Ordered Dither method of dithering when displaying images on a monochrome display. For information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Pixels – By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Pixels* is present and nonzero, they are taken to be in pixels instead. Note that the selected resolution will determine how large a region is actually written on the page.

Portrait – If *Portrait* is present, PV-WAVE will generate plots using portrait orientation, the default.

Resolution – The resolution at which the PCL printer will print. PCL supports resolutions of 75, 100, 150, and 300 dots per inch. The default is 300 dpi. Lower resolution gives smaller output files, while higher resolution gives superior quality.

Threshold – Specifies use of the threshold dithering algorithm. For information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Xoffset – Specifies the X position on the page of the lower-left corner of output generated by PV-WAVE. *Xoffset* is specified in centimeters unless *Inches* or *Pixels* is specified.

Xsize – Specifies the width of the PV-WAVE output. *Xsize* is specified in centimeters unless *Inches* or *Pixels* is specified.

Yoffset – Specifies the Y position on the page of the lower-left corner of output generated by PV-WAVE. *Yoffset* is specified in centimeters unless *Inches* or *Pixels* is specified.

Ysize— Specifies the height of the PV-WAVE output. *Ysize* is specified in centimeters unless *Inches* or *Pixels* is specified.

PCL Image Background Color

Images that are displayed with a black background on a monitor frequently look better if the background is changed to white when displayed with PCL. This is easily done with the following statement, where A is the image variable:

```
A(WHERE(A EQ 0B)) = 255B
```

PICT Output

The PICT driver produces PV-WAVE output that can be transferred to a Macintosh computer. To direct graphics output to a PICT file, enter the command:

```
SET_PLOT, 'PICT'
```

This causes PV-WAVE to use the PICT driver for producing graphical output. Once the PICT driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in the next section, *Controlling PICT Output with DEVICE Keywords.*

Table A-10: Default PICT Driver Settings

Setting	Default Value
Output file name	wave.pict
Bitreverse	0
Dither Method	Ordered
Pen Pattern	Black
X and Y Pen Size	1
X, Y Screen Size	512, 342
Font	Geneva
Text size	12 point

Use `INFO, /Device` to view the driver's current settings.

Controlling PICT Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the PICT driver:

Bitreverse – When this keyword is specified and nonzero, the output image appears as white on a black background. When disabled, as it is by default, the image appears as black on a white background. This feature can be useful, for example, if your graphics contain large dark or black areas that you do not want to show up in the output.

Close – PV-WAVE creates, opens and writes a file containing the generated graphics output. The *Close* keyword outputs any buffered commands and closes the file.

Caution Under UNIX, if you close the output file with the *Close* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

See the discussion of printing output files in the section, *Producing Hardcopy Output* on page A-3, for more information on this topic.

Filename – By default all generated output is sent to a file named `wave.pict`. The *Filename* keyword can be used to change this default. When you specify a filename, the following occurs:

- If the file is already open (as happens if graphics have been directed to the file since the call to SET_PLOT), then the file is completed and closed as if *Close* had been specified.
- The specified file is opened for subsequent graphics output.

Floyd – If present and nonzero, selects the Floyd-Steinberg dithering method. For more information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Ordered – (Default) Selects the Ordered Dither method of dithering when displaying images on a monochrome display. For more information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Penpat – Converts PV-WAVE linestyles to PICT pen patterns, as shown in Table A-11. Specify one of five pen patterns: Black, Dark Gray, Gray, Light Gray, or White. The default pen pattern is Black.

For example:

```
DEVICE, Penpat='Gray'
```

Table A-11: Mapping of PV-WAVE Linestyles to PICT

PV-WAVE Linestyles	PICT Pen Pattern
Solid	Black
Dotted	Light Gray
Dashed	Gray
Dash Dot	Dark Gray
Dash Dot Dot Dot	Dark Gray
Long Dashes	Dark Gray

Penxsize – Specifies a value for the width of the pen, measured in pixels. The default is 1. Increasing or decreasing this value increases or decreases the thickness of lines.

Penysize – Specifies a value for the height of the pen, measured in pixels. The default is 1. Increasing or decreasing this value increases or decreases the thickness of lines. For example:

```
DEVICE, Penxsize='2', Penysize='2'
```

Screenx – Specifies the width of the graphics display screen in pixels. The default is 512 pixels.

Screeny – Specifies the height of the graphics display screen in pixels. The default is 342 pixels. For example:

```
DEVICE, Screenx=720, Screeny=576
```

Textfont — Specifies the name of a Macintosh font to be used in the output. Fonts include Systemfont, Applfont, Newyork, Geneva, Monaco, Venice, London, Helvetica, Courier, Symbol, Taliesin, Athens, Sanfran, Toronto, Cairo, Losangeles, and Times. The default font is Geneva. For example:

```
DEVICE, Textfont='Times'
```

Textsize — Specifies a value for the font size. Font size is measured in points, a typesetting measurement equal to 1/72 inch. Increasing the font size increases the size of text in the output. The default is 12, a text size equivalent to 1/6 inch. For example:

```
DEVICE, Textsize=9
```

Threshold — Specifies use of the threshold dithering algorithm — the simplest method. For more information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

PostScript Output

PostScript is a programming language designed to convey a description of virtually any desired page containing text and graphics. It is widely available on laser printers and typesetters.

To direct graphics output to a PostScript file, enter the command:

```
SET_PLOT, 'PS'
```

This causes PV-WAVE to use the PostScript driver for producing graphical output. Once the PostScript driver is enabled via SET_PLOT, the DEVICE procedure controls its actions, as described in *Controlling PostScript Output with DEVICE Keywords* on page A-32. The default settings for the PostScript driver are given in Table A-12.

Table A-12: Default PostScript Driver Settings

Setting	Default Value
Output file name	wave.ps
Mode	portrait, non-encapsulated, no color
Horizontal offset	1.905 cm (3/4 in.)
Vertical offset	12.7 cm (5 in.)
Width	17.78 cm (7 in.)
Height	12.7 cm (5 in.)
Scale factor	1.0
Font size	12 pt.
Font	Helvetica
bits / image pixel	4

Note Unlike monitors where white is the most visible color, PostScript writes black on white paper. Setting the output color index to 0, the default when PostScript output is selected, writes black. A color index of 255 writes white which is invisible on white paper. Color tables are not used with PostScript unless the color mode has been enabled using the DEVICE procedure. See below for information on using PV-WAVE with color PostScript.

Note All PostScript printers impose a limit on the number of vertices a polygon may contain. This limit (750 vertices for most printers) is checked, and an error message is printed if it is exceeded.

Use `INFO, /Device` to view the driver's current settings.

Controlling PostScript Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the PostScript driver:

Note PostScript fonts are selected using DEVICE keywords. For example:

```
DEVICE, /Helvetica, /Bold
```

See Table A-13 on page A-37 for a complete list of these keywords.

Bits_Per_Pixel – PV-WAVE is capable of producing PostScript images with 1, 2, 4, or 8 bits per pixel. Using more bits per pixel gives higher resolution at the cost of generating larger files. *Bits_Per_Pixel* is used to specify the number of bits to use. The default number of bits is four.

The Apple Laserwriter is capable of only 32 different shades of gray (which can be represented by 5 bits). Thus, specifying 8 bits per pixel does not give 256 levels of gray as might be expected, only 32, at a cost of sending twice the number of bits to the printer. Often, 4 bits (16 levels of gray) will give acceptable results with a large savings in file size.

Close_File – PV-WAVE creates, opens and writes a file containing the generated graphics output. The *Close_File* keyword outputs any buffered commands and closes the file.

Caution Under UNIX, if you close the output file with the *Close_File* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

Color – Enables color PostScript output if present and nonzero. For details on using color PostScript devices see *Using Color PostScript Devices* on page A-39.

Encapsulated – Specifies that the PostScript produced by PV-WAVE is to be included in another PostScript document, such as one produced by $\mathrm{T_EX}$, $\mathrm{LAT_EX}$, FrameMaker, Microsoft Word, or Ventura Publisher.

By default PV-WAVE assumes that its PostScript-generated output will be sent directly to a printer. It therefore includes PostScript commands to center the plot on the page and to eject the page from the printer. These commands are undesirable if the output is going to be inserted into the middle of another PostScript document. If *Encapsulated* is present and nonzero, PV-WAVE does not generate these commands.

PV-WAVE follows the standard PostScript convention for encapsulated files. It assumes the standard PostScript scaling is in effect (72 points per inch). In addition, it declares the size, or bounding box, of the plotting region at the top of the output file. This size is determined when the output file is opened (when the first graphics command is given) by multiplying the size of the plotting region (as specified with the *Xsize* and *Ysize* keywords) by the current scale factor (as specified by the *Scale_Factor* keyword). Changing the size of the plotting region or scale factor once any graphics have been output will not be reflected in the declared bounding box, and will confuse programs that attempt to import the resulting graphics. Therefore, when generating encapsulated PostScript do not change the plot region size of scaling factor after issuing graphics commands. If you want to change these parameters, use the *Filename* keyword to start a new file.

To explicitly disable the encapsulated PostScript option, use `DEVICE, encaps=0`.

Note You cannot reposition encapsulated output (including EPSI output) via *Xoffset* or *Yoffset* keywords. This allows full positioning control for the including product, such as MS Word.

Epsi – Produces an output file in encapsulated PostScript interchange (EPSI) format. When the *Epsi* keyword is present and nonzero, the *Encapsulated* keyword is automatically assumed. For detailed information on the *Encapsulated* keyword, see the description given previously in this section.

Like normal encapsulated PostScript files, EPSI files can be imported into many desktop publishing and word processing systems; however, EPSI format provides a previewing capability that allows an approximation of the printed PostScript output to be displayed on the screen. Previewed EPSI graphics are displayed in monochrome only.

Note You can print EPSI files directly, but only when the file is generated in portrait mode (see the description of the *Portrait* keyword below).

Filename – By default all generated output is sent to a file named `wave.ps`. The *Filename* keyword can be used to change this default. If you specify a filename, one or both of the following actions occur:

- If the file is already open (as happens if plotting commands have been directed to the file since the call to SET_PLOT), then the file is completed and closed as if *Close_File* had been specified.
- The specified file is opened for subsequent graphics output.

Font_Size – Specifies the default height used for displayed text. *Font_Size* is given in points (a common unit of measure used in typesetting). The default size is 12 point text.

Inches – By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, they are taken to be in inches instead.

Landscape – PV-WAVE normally generates plots with portrait orientation (with the X axis along the short dimension of the page).

If *Landscape* is present, landscape orientation (with the X axis is along the long dimension of the page) is used instead.

Tip In both portrait and landscape mode, the X offset is measured as a displacement along the page's short dimension, and Y offset is measured as a displacement along the page's long dimension. This may cause you some confusion when you are trying to orient graphics on a landscape page. The following figure demonstrates the correct way to specify offset for PostScript's landscape mode:

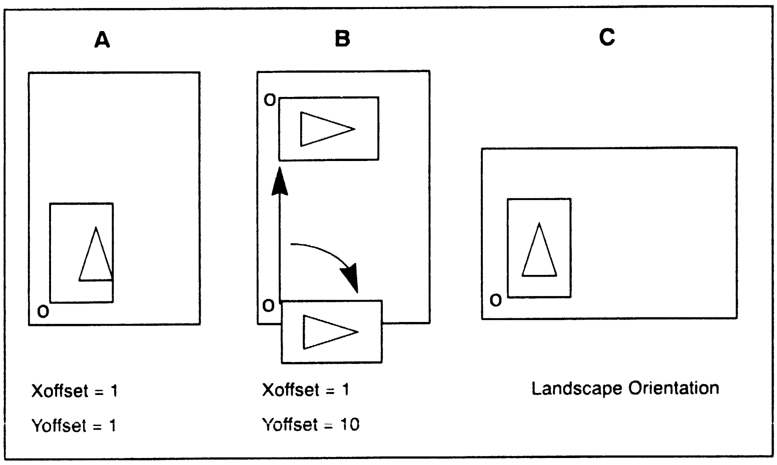


Figure A-1 This figure illustrates the way in which the *Xoffset* and *Yoffset* keywords are used to position graphics on a page in both portrait and landscape orientation. Assume that the page size is 8.5 x 11 inches. Portrait orientation is shown in A. The graphic is oriented with its origin (o) in the lower-left corner of the page. The X offset is 1 inch and the Y offset is 1 inch. The graphic's X axis falls along the page's short dimension. The same graphic shown in B is, in effect, rotated 90 degrees clockwise and offset 10 inches vertically (Y offset) to produce the landscape orientation shown in C. The important point to note is that the Y offset needed to position the graphic as shown in C is 10 inches, not 1 inch as you might expect.

Output – Specifies a scalar string that is sent directly to the graphics output file without any processing, allowing you to send arbitrary commands to the file. Since PV-WAVE does not examine the string, it is your responsibility to ensure that the string will be recognized by the target device.

Portrait – If *Portrait* is present, PV-WAVE generates plots using portrait orientation (the default).

Scale_Factor – Specifies a scale factor applied to the entire plot. Its default value is 1.0, allowing output to appear at its normal size. *Scale_Factor* magnifies or shrinks the resulting output.

User_Font – A scalar string that gives the name of a PostScript font to use. This font name must be specified exactly as the PostScript interpreter expects it, with the correct case and spelling. *User_Font* lets you use fonts not known to PV-WAVE.

Xoffset – Specifies the X position on the page of the lower-left corner of output generated by PV-WAVE. By default, *Xoffset* is specified in centimeters unless the *Inches* keyword is specified. *Scale_Factor* does not affect the value of *Xoffset*.

Xsize – Specifies the width of output. By default, *Xsize* is specified in centimeters unless the *Inches* keyword is specified *Scale_Factor* modifies the value of *Xsize*. Hence, the statement:

```
DEVICE, /Inches, Xsize=7.0, Scale_Factor=0.5
```

results in a real width of 3.5 inches.

Yoffset – Specifies the Y position on the page of the lower-left corner of output generated by PV-WAVE. By default, *Yoffset* is specified in centimeters unless the *Inches* keyword is specified. *Scale_Factor* does not affect the value of *Yoffset*.

Ysize – Specifies the height of output. By default, *Ysize* is specified in centimeters unless the *Inches* keyword is specified. *Scale_Factor* modifies the value of *Ysize*. Hence, the statement:

```
DEVICE, /Inches, Ysize=5.0, Scale_Factor=0.5
```

results in a real width of 2.5 inches.

Using PostScript Fonts

By default, PV-WAVE uses software characters for annotating plots (i.e., !P.Font is –1). These characters are of good quality and are extremely flexible. However, if this flexibility is not required, higher quality characters are available via PostScript fonts. In order to get PV-WAVE to use the PostScript fonts, do *one* of the following:

- Set !P.Font to 0 by entering:

```
!P.Font = 0
```

- Use the *Font* keyword with the plotting and graphics procedures to specify font 0. For example:

```
PLOT, temps, Title='Average Temp', Font=0
```

The default PostScript font is 12 point Helvetica. To change this font, use the DEVICE procedure keywords, as shown in Table A-13. Note that not all fonts may be available on a particular device.

Note When generating three-dimensional plots, it is best to use the software fonts, because PV-WAVE can draw them in perspective with the rest of the plot. For details on software fonts, see Chapter 9, *Software Fonts.*

Table A-13: PostScript Fonts

PostScript Font	DEVICE Keywords
Courier	/Courier
Courier Bold	/Courier, /Bold
Courier Oblique	/Courier, /Oblique
Courier Bold Oblique	/Courier, /Bold, /Oblique
Helvetica	/Helvetica
Helvetica Bold	/Helvetica, /Bold
Helvetica Oblique	/Helvetica, /Oblique
Helvetica Bold Oblique	/Helvetica, /Bold, /Oblique
Helvetica Narrow	/Helvetica, /Narrow

Table A-13: PostScript Fonts

PostScript Font	DEVICE Keywords
Helvetica Narrow Bold	/Helvetica, /Narrow, /Bold
Helvetica Narrow Oblique	/Helvetica, /Narrow, /Oblique
Helvetica Narrow Bold Oblique	/Helvetica, /Narrow, /Bold, /Oblique
ITC Avant Garde Gothic Book	/Avantgarde, /Book
ITC Avant Garde Gothic Book Oblique	/Avantgarde, /Book, /Oblique
ITC Avant Garde Gothic Demi	/Avantgarde, /Demi
ITC Avant Garde Gothic Demi Oblique	/Avantgarde, /Demi, /Oblique
ITC Bookman Demi	/Bkman, /Demi
ITC Bookman Demi Italic	/Bkman, /Demi, /Italic
ITC Bookman Light	/Bkman, /Light
ITC Bookman Light Italic	/Bkman, /Light, /Italic
ITC Zapf Chancery Medium Italic	/Zapfchancery, /Medium, /Italic
ITC Zapf Dingbats	/Zapfdingbats
New Century Schoolbook	/Schoolbook
New Century Schoolbook Bold	/Schoolbook, /Bold
New Century Schoolbook Italic	/Schoolbook, /Italic
New Century Schoolbook Bold Italic	/Schoolbook, /Bold, /Italic
Palatino	/Palatino
Palatino Bold	/Palatino, /Bold
Palatino Italic	/Palatino, /Italic
Palatino Bold Italic	/Palatino, /Bold, /Italic
Symbol	/Symbol
Times	/Times
Times Bold	/Times, /Bold
Times Italic	/Times, /Italic
Times Bold Italic	/Times, /Bold, /Italic

When using PostScript fonts, commands may be inserted into the text in order to specify positioning and special characters. These commands are described in Table A-14. They are similar to the commands provided for the standard software characters.

Table A-14: PostScript Text Positioning Commands

Command	Description
!A	Shift above the division line.
!E	Shift up to the exponent level and decrease the character size by a factor of 0.44.
!MX	Insert a bullet character.
!N	Shift back to the normal level and original character size.
!B	Shift below the division line.
!I	Shift down to the index level and decrease the character size by a factor of 0.44.
!!	Display the ! symbol.

Using Color PostScript Devices

If you have a color PostScript device you can enable the use of color with the statement:

```
DEVICE, /Color
```

Enabling color also enables the color tables. Text and graphic color indices are translated to RGB by dividing the red, green, and blue color table values by 255. As with most display devices, color indices range from 0 to 255. Zero is normally black and white is normally represented by an index of 255.

PostScript Supports Color Images

As with black and white PostScript, images may be output with 1, 2, 3, 4 or 8 bits, yielding 1, 2, 16, or 256 possible colors. In addition, images can be either pseudo-color or true color. For a thorough comparison of pseudo-color and true color, see *Not all Color Images are True-color Images* on page 146.

Changing the Image Background Color

Images that are displayed with a black background on a monitor frequently look better if the background is changed to white when displayed with PostScript. This is easily done with the following statement, where A is the image variable:

```
A(WHERE(A EQ 0B)) = 255B
```

Creating Publication-quality Documents

The combination of PV-WAVE and the PostScript page description language gives you a powerful tool for creating publication-quality documents with text, graphics, and images arranged together on the page. Three examples of how to insert PV-WAVE PostScript output into text documents follow, but these are not exclusive. Consult your word processing or desktop publishing manual to see how your software handles encapsulated PostScript files.

Inserting PV-WAVE Plots into LAT$_E$X Documents

L$_A$T$_E$X is a special version of the T$_E$X page formatting language developed by Donald E. Knuth for producing publication-quality documents. LAT$_E$X was developed by Leslie Lamport to make the T$_E$X language easier to use. Although T$_E$X is released into the public domain, commercial versions of T$_E$X are supported by private vendors. One such vendor is ArborText, Inc., of Ann Arbor, Michigan. The examples below describe how Visual Numerics has used ArborText's T$_E$X, LAT$_E$X, and DVILASER/PS software on Sun workstations to incorporate PV-WAVE graphs and images into publication-quality documents.

Figure A-2 is an example of a PV-WAVE-generated PostScript plot that has been inserted into a LAT$_E$X document. It was produced with the following PV-WAVE statements:

```
SET_PLOT, 'PS'
```

Select the PostScript driver.

```
DEVICE, /Encapsulated, Filename='pic1.ps'
```

Use ENCAPSULATED because output is for use with LATEX.

```
x = FINDGEN(200)
PLOT, 10000 * SIN(x/5) / EXP(x/100), $
   Linestyle=2, Title='PV-WAVE ' + $
   'PostScript plot', Xtitle='Point Number', $
   Ytitle='Y Axis Title', Font=0
```

Plot the sine wave. Set the font to hardware font.

```
OPLOT, 10000 * COS(x/5) / EXP(x/100), $
   Linestyle=4
```

Add the cosine.

```
XYOUTS, 100, -6000, 'Sine', Font=0
```

Annotate the plot.

```
OPLOT, [120, 180], [-6000, -6000], Linestyle=2
```

Annotate the plot

```
XYOUTS, 100, -8000, 'Cosine', Font=0
```

Annotate the plot

```
OPLOT, [120, 180], [-8000, -8000], Linestyle=4
```

Note the use of the *Encapsulated* keyword in the call to DEVICE. This is what allows you to import the file into a LAT$_E$X document. Simply omit the *Encapsulated* keyword from the call to DEVICE if you want to produce a plot that can be printed directly.

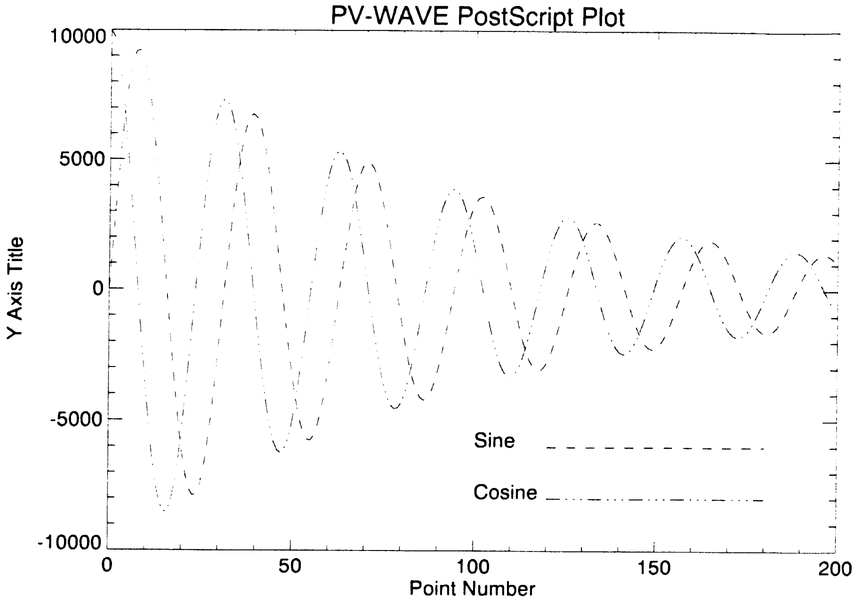


Figure A-2 : Sample PostScript plot using Helvetica font.

Any kind of PostScript plot can be included in LAT$_E$X documents. In Figure A-3, a PV-WAVE PostScript image has been included. In this case, the same image is reproduced four times. Each time, a different number of bits are used per image pixel. The illustration was produced with the following PV-WAVE statements:

```
SET_PLOT, 'PS'
DEVICE, /Encapsulated, Filename='pic4.ps'
OPENR, 1, !Dir+'/data/mandril.img'
```

Open image file.

```
a = BYTARR(256, 256, /Nozero)
```

Variable to hold image.

```
READU, 1, a
```

Input the image.

```
CLOSE, 1
```

Done with the file.

```
FOR i = 0,3 DO BEGIN
```
Output the image four times.
```
    DEVICE, Bits_Per_Pixel=2^i
```
Use 1, 2,4, and 8 bits per pixel.
```
    TV,a,i ,Xsize=2.5, Ysize=2.5, /Inches
```
Output using TV with position numbers 0, 1, 2, and 3.
```
    ENDFOR
```

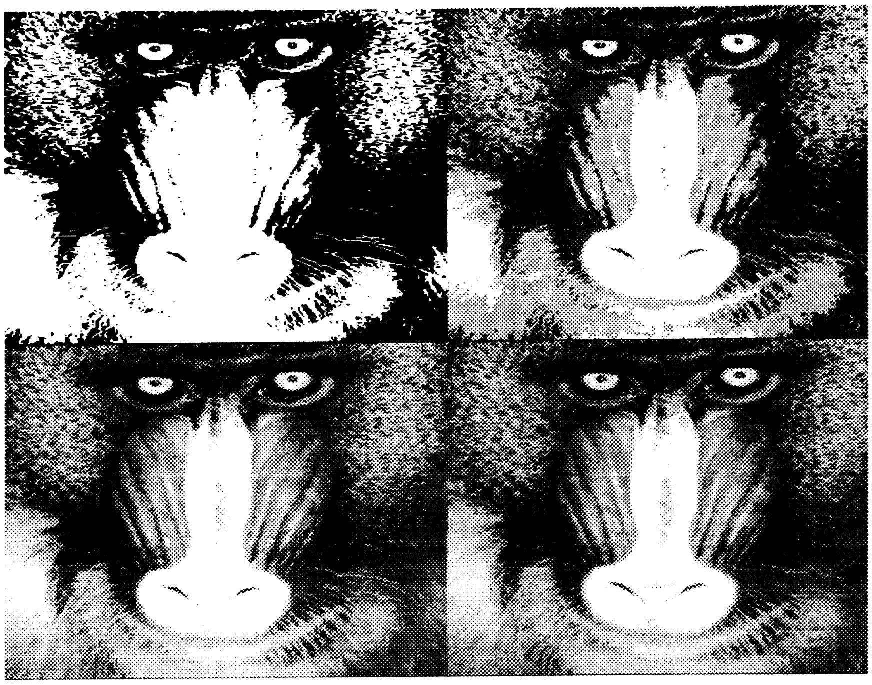


Figure A-3 1, 2, 4, and 8-bit PostScript images.

The LAT$_E$X Insertplot Macro

The following LAT$_E$X macro, named `insertplot`, is used to insert PV-WAVE-generated PostScript files into LAT$_E$X documents. The definition of this macro depends upon the T$_E$X DVI to PostScript translation program used and is therefore not portable between various DVI programs.

However, given familiarity with T_EX it is relatively easy to modify the macro to suit the various DVI programs encountered in practice. The `insertplot` macro is as follows:

```
% Insert a PostScript plot made by
% PV-WAVE into a LATEX document:
% For the ArborText program dvips.
%
% This macro creates a captioned figure
% of the specified size and uses the
% \special command to insert the Post
% Script.
%
% Usage: \insertplot{file}{caption}
% {label}{width}{height} file = name of
% file containing the PostScript.
% caption = caption of figure
% label = latex \label{} for figure to
% be used by \ref{} macro.
% width = width of figure, in inches.
% height = height of figure, in inches.
%
%
%
% Insert a plot.
\newcommand{\insertplot}[5]{%
% Usage: \insertplot{file}{caption}
% {label}{width in inches}{height}
\begin{figure}%
\hfill\hbox to 0.05in{\vbox to #5in{\vfil%
\inputplot{#1}{#4}{#5}%Include the plot
% file
}\hfill}%
\hfill\vspace{-.1in}% Fudge factor to
% tighten things up a bit.
\caption{#2}\label{#3}
\end{figure}}
%
%
% Include a PostScript File, this varies
% according to the DVI program: Usage:
```

```
% \inputplot{filename}{width}{height}
% When called from insertplot, the current
% position is at the bottom CENTER of the
% figure box.
\newcommand{\inputplot}[3]{%
% Output PostScript commands to scale
% default sized (7 wide by 5 high) PV-
% WAVE plot into the specified size. Also
% set origin to the current point less
% half the width of the box, centering the
% box above the current point.
\special{ps:: gsave #2 -36 mul 0 rmoveto
  currentpoint translate #2 7.0 div #3 5.0
  div scale}%
\special{ps: plotfile #1}\special{ps::
   grestore}}
%
%
```

Inserting PV-WAVE Plots into Microsoft Word Documents

PV-WAVE graphics and images can also be inserted into documents created with Microsoft Corporation's Microsoft Word. The example below illustrates how PV-WAVE graphics have been included in Microsoft Word documents written on an Apple Macintosh computer. The plot in Figure A-4 can be included in a Microsoft Word file by doing the following. First, type the following PV-WAVE commands to create the plot:

```
SET_PLOT, 'PS'
```

Use the PostScript driver as the output graphics device.

```
DEVICE, /Encapsulated, /Inches, $
   Xsize=5.0, Ysize=3.5, $
   Filename='polygon.ps'
```

Use encapsulated PostScript. Specify the size of the plot in inches.

```
X = FINDGEN(200)
```

```
A = 10000*sin(X/5) / exp(X/100)
```

Create upper sine wave.

```
PLOT, A, /Nodata, $
   Title='PV-WAVE PostScript Plot', $
   Xtitle='Point Number', $
   Ytitle='Y Axis Title', Font=0
```

Make axes and titles.

```
C = [X, rotate(X, 2)]
```

Vector of X vertices for polygon filling. Note that the ROTATE(V,2) function call returns the vector V in reverse order.

```
D = [A, rotate(A – 2000, 2)]
```

Vector of Y vertices for polygon filling.

```
POLYFILL, C, D, Color=192
```

Fill the region using an intensity of about 75% white.

```
DEVICE, /Close_File
```

Close the PostScript file.

Then quit PV-WAVE and transfer the encapsulated PostScript file (`polygon.ps`) to the Macintosh computer. This can be done with the proper communication software and cables. The PostScript file is a standard ASCII text file.

Next, start Microsoft Word and edit the encapsulated PostScript file. Add the following line to the top of the file:

```
%.pic.
```

The periods before and after the letters are necessary. This line should be the very first line in the file. (This is only one of several ways to include PostScript files in Word documents. See the Microsoft Word manual for others.) Select the entire file and choose *Define Styles* from the *Format* menu. Enter "PostScript" as the name of the *New Style* in the *Style* box, and then click the *Apply* button. Word does not display this style in the *Define Styles* box until it is used. The style's format is Normal plus 10-point font, bold, hidden text. Select the entire file and paste it to the Clipboard.

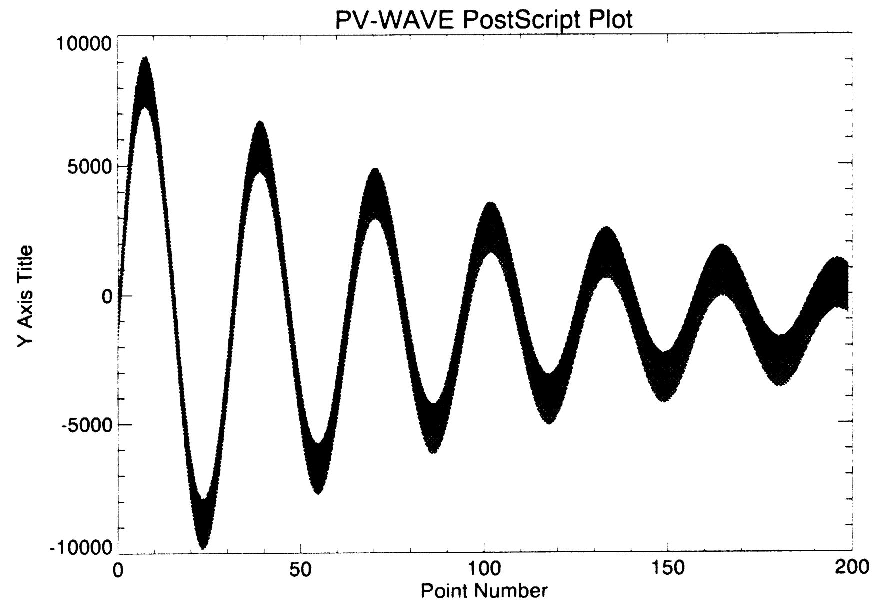


Figure A-4 Polygon filling example.

Finally, open the Microsoft Word document that will receive the graphic. At the place where the graphic is to be inserted, place a graphic box (with the *Insert Graphic* command) and size it to the appropriate size, in this case 5 inches wide and 3.5 inches high. Directly above the graphic box, paste the encapsulated PostScript file. It is important that the graphic box be on the first non-PostScript style line following the pasted PostScript commands. You will not see the graphic displayed in the graphic box, but if you print the file normally the graphic will be included inside the box.

Inserting PV-WAVE Plots into Ventura Publisher Documents

Ventura Publisher, developed by Xerox Corporation for IBM and compatible personal computers, is also able to accept PV-WAVE graphics and images in the form of encapsulated Post-

Script files. Produce the graphic file with the same PV-WAVE commands used above for inclusion into Microsoft Word files. Then transfer the PostScript file to your personal computer using appropriate file transfer software and cables. The PostScript file is a standard ASCII text file.

To incorporate a graph or image in a Ventura Publisher document it must have an `.EPS` file extension appended to its name. Once this is accomplished, run Ventura Publisher and open the chapter or load the document that will include the inserted graph or image. Make a frame the appropriate size for the insert and be sure the frame is selected or active. Then choose the LOAD TEXT/PICTURE command from the appropriate menu and choose the PostScript file containing the insert. You will not be able to see the graph or image from within Ventura (the frame will have a large X through it), but the insert will print properly.

QMS QUIC Output

QUIC (version 3.2 or higher) is a language used to produce graphics on the QMS LaserGrafix series of laser printers.

To use QMS QUIC as the current graphics device, issue the PV-WAVE command:

```
SET_PLOT, 'QMS'
```

This causes PV-WAVE to use QUIC for producing graphical output. Once the QMS QUIC driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in *Controlling QMS QUIC Output with DEVICE Keywords* on page A-51. The default settings for the QMS QUIC driver are given in Table A-15.

Table A-15: Default QMS QUIC Driver Settings

Setting	Default Value
Output file name	wave.qms
Orientation	portrait
Paper_Size	A
Scale_Factor	1.0
Term_Orient	portrait
Term_Font	204
Term_Char_Spacing	10 chars/inch
Term_Line_Spacing	6 lines/inch
Term_Top_Margin	0.5 inches
Term_Bottom_Margin	0.3333 inches
Term_Left_Margin	0.5 inches
Term_Right_Margin	0.0 inches
X offset	1.905 cm (0.75 in.)
X size	17.78 cm (7.0 in.)
Yoffset	5.08 cm (2.0 in.)
Y size	17.78 cm (7.0 in.)

Use the statement:

```
INFO, /Device
```

to view the current QMS QUIC driver settings.

Note Unlike monitors where white is the most visible color, QMS QUIC writes black on white paper. Setting the output color index to 0, the default when QMS QUIC output is selected, writes black. A color index of 1 writes white lines, white on black text, and reverses polygon fill patterns.

Supported Features of QMS QUIC

PV-WAVE is able to produce a wide variety of graphical output using QMS QUIC.

Here is a list of what is supported:

- All types of vector graphics can be generated, including line plots, contours, and surfaces.
- Images can be output, although only black or white pixel values exist. (Color index 0 is mapped to black.)
- QUIC (version 3.2 or higher) can do polygon filling in hardware.
- It is possible to generate graphics using the hardware-generated exact-sized text characters, for a considerable improvement in speed. To use hardware characters, set the !P.Font system variable to zero, or set the *Font* keyword to the plotting routines to zero.

The following is not supported:

- Since laser printers are not interactive devices, they cannot support such operations as cursors and windows.

Specifying Linestyles in QMS QUIC Output

The *Linestyle* graphics keyword allows specifying any of six linestyles, all of which are supported by the QMS/QUIC driver. For information on the *Linestyle* keyword, see Chapter 3 *Graphics and Plotting Keywords*, in the *PV-WAVE Reference.*

Printing QMS Graphics Output Files

QMS graphics output files can be printed normally, with the following exception. The `/HEADER` command qualifier should not be used with the VAX VMS `PRINT` command because it sometimes causes undesired lines to be printed on the page.

Controlling QMS QUIC Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the QMS QUIC driver:

Close_File – PV-WAVE creates, opens, and writes a file containing the generated graphics output. The *Close_File* keyword outputs any buffered commands and closes the file.

Caution Under UNIX, if you close the output file with the *Close_File* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

See the discussion of printing output files in the section, *Producing Hardcopy Output* on page A-3, for more information on this topic.

Filename – By default all generated output is sent to a file named `wave.qms`. The *Filename* keyword can be used to change this default. When you specify a filename, the following occurs:

- If the file is already open (as happens if plotting commands have been directed to the file since the call to SET_PLOT), then the file is completed and closed as if *Close_File* had been specified.
- The specified file is opened for subsequent graphics output.

If you wish to send QMS QUIC output directly to a printer without generating an intermediate file, you should specify the device-specific file for the printer as the argument to *Filename*. For example, if your printer is connected to a serial input/output port known on your system as `/dev/ttya`, you would issue the command:

```
DEVICE, Filename='/dev/ttya'
```

All subsequent QMS QUIC output is sent directly to the printer connected to serial port `/dev/ttya`.

Inches – By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, they are taken to be in inches instead.

Landscape – PV-WAVE normally generates plots with portrait orientation (the X axis is along the short dimension of the page). If *Landscape* is present, landscape orientation (the X axis is along the long dimension of the page) is used instead.

Paper_Size – A string value that determines the size of the paper. Possible values are given in Table A-16. The default value is A (8.5-by-11 inches).

Paper_Size must be changed before output is sent to the file. If output has already been written to the file before the *Paper_Size* value is changed, the new value will have no effect.

Also, this value affects the default values of *Xsize*, *Ysize*, *Xoffset*, and *Yoffset*. To see what the new values are, set the *Paper_Size* with the DEVICE procedure, then enter `INFO, /Device` to see the new default values.

Table A-16: Paper Size

Value	Meaning
A	Letter size, 8.5 by 11 inches
LEGAL	Legal size, 8.5 by 14 inches
A4	Metric size, 210 by 297 mm
B	Large size, 11 by 17 inches

Portrait – If *Portrait* is present, PV-WAVE generates plots using portrait orientation, the default.

Scale_Factor – Specifies a scale factor applied to the entire plot. The default value is 1.0, which allows output to appear at its normal size. *Scale_Factor* shrinks or magnifies output.

Note The driver may substitute a smaller scale factor for magnification if the value supplied for *Scale_Factor* exceeds the device's capabilities.

Term_Bottom_Margin – Determines the offset from the bottom of the page to the bottom margin of the paper after driver termination. The value is expressed in centimeters unless *Inches* is specified.

Note All keywords that begin with the letters "*Term_*" select attributes for normal (non-graphics) line printer mode. These Line Printer mode reset values are used by the driver to reset the device to a state in which line printer jobs and word processing jobs can properly use the device. In other words, graphics and normal printing jobs can be intermixed. (If there is no graphics output, no reset is necessary, and the output file will have zero length.)

No error checking is done on these values; you must be sure they are valid.

Term_Char_Spacing – Sets the character spacing after driver termination. Valid choices are 10, 12, or 15 characters per inch.

Term_Font – Selects the default font for the device after driver termination. Set the keyword to the number of the desired font.

Term_Left_Margin – Selects the offset from the left side of the page to the left margin after driver termination. The value is expressed in centimeters unless *Inches* is specified.

Term_Line_Spacing – Determines the line spacing for the device after driver termination. Alternatives are 6 or 8 lines per inch.

Term_Orient – Determines the orientation of the printer for text after the driver terminates. If P is selected, the long axis of the paper is vertical (i.e., portrait mode). If L is selected, the long axis is horizontal (landscape mode).

Term_Right_Margin – Selects the offset from the right of the page to the right margin after driver termination. The value is expressed in centimeters unless *Inches* is specified.

Term_Top_Margin – Determines the offset from the top of the page to the top margin of the paper after driver termination. The value is expressed in centimeters unless *Inches* is specified.

Xoffset – Specifies the X position on the page of the lower-left corner of output generated by PV-WAVE. By default, *Xoffset* is specified in centimeters unless *Inches* is specified. The minimum value of *Xoffset* is 0.25 inches.

Xsize – Specifies the width of output that PV-WAVE generates. By default, *Xsize* is specified in centimeters unless *Inches* is specified. The minimum value of *Xsize* is 1.0 inches.

Yoffset – Specifies the Y position on the page of the lower-left corner of output generated by PV-WAVE. By default, *Yoffset* is specified in centimeters unless *Inches* is specified. The minimum value of *Yoffset* is 0.25 inches.

Ysize – Specifies the height of output that PV-WAVE generates. By default, *Ysize* is specified in centimeters unless *Inches* is specified. The minimum value of *Ysize* is 1.0 inches.

Note If any output has been written to a page and the *Landscape*, *Portrait*, *Scale_Factor*, *Xoffset*, *Yoffset*, *Xsize*, or *Ysize* keywords are changed, the existing page will be output and a new blank page will be started with the new values.

Note If anything has been drawn on a page, and *Orientation*, X or Y offset or size, or *Scale_Factor* is changed, the current page will be ejected, and a new blank page will be started with the new values.

Regis Output

PV-WAVE provides Regis graphics output for the DEC VT240, VT330, and VT340 series of terminals. To output graphics to such terminals, enter the PV-WAVE command:

```
SET_PLOT, 'REGIS'
```

This causes PV-WAVE to use the Regis driver for producing graphical output. Once the Regis driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in the next section, *Controlling Regis Output with DEVICE Keywords*.

Note The Regis driver is included for all available PV-WAVE platforms.

Controlling Regis Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the Regis driver:

Average_Lines – Controls the method of writing images to the VT240. If this keyword is set, as it is by default, even and odd pairs of image lines are averaged and written to a single line. If clear, each image line is written to the screen (see the discussion below).

This keyword has no effect when using a VT300 series terminal.

Close_File – PV-WAVE creates, opens, and writes a file containing the generated graphics output. The *Close_File* keyword outputs any buffered commands and closes the file.

Caution Under UNIX, if you close the output file with the *Close_File* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

See the discussion of printing output files in the section, *Producing Hardcopy Output* on page A-3, for more information on this topic.

Filename – By default all generated output is sent to a file named `wave.regis`. The *Filename* keyword can be used to change this default. When you specify a filename, the following occurs:

- If the file is already open (as happens if graphics have been directed to the file since the call to SET_PLOT), then the file is completed and closed as if *Close_File* had been specified.
- The specified file is opened for subsequent graphics output.

Plot_To – Directs the Regis output that would normally go to the user's terminal to the specified I/O unit. The logical unit specified should be open with write access to a device or file.

Do not use the interactive graphics cursor when graphic output is not directed to your terminal. To direct the graphic data to both the terminal and the file, set the unit to the negative of the actual unit number. If the specified unit number is zero, then Regis output to the file is stopped.

Tty – Directs output to both a file and the terminal.

VT240 – Sets driver for VT240 series terminals.

VT241 – Same as *VT240*.

VT340 – (The Default) Sets driver for VT340 series terminals.

VT341 – Same as *VT340*

The default setting for Regis output is: VT340, 16 colors, 4 bits per pixel.

Limitations of REGIS Output

- Four colors are available with VT240 and VT241 terminals, sixteen colors are available with the VT330 and VT340.
- Thick lines are emulated by filling polygons. There may be a difference in linestyle appearance between thick and normal lines.
- Image output is slow and poor quality, especially on the VT240 series.
- The VT240 is only able to write pixels on even numbered screen lines. PV-WAVE offers two methods of writing images to the VT240:

  Even and odd pairs of rows are averaged and written to the screen. An *n*, *m* image will occupy *n* columns and *m* screen rows. If this method is selected, graphics and image coordinates coincide. This method is the default: `Average_Lines=1`. Routines that rely on a uniform graphics and image coordinate system, such as SHADE_SURF, will only work in this mode.

Each line of the image is written to the screen, displaying every image pixel. An *n*, *m* image occupies 2*m* lines on the screen (`Average_Lines=0`). Graphics and image coordinates coincide only at the lower left corner of the image.

- Pixel values may not be read back from the terminal, rendering the TVRD function inoperable.

SIXEL Output

SIXEL is used by Digital Equipment Corporation printers to produce graphics output.

Note The SIXEL driver is included for all available PV-WAVE platforms.

To direct graphics output to a SIXEL file, issue the command:

```
SET_PLOT,'SIXEL'
```

This causes PV-WAVE to use the SIXEL driver for producing graphical output. Once the SIXEL driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in *Controlling SIXEL Output with DEVICE Keywords* on page A-59. The default settings for the SIXEL driver are given in Table A-17.

Table A-17: Default SIXEL Driver Settings

Setting	Default Value
Output file name	WAVE.SXL
Mode	Portrait
Dither Method	Floyd-Steinberg
Resolution	150 dpi
Horizontal Offset	2.54 cm (1 in.)
Vertical Offset	2.54 cm (1 in.)
Width	15.24 cm (6 in.)
Height	15.24 cm (6 in.)

Note The SIXEL driver supports monochrome output only. It does not support colors or color table definitions.

Use `INFO, /Device` to view the driver's current settings.

The SIXEL driver has been graciously provided by Mark Rivers and Brookhaven National Labs. The driver software is considered "public domain" and is distributed by Visual Numerics without restriction.

Controlling SIXEL Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the SIXEL driver:

Centimeters — Use the *Centimeters* keyword to specify the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords in centimeters. Centimeters is the default unit of measurement for these keywords.

Close — PV-WAVE creates, opens and writes a file containing the generated graphics output. The *Close* keyword outputs any buffered commands and closes the file.

Caution Under UNIX, if you close the output file with the *Close* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

See the discussion of printing output files in the section, *Producing Hardcopy Output* on page A-3, for more information on this topic.

Filename — By default all generated output is sent to a file named `WAVE.SXL`. The *Filename* keyword can be used to change this default. When you specify a filename, the following occurs:

- If the file is already open (as happens if graphics have been directed to the file since the call to SET_PLOT), then the file is completed and closed as if *Close* had been specified.
- The specified file is opened for subsequent graphics output.

Floyd — If present and nonzero, selects the Floyd-Steinberg method of dithering. (This is the default method.) For more information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Inches – By default, the *Xoffset, Xsize, Yoffset,* and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, they are taken to be in inches instead.

Landscape – PV-WAVE normally generates plots with portrait orientation (the X axis is along the short dimension of the page). If *Landscape* is present, landscape orientation (the X axis is along the long dimension of the page) is used instead.

Ordered – (Default) Selects the Ordered method of dithering when displaying images on a monochrome display. For more information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Portrait – If *Portrait* is present, PV-WAVE will generate plots using portrait orientation, the default.

Resolution – The resolution at which the SIXEL printer works. Must be specified for X and Y axes. For example:

```
DEVICE, Resolution=[300,300]
```

The default is 150 dpi. Lower resolution gives smaller output files, while higher resolution gives superior quality.

Threshold – Specifies use of the threshold dithering algorithm – the simplest method. For more information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Xoffset – Specifies the X position on the page of the lower-left corner of output generated by PV-WAVE. By default, *Xoffset* is specified in centimeters unless *Inches* is specified.

Xsize – Specifies the width of output PV-WAVE generates. By default, *Xsize* is specified in centimeters unless *Inches* is specified.

Yoffset – Specifies the Y position on the page of the lower-left corner of output generated by PV-WAVE. By default, *Yoffset* is specified in centimeters unless *Inches* is specified.

Ysize – Specifies the height of output generated by PV-WAVE. By default, *Ysize* is specified in centimeters unless *Inches* is specified.

SIXEL Device Considerations

The SIXEL driver uses device escape sequences for the DEVICE keywords *Resolution*, *Portrait*, *Landscape*, *Xoffset*, and *Yoffset.* Some older SIXEL devices do not recognize these escape sequences and the presence of them in the output file could cause unexpected results. The SIXEL driver does not simulate these features for the older SIXEL devices.

SIXEL Image Background Color

Images that are displayed with a black background on a monitor frequently look better if the background is changed to white when displayed with SIXEL. This is easily done with the following statement, where A is the image variable:

```
A(WHERE(A EQ 0B)) = 255B
```

Tektronix 4510 Rasterizer

To get output for the Tektronix 4510 rasterizer, use the command:

```
SET_PLOT, '4510'
```

PV-WAVE then creates a file that can be sent to the 4510. Once the 4510 driver has been selected, the DEVICE procedure is used to control the appearance of the output, as explained in *Controlling Tektronix 4510 Output with DEVICE Keywords* on page A-63. The default values for the DEVICE procedure were chosen for best results with a Tektronix 4692 ink-jet copier. The default values are listed in Table A-18.

Table A-18: Default 4510 Driver Settings

Setting	Default Value
Output file name	wave.4510
Orientation	Landscape
Pixel aspect ratio	1.00
Pixel size	4

Use the statement:

```
INFO, /Device
```

to view the current 4510 driver settings.

Supported Features of the 4510 Driver

Here is a list of what is supported for the 4510:

- Vector graphics are fully supported.
- 256 colors are available from the 4510. The indices range from 0 to 255. The initial color table index definitions are shown in Table A-19.

  All the color indices (including 0, which is *always* the background) can be redefined as necessary using LOADCT. There is no black/white reversal. A color defined with Red, Green, and Blue (RGB) values of (0, 0, 0) will give black. Some of the color tables that are predefined by PV-WAVE define color 0 as (0, 0, 0), which will give a black background.

Tip To avoid a black background, call the TVLCT function with RGB values of 255, 255, 255 for index 0 after loading a predefined color table.

Table A-19: Initial Color Indices of the 4510

Index Value	Color
0	Background, Initially white
1	White
2	Red
3	Green
4	Blue
5	Yellow
255	Black

- Polygon fill is done by the 4510 rasterizer. Patterns –255 through 0 fill the polygon with a single color index. Patterns 1 through 16 give the standard Tektronix 4112/3 fill patterns, and patterns 50 through 174 give dithered patterns.

- Hardware characters are available with the 4510.
- Raster images are made available by simulating pixels with solid line segments.
- The simulated pixels of raster images are available in four sizes (1 – 4). The aspect ratio of the simulated pixels can be changed.

The following is not supported:

- The 4510 is not an interactive device and thus cannot support windows and cursors.

Controlling Tektronix 4510 Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the 4510 driver:

Close_File – PV-WAVE creates, opens and writes a file containing the generated graphics output. The *Close_File* keyword outputs any buffered commands and closes the file. Usage example:

```
DEVICE, /Close_File
```

Caution Under UNIX, if you close the output file with the *Close_File* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

Filename – By default, all 4510 output is sent to a file named `wave.4510`. *Filename* allows you to specify the name of the output file. If *Filename* is used, the specified file is opened for subsequent graphics output. If the file is already open (as happens if plotting commands have been directed to the file since the call to SET_PLOT), then the file is completed and closed as if *Close_File* had been specified. Usage example:

```
DEVICE, Filename='fractal.dat'
```

Landscape/Portrait – Output is normally generated with a landscape orientation. Specifying *Portrait* will place the X axis on the short dimension of the paper. To specify landscape orientation, use the *Landscape* keyword. Usage examples:

```
DEVICE, /Landscape
DEVICE, /Portrait
```

Pixel_Aspect – This keyword controls the aspect ratio of the simulated pixels to ensure a square raster image. The 4510 has a range of coordinates of {0...4096} in both the X and Y directions. Since the drawing surface of the paper is not square, some distortion can appear. The default ratio of 1.0 is set by PV-WAVE when the 4510 driver is initialized. Use *Pixel_Aspect* to give pixel output the correct aspect ratio. Usage example:

```
DEVICE, Pixel_Aspect=1.25
```

Pixel_Size – Raster operations are simulated using lines. Since the 4510 supports four widths of lines, there are four pixel sizes. A *Pixel_Size* of 4 will generally provide the best results. The available sizes are 1, 2, 3, 4. Usage example:

```
DEVICE,Pixel_Size=4
```

Usage Warnings

There are three potential areas of trouble when using the 4510 driver:

- The 4510 uses dithering patterns in hardware to simulate different color mixtures. When drawing lines and text with a width of less than eight pixels, some colors may not be faithfully reproduced, and in some extreme cases, the line or text may disappear entirely.
- Larger or more complicated raster images may not be entirely printed on one page. Raster images are simulated by drawing horizontal lines of various lengths and colors. Because of the simulation, raster image files can get quite large. If the files are so large that the 4510 Rasterizer runs out of display list mem-

ory, an error will occur. That portion of the image that has filled the display list will be printed. The rest of the image will be printed on subsequent pages. The only known solution to this problem is to add memory to the 4510 Rasterizer.

- Although the 4510 driver supports raster images, shaded surfaces will cause problems. With a resolution of 4096 in both the X and Y axes, PV-WAVE cannot allocate the amount of memory required by the shaded surface routine and PV-WAVE will abort.

Tektronix Terminals

The Tektronix 4000 (4010, 4014, etc.), 4100 and 4200 series of graphics terminals (and the multitude of terminals and microcomputers that emulate them) are among the most common graphics devices available. To use PV-WAVE graphics with such terminals, enter the PV-WAVE command:

```
SET_PLOT, 'TEK'
```

This causes PV-WAVE to use the Tektronix driver for producing graphical output.

Once the Tektronix driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, and to configure PV-WAVE for the specific features of your terminal. See *Controlling Tektronix Output with DEVICE Keywords* on page A-66 for more information. If you never call the DEVICE procedure, PV-WAVE assumes a standard Tektronix 4000 series compatible terminal.

The 4200 series is upwardly compatible with the 4100 series; all references to the 4100 series also include the 4200 series. To set up PV-WAVE for use with a 4100 series compatible terminal with *n* colors:

```
SET_PLOT, 'TEK'
DEVICE, /TEK4100, Colors=n
```

The number of colors should be set to 2^B where B is the number of bit planes in your terminal. If you use a Tektronix compatible terminal that requires calling the DEVICE procedure for configuration, you should probably create and use a start-up procedure that calls the DEVICE procedure, as described in the section, *Modifying Your PV-WAVE Environment* on page 44.

The line drawing procedures work with all models. Color and the display of images (albeit very slowly and frequently of a poor quality because of the low number of colors) is usable only with 4100 series terminals. Hardware polygon fill works only on the 4100 series.

Because of the tremendous variation among the requirements and abilities of these terminals, it is crucial that you configure PV-WAVE properly for your terminal.

Controlling Tektronix Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the Tektronix driver:

Colors – The number of colors supported by the terminal. Only used with 4100 series terminals. For example, if your terminal has 4-bit planes, the number of colors is $2^4 - 16$:

```
DEVICE, Colors=16
```

Valid values of this parameter are: 2, 4, 8, 16, or 64; other values cause problems. Some Tektronix terminals do not operate properly if this parameter does not exactly match the number of colors available in the terminal hardware.

This parameter sets the field !D.N_Colors, which affects the loading of color tables via the Standard Library procedures, the scaling used by the TVSCL procedure, and the number of bits output by the TV procedure to the terminal. It also changes the default color, !P. Color, to the number of colors minus one.

Gin_Chars – The number of characters PV-WAVE reads when accepting a GIN (Graphics Input) report. The default is 5. If your terminal is configured to send a carriage return at the end of each

GIN report, set this parameter to 6. If the number of GIN characters is too large, the CURSOR procedure will not respond until two or more keys are struck. If it is too small, the extra characters sent by the terminal will appear as input to the next PV-WAVE prompt.

Plot_To – Directs the Tektronix graphic output that would normally go to the user's terminal to the specified I/O unit. The logical unit specified should be open with write access to a device or file. Save graphic output in files for later playback, redirection to other terminals, or to devices that accept Tektronix graphic commands.

Do not use the interactive graphics cursor when graphic output is not directed to your terminal. To direct the graphic data to both the terminal and the file, set the unit to the negative of the actual unit number. If the specified unit number is zero, then Tektronix output to the file is stopped.

Reset_String – The string used to place the terminal back into the normal interactive mode after drawing graphics. Use this parameter, in conjunction with the *Set_String* keyword, to control the mode switching of your terminal. For example, the GraphON 200 series terminals require the string `<ESC>2` to activate the alphanumeric window after drawing graphics. The call to set this is:

```
DEVICE, Reset=string(27b) + '2'
```

If the 4100 series mode switch is set, using the keyword *Tek4100*, the default mode re-setting string is `<ESC>%!1`, which selects the ANSII code mode.

Set_String – The string used to place the terminal into the graphics mode from the normal interactive terminal mode. If the 4100 series mode switch is set, using the keyword *Tek4100*, the default graphic mode setting string is `<ESC>%!0`, which selects the Tektronix code mode.

Tek4014 – If nonzero, specifies that coordinates are to be output with full 12-bit resolution. If this keyword is not present or is zero, 10-bit coordinates are output. By default, PV-WAVE sends 10-bit coordinates. 12-bit coordinates are compatible with most terminals, even those without the full resolution, but require more characters to send.

Note The 4014 and the 4100 modes may be used together. The coordinate system PV-WAVE uses for the Tektronix is 0 to 4095 in the X direction and 0 to 3120 in the Y direction, even when not in the 4014 mode – in the 10-bit case the internal coordinates are divided by 4 prior to output.

Tek4100 – Indicates that the terminal is a 4100 series terminal. The use of color, ANSII and Tektronix mode switching, hardware line styles, and pixel output with the TV procedure is supported with 4100 series terminals. Also, text is output differently.

The default setting for Tektronix output is: 10-bit coordinates, 4000 series terminals, and no use of color.

Limitations of Tektronix and Tektronix-compatible Terminals

Because of hardware restrictions, you will encounter the following limitations when using Tektronix and Tektronix-compatible terminals with PV-WAVE:

- Pixel coordinates do not match the coordinates used by the rest of the graphic procedures. This is because no two models of Tektronix terminals are alike. The graphics procedures use the default coordinate system of 1024-by-780, or 4096-by-3120 in the 12-bit mode. The size of the pixel memory and coordinate system vary widely between models. The *Position* parameter for the TV and TVSCL procedures does not work.
- The cursor cannot be positioned from the computer, meaning the TVCRS procedure may not be used in the Tektronix mode.
- Pixel values may not be read back from the terminal, rendering the TVRD function inoperable.

Caution If you try to display images produced with the PV-WAVE SHADE_SURF and SHOW3 procedures, PV-WAVE may abort. Because of a limitation in the range of image coordinates available on Tektronix devices, they are not well suited to the display of images.

X Window System

PV-WAVE uses the X Window System to provide an environment in which you can create one or more independent windows, each of which can be used for the display of graphics and/or images.

Note The X Window System is the default windowing system for all available PV-WAVE platforms.

In X there are two basic cooperating processes, clients and servers. A server usually consists of a display, keyboard, and pointer (such as a mouse) as well as the software that controls them. Client processes (such as PV-WAVE) display graphics and text on the screen of a server by sending X protocol requests across the network to the server. Although in the simplest case, the server and client reside on the same machine, this network-based design allows more elaborate configurations.

Controlling Where PV-WAVE Graphics are Displayed

When you use X, the environment variable `$DISPLAY` (UNIX) or the logical `DECW$DISPLAY` (DECwindows Motif) must be set properly. Otherwise, PV-WAVE may appear to "hang". The following command allows PV-WAVE windows to appear on the local display (the current workstation). Enter one of these commands at the prompt of the current workstation before you start PV-WAVE, depending on whether you are using UNIX or VMS (DECwindows Motif):

UNIX
```
setenv DISPLAY nodename:0.0
```

VMS
```
SET DISPLAY /CREATE /NODE=nodename -
    /SCREEN=0.0 /TRANSPORT=transport_type
```

You must also be sure that an X server is running on the host machine specified with the `$DISPLAY` environment variable or `DECW$DISPLAY` logical, and that the machine PV-WAVE is running on has permission to communicate with that X server. Refer to the documentation for your X-compatible window manager to learn how to modify your X server's permissions list.

Selecting the X Driver

To use X as the current graphics device, enter the following PV-WAVE command:

```
SET_PLOT, 'X'
```

This causes PV-WAVE to use X for producing subsequent graphical output. Once the X driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in *Controlling the X Driver with DEVICE Keywords* on page A-71.

Listing the Current Settings for the X Driver

Use the command:

```
INFO, /Device
```

to view the current X driver settings.

Graphical User Interfaces (GUIs) for PV-WAVE Applications Running Under X

If you wish to include a graphical user interface (GUI) with a PV-WAVE application that you are writing for use with the X Window System, you have several choices:

- **WAVE Widgets** – A versatile, easy-to-use set of functions for creating Motif or OPEN LOOK GUIs for PV-WAVE applications. WAVE Widgets are designed for PV-WAVE developers with little or no experience using the Motif or OLIT GUI toolkits. See Chapter 15, *Using WAVE Widgets*, in the *PV-WAVE Programmer's Guide* for detailed information on the Widget Toolbox functions.
- **Widget Toolbox** – A set of highly flexible PV-WAVE functions used to create Motif or OPEN LOOK Graphical User Interfaces (GUIs) for PV-WAVE applications. The Widget Toolbox functions call Motif and OPEN LOOK Intrinsics Toolkit (OLIT) routines directly, and are designed primarily for developers who are already experienced using either Motif or OLIT. See Chapter 16, *Using the Widget Toolbox*, in the

PV-WAVE Programmer's Guide for detailed information on the Widget Toolbox functions.

- **GUI Builders** – This separate PV-WAVE option can be used to develop extensive, professional applications with PV-WAVE and the Motif GUI. This option must be purchased separately, and requires familiarity with the C programming language. For more information, refer to a separate volume, the *PV-WAVE GUI Builders User's Guide.*
- **C-based Applications** – PV-WAVE can be used to add visual data analysis capability to an existing C application. The application interface can be developed in C, and, via PV-WAVE's interapplication communication functions, the C application can call PV-WAVE to perform data processing and display functions. For more details, refer to Chapter 13, *Interapplication Communication*, in the *PV-WAVE Programmer's Guide.*

Note All the options listed above are fully compatible with the X Window System and can be used to facilitate access to your application. However, you are not required to use any of them – your application can still run under X, even though it does not have a Motif or OPEN LOOK GUI.

Controlling the X Driver with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the X driver:

Bypass_Translation – When this keyword is set, the translation table is bypassed and color indices can be directly specified. Pixel values read via the TVRD function are not translated if this keyword is set, and thus the result contains the actual (hardware) pixel values present in the display. By default, the translation table is used with shared color tables. When displays with static (read-only) visual classes and with private color tables are used, the translation table is *always* bypassed. For more information about the translation table, refer to *PV-WAVE's Color Translation Table* on page A-84.

Close_Display — Causes PV-WAVE to sever the connection with the X server. This has the effect of deleting all generated windows from the screen of the server, and returns PV-WAVE to the initial graphics state. One use for this option is to change the number of colors used. See the section, *When Color Characteristics are Determined* on page A-82, for details.

Copy — Copies a rectangular area of pixels from one region of a window to another. The argument of *Copy* is a six- or seven-element array: [X_s, Y_s, N_x, N_y, X_d, Y_d, W], where: (X_s, Y_s) is the lower-left corner of the source rectangle, (N_x, N_y) are the number of columns and rows in the rectangle, and (X_d, Y_d) is the coordinate of the destination rectangle. Optionally, W is the index of the window from which the pixels should be copied to the current window. If it is not supplied, the current window is used for both source and destination. *Copy* can be used to increase the pace of animation. This is described in *Using Pixmaps to Improve Application Performance* on page A-85.

Cursor_Crosshair — Selects the crosshair cursor type. The crosshair cursor style is the PV-WAVE default.

Cursor_Image — Specifies the cursor pattern. The value of this keyword must be a 16-line by 16-column bitmap, contained in a 16-element short integer vector. Each line of the bitmap must be a 16-bit pattern of ones and zeros, where one (1) is white and zero (0) is black. The offset from the upper-left pixel to the point that is considered the "hot spot" can be provided via the *Cursor_XY* keyword.

Cursor_Original — Selects the default cursor for the window system. Under X, it is the cursor in use by the window manager when PV-WAVE starts.

Cursor_Standard — Selects one of the predefined cursors provided by X. The available cursor shapes are defined in the file:

UNIX
```
/usr/include/X11/cursorfont.h
```

VMS
```
DECW$INCLUDE:CURSORFONT.H
```

In order to use one of these cursors, select the number of the font and provide it as the value of the *Cursor_Standard* keyword. For example, the `cursorfont.h` file gives the value of *Xc_Cross* as being 30. In order to make that the current cursor, use the statement:

```
DEVICE, Cursor_Standard=30
```

Cursor_XY – A two-element integer vector giving the X, Y pixel offset of the cursor "hot spot", the point which is considered to be the mouse position, from the upper-left corner of the cursor image. This parameter is only applicable if *Cursor_Image* is provided. The cursor image is displayed top-down – in other words, the first row is displayed at the top.

Floyd – If present and nonzero, selects the Floyd-Steinberg dithering method. For more information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Font – Specifies the name of the font used when the hardware font is selected. For example, to select the font `8X13`:

```
DEVICE, Font='8X13'
```

Usually, the window system provides a directory of font files that can be used by all applications. List the contents of that directory to find the fonts available on your system. The size of the font selected also affects the size of software-drawn text (e.g., the Hershey fonts). On some machines, fonts are kept in subdirectories of:

UNIX
```
/usr/lib/X11/fonts
```

VMS
```
SYS$SYSROOT:[SYSCOMMON.SYSFONT]
```

Note that hardware fonts cannot be rotated, scaled, or projected, and that the "!" commands accepted for software fonts for subscripts and superscripts do not work. When generating 3D plots, it is best to use the software characters because PV-WAVE can draw them in perspective with the rest of the plot. For more information on software fonts, see Chapter 9, *Software Fonts.*

Get_Graphics_Function – Returns the value of the current graphics function (set with the *Set_Graphics_Function* keyword). This can be used to remember the current graphics function, change it temporarily, and then restore it. For an example, see the example in the section *Using Graphics Functions to Manipulate Color* on page A-94.

Get_Window_Position – Places the X and Y device coordinates of the window's lower-left corner into a named variable.

Get_Write_Mask – Specifies the name of a variable to receive the current value of the write mask. For example:

```
    Get_Write_Mask=mask
```

For more information on the write mask, refer to *Using the Write Mask and Graphics Functions to Manipulate Color* on page A-93.

Ordered – If present and nonzero, selects the Ordered method of dithering. For more information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Pseudo_Color – When specified with a *Depth* value, selects the PseudoColor visual with *Depth* bits per pixel. For example:

```
    Pseudo_Color=8
```

The *Pseudo_Color* keyword has effect only if no windows have been created.

Retain – Specifies the default method used for backing store when creating new windows. This is the method used when the *Retain* keyword is not specified with the WINDOW procedure. Backing store is discussed in more detail in the subsection, *How Is Backing Store Handled?* on page A-7. The possible values for this keyword are summarized in Table A-3 on page A-8.

If *Retain* is not used to specify the default method, method 1 (server-supplied backing store) is used.

Set_Graphics_Function – X allows applications to specify the *Graphics Function.* This is a logical function which specifies how the source pixel values generated by a graphics operation are combined with the pixel values already present on the screen. The complete list of possible values is given in Table A-20. The default graphics function is GXcopy, which causes new pixels to completely overwrite any previous pixels.

Table A-20: Graphics Function Codes

Logical Function	Code	Definition
GXclear	0	0
GXand	1	src AND dst
GXandReverse	2	src AND (NOT dst)
GXcopy	3	src
GXandInverted	4	(NOT src) AND dst
GXnoop	5	dst
GXxor	6	src XOR dst
GXor	7	src OR dst
GXnor	8	(NOT src) AND (NOT dst)
GXequiv	9	(NOT src) XOR dst
GXinvert	10	(NOT dst)
GXorReverse	11	src OR (NOT dst)
GXcopyInverted	12	(NOT src)
GXorInverted	13	(NOT src) OR dst
GXnand	14	(NOT src) OR (NOT dst)
GXset	15	1

Set_Write_Mask – Sets the write mask to the specified value. For an *n*-bit system, the write mask can range from 0 to $2^n - 1$. For more information on the write mask, refer to *Using the Write Mask and Graphics Functions to Manipulate Color* on page A-93.

Static_Color – When specified with a *Depth* value, selects the StaticColor visual with *Depth* bits per pixel. This keyword has effect only if no windows have been created.

Static_Gray – When specified with a *Depth* value, selects the StaticGray visual with *Depth* bits per pixel. This keyword has effect only if no windows have been created.

Threshold — Specifies use of the threshold dithering algorithm — the simplest dithering method. For more information on this dithering method, see *Displaying Images on Monochrome Devices* on page 148.

Translation — Using the shared colormap (which is normally recommended) causes PV-WAVE to translate between PV-WAVE color indices (which always start with zero and are contiguous) and the pixel values actually present in the display. The *Translation* keyword specifies the name of a variable to receive the translation vector. To read the translation table:

```
DEVICE, Translation=Transarr
```

The result is a 256-element byte vector, `Transarr`. Element zero of `Transarr` contains the pixel value allocated for the first color in the PV-WAVE colormap, and so forth.

For more information on the translation table, refer to *PV-WAVE's Color Translation Table* on page A-84.

True_Color — When specified with a *Depth* value, selects the TrueColor visual with *Depth* bits per pixel. For example:

```
True_Color=12
```

The *True_Color* keyword has effect only if no windows have been created.

Window_State — Returns an array containing values indicating the status (open = 1, closed = 0) for all available PV-WAVE windows. For example:

```
WINDOW, 3
```

Open window 3.

```
DEVICE, Window_State=winarray
PRINT, winarray
```

X Window Visuals

Visuals specify how the hardware deals with color. The X server of your display may provide colors or only gray scale (black and white), or both. The color tables may be changeable from within PV-WAVE (read-write), or they may be static (read-only). The value of each pixel may be mapped to any color (Undecomposed Colormap), or certain bits of each pixel are dedicated to the red, green, and blue primary colors (Decomposed Colormap).

The X server provides six visual classes – read-write and read-only visuals for three types of displays: Gray Scale, Undecomposed Color, and Decomposed Color. The names of the visual classes are listed in Table A-21:

Table A-21: X Window System Visual Classes

Visual Class Name	Writable	Description
StaticGray	no	Gray scale
GrayScale	yes	Gray scale
StaticColor	no	Undecomposed color
PseudoColor	yes	Undecomposed color
TrueColor	no	Decomposed color
DirectColor	yes	Decomposed color

PV-WAVE supports all six types of visual classes, although not at all possible depths (e.g., 4-bit, 8-bit, 24-bit).

Note For more information on the differences between pseudo color and 24-bit ("true") color, see *Not all Color Images are True-color Images* on page 146.

X servers from different manufacturers will each have a default visual class. Many servers may provide multiple visual classes. For example, a server with display hardware that supports an 8-bit deep, undecomposed, writable color map (PseudoColor), may also easily support StaticColor, StaticGray, and GrayScale visual classes.

Tip For more detailed information about X visual classes, refer to *Volume 1* of the *Xlib Programming Manual, Second Edition,* O'Reilly & Associates, Inc., Sebastopol, CA, 1990.

Selecting a Visual Class

When opening the display, PV-WAVE asks the display for the following visual classes, in order, until a supported visual class is found:

1. DirectColor, 24-bit
2. TrueColor, 24-bit
3. PseudoColor, 8-bit, then 4-bit
4. StaticColor, 8-bit, then 4-bit
5. GrayScale, any depth
6. StaticGray, any depth

You can override this default choice by using the DEVICE procedure to specify the desired visual class and depth before you create a window.

For example, if you are using a display that supports both the 24-bit deep DirectColor visual class, and an 8-bit deep PseudoColor visual class, PV-WAVE will select the 24-bit deep DirectColor visual class. To use PseudoColor, enter the following command before creating a window:

```
DEVICE, Pseudo_Color=8
```

Note If a visual type and depth is specified, using the DEVICE procedure, and it does not match the default visual class of the screen, a new colormap is created.

Colormapped Graphics

Colormaps define the mapping from color index to screen pixel value. In this discussion, the term *colormap* is used interchangeably with the terms *color table*, *color translation table*, or *color lookup table* – other terminology that you may be familiar with from working with other systems. Colormaps perform a slightly different role on 8-bit workstations than they do on 24-bit workstations.

8-bit Graphics Primer

On an 8-bit workstation, the screen pixel value in video memory "looks up" the red-green-blue color combination in its corresponding color table index (hence, the term *color lookup table*). For example, a pixel value of 43 looks in color table location 43 and may find Red=255, Green=0, and Blue=255. The three values at that colortable location tell the red, green, and blue electron guns in the CRT what intensities to display for that pixel. In this example, any pixel with a value of 43 will be displayed as magenta (100% red, 0% green, 100% blue).

24-bit Graphics Primer

On a 24-bit workstation, each pixel on the screen can be displayed in any one of a possible 16.7 million (2^{24}) colors. The video memory on the machine is capable of addressing each pixel on the screen with a 24-bit value, "decomposed" to eight bits each for the red, green, and blue intensity values for that pixel.

Since each pixel in video memory directly references a set of three 8-bit red-green-blue intensities, there is no need for a color lookup table as in an 8-bit system. However, because PV-WAVE is an 8-bit colortable-based software package (instead of a "true" 24-bit software package), it performs similar conversions internally when drawing to a 24-bit display.

Some 24-bit displays allow the screen to be treated as two separate 12-bit PseudoColor visuals. This allows for "double-buffering", a technique useful for animation, or for storing distance data to simplify hidden line and plane calculations in 3D applications.

Although a 24-bit display takes up three times as much video memory as an 8-bit system, the number of concurrent colors allowed on the screen is not limited by memory – the number is limited only by the number of pixels on the screen (assuming, of course, that the screen contains less than 16.7 million pixels). For example, a 24-bit X Window display with 1280-by-1024 pixel resolution contains 1,310,720 pixels, and thus it can potentially display that many colors on the screen at one time.

Note To see a formula for calculating how many colors a display is capable of displaying, given the number of bits it has for describing each pixel, refer to *How RGB Color Triples Map into Pixels* on page 307.

For more information about how colors are represented on 24-bit displays, refer to *Understanding 24-bit Graphics Displays* on page A-90.

How PV-WAVE Allocates the Colormap

Many factors affect how PV-WAVE chooses the type of colormap. For example, the keywords you supply with the DEVICE procedure control the way color is used in PV-WAVE graphics windows throughout that session.

PV-WAVE colormaps can be either shared or private, and either read-write or read-only:

- Colormaps may be private or shared. This characteristic is determined by the number and type of application(s) running during your session.
- Colormaps may be static (read-only) or writable. This characteristic is controlled by the visual class that was invoked at the time the X server was started (or restarted).

For more information about how to select PV-WAVE's visual class, refer to *Selecting a Visual Class* on page A-78.

Shared Colormaps

The window manager creates a colormap when it is started. This is known as the default colormap, and it can be shared by all applications using the display. When any application requires a colormap entry, it can allocate one from this *shared colormap.*

Advantages

Using the shared colormap ensures that all applications share the available colors without conflict, and all color indices are available to every application. No application is allowed to change a color that is allocated to another application.

In other words, PV-WAVE can change the colors it has allocated without changing the colors that have been allocated by the window manager or other applications that are running.

Disadvantages

On the other hand, using a shared colormap can involve the following disadvantages:

- The window system's interface routines must translate between internal pixel values and the values required by the server, significantly slowing the transfer of images.
- The number of available colors in the shared colormap depends on the window manager being used and the demands of other applications. Thus, the number of available colors can vary, and the shared colormap might not always have enough colors available to perform the desired PV-WAVE operations.
- The allocated colors in a shared colormap do not generally start at 0 (zero) and they are not necessarily contiguous. This makes it difficult to use the write mask for certain operations.

For more information about the write mask, refer to *Using the Write Mask and Graphics Functions to Manipulate Color* on page A-93.

Private Colormaps

An application can create its own *private colormap.* Most hardware can only display a single colormap at a time, so these private colormaps are called virtual colormaps, and only one at a time is visible. When the window manager gives the input focus to a window with a private colormap, the X server loads its virtual colormap into the hardware colormap.

Advantages

The advantages of private colormaps include:

- Every color supported by the hardware is available to PV-WAVE, improving the quality of images.
- Allocated colormaps always start at 0 (zero) and use contiguous colors, which simplifies your use of the write mask.
- No translation between internal pixel values and the values required by the server is required, which optimizes the transfer of images.

Disadvantages

On the other hand, using a private colormap can involve the following disadvantages:

- You may see "flashing" when you move the pointer in and out of PV-WAVE graphics windows. This happens because when the pointer moves inside a PV-WAVE window, the PV-WAVE colormap is loaded, and other applications are displayed using PV-WAVE's colors. The situation is reversed when the pointer moves out of the PV-WAVE window into an area under the jurisdiction of a different application.
- Colors in a private colormap are usually allocated from the lower end of the map first. But these typically are the colors allocated by the window manager for such things as window borders, the color of text, and so forth.

  Since most PV-WAVE colormaps have very dark colors in the lower indices, the result of having other applications use the PV-WAVE colormap is that the portions of the screen that are not PV-WAVE graphics windows look dark and unreadable.

When Color Characteristics are Determined

PV-WAVE decides how many colors and which combination of colormap and visual class to use when it creates its first graphics window of that session. You can create windows in two ways:

- Use the WINDOW procedure. WINDOW allows you to explicitly control many aspects of how the window is created, including its X visual class.
- If no windows exist and a graphics operation requiring a window is executed, PV-WAVE implicitly creates a window (window 0) using the default characteristics.

Once the first window is created, all subsequent PV-WAVE windows share the same colormap. The number of colors available is stored in the system variable !D.N_Colors. For more information about !D.N_Colors, see the section entitled *Determining the Number of Available Plot Colors* on page 322.

For more information about when color characteristics are determined when drawing or plotting to a window running in a 24-bit visual class, refer to *24-bit Visual Classes Supported by PV-WAVE* on page A-88.

Closing the Connection to the X Server to Reset Colors

To change the type of colormap used or the number of colors, you must first completely close the existing connection to the X server using the following command:

```
DEVICE, /Close_Display
```

You can then use the WINDOW procedure to specify the new configuration. However, remember that if you enter the command shown above, it will cause every PV-WAVE graphics window that is currently open to be deleted.

Tip Another way to close the connection to the X server is to delete every PV-WAVE graphics window that is currently open; this automatically closes the connection.

How Many Colors PV-WAVE Maps into the Color Table

If the number of colors to use is explicitly specified using the *Colors* keyword with the WINDOW procedure, PV-WAVE attempts to allocate the number of colors specified from the shared colormap using the default visual class of the screen. If enough colors aren't available, a private colormap with that number of colors is created instead.

For more information about the advantages and disadvantages of private colormaps, refer to *Private Colormaps* on page A-81.

By default, the shared colormap is used whenever possible, and PV-WAVE allocates all available colors from the shared colormap. The allocation occurs if no window currently exists and a graphics operation causes PV-WAVE to implicitly create one.

Reserving Colors for Other Applications' Use

Specifying a negative value for the *Colors* keyword to the WINDOW procedure causes PV-WAVE to use the shared colormap, allocating all but the specified number of colors. For example:

```
WINDOW, Colors = -16
```

allocates all but eight of the currently available colors. This allows other applications that might need their own colors (such as the window manager) to run in tandem with PV-WAVE.

PV-WAVE's Color Translation Table

Colors in the shared colormap do not have to start from index zero, nor are they necessarily contiguous. PV-WAVE preserves the illusion of a zero-based contiguous colormap by maintaining a translation table between applications' color indices and the actual pixel values allocated from the X server. The color indices range from 0 to !D.N_Colors – 1. Normally, you need not be concerned with this translation table, but it is available using the statement:

```
DEVICE, Translation=Trans
```

This statement stores the current translation table, a 256-element byte vector, in the variable `Trans`. Element zero of this translation vector contains the value pixel allocated for the zeroth color in the PV-WAVE colormap, and so forth.

Tip In the case of a private colormap, each element of the translation vector contains its own index value, because private colormaps start at zero and are contiguous.

The translation table may be bypassed, allowing direct access to the display's color hardware pixel values, by specifying the *Bypass_Translation* keyword with the DEVICE procedure. Translation is disabled by clearing the bypass flag, as shown in the following command:

```
DEVICE, Bypass_Translation=0
```

When a private or static (read-only) color table is initialized, the bypass flag is cleared. The bypass flag is always set when initializing a shared color table.

For more information about !D.N_Colors, refer to *Determining the Number of Available Plot Colors* on page 322.

To see an example of how the translation table can affect displayed colors, see *Interaction Between the Set_Graphics_Function Keyword and Hardware Pixel Values* on page A-96.

Using Pixmaps to Improve Application Performance

The X Window System can direct graphics to either windows or pixmaps.

- **Windows** – Windows are the usual windows that appear on the screen and contain graphics. Drawing to a window produces a viewable result.
- **Pixmaps** – Pixmaps are areas of invisible graphics memory contained in the server. Drawing to a pixmap simply updates the pixmap memory.

Pixmaps are useful because it is possible to write graphics to a pixmap and then copy the contents of the pixmap to a window where it can be viewed. This copy operation is very fast because it happens entirely within the server, instead of communicating across the network to the client. Provided enough pixmap memory is available for the server's use, this technique works very well for animating a series of images. The process works by placing the desired images into pixmap memory and then copying them to a visible window.

To read a brief description of the relationship between the X server and the X client, refer to *X Window System* on page A-69.

Creating a Pixmap

To create a pixmap, use the *Pixmap* keyword with the WINDOW procedure. For example, to create a 128-by-128 pixmap in PV-WAVE window 1:

```
WINDOW, 1, /Pixmap, Xsize=128, Ysize=128
```

Once they are created, pixmaps are treated just like normal windows, although some operations (WSHOW, for instance) don't have any noticeable effect when applied to a pixmap.

Example — Animating a Series of Pixmap Images

The following example shows how animation can be performed using pixmap memory. It uses a series of 15 heart images taken from the file `heartbeat.dat`. (This file is located in the `data` subdirectory of the main PV-WAVE directory.) It creates a pixmap and writes all 15 images to it. It then uses the *Copy* keyword of the DEVICE procedure to copy the images to a visible window. Pressing any key causes the animation to halt:

```
PRO animate_heart
```

This procedure animates a series of heart data.

```
IF !Version.platform EQ 'VMS' THEN $
   OPENR, u, GETENV('WAVE_DIR')+$
   '[data]heartbeat.dat', /Get_Lun $

ELSE $
OPENR, u, '$WAVE_DIR/data/heartbeat.dat', $
   /Get_Lun
```

Open the file containing the images.

```
frame = ASSOC(u, BYTARR(256, 256))
```

Associate a file variable with the file. Each heart image is 256-by-256 pixels.

```
WINDOW, 0, /Pixmap, Xsize=7680, Ysize=512
```

Window 0 is a pixmap that is one double-sized image tall and 15 double-sized images wide. The images will be placed in this pixmap.

```
FOR i = 0, 15-1 DO TV, $
    REBIN(SMOOTH(frame(i), 3), 512, 512), i
```

Write each image to pixmap memory. SMOOTH is used to improve the appearance of each image and REBIN is used to enlarge/shrink each image to the final display size.

```
FREE_LUN, u
```

Close the image file and free the file unit.

```
WINDOW, 1, Xsize=512, Ysize=512, Title='Heart'
```

Window 1 is a visible window that is used to display the animated data, creating the appearance of a beating heart.

```
i = 0L
```

Load the current frame number.

```
WHILE GET_KBRD(0) EQ '' DO BEGIN
    DEVICE, Copy=[i * 512, 0, 512, 512, 0, 0, 0]
```

Display frames until any key is pressed. Copy the next image from the pixmap to the visible window.

```
    i = (i + 1) MOD 15
```

Keep track of total frame count.

```
ENDWHILE

END
```

In this example, the pixmap was made one image tall and 15 images wide for simplicity. However, some X servers will refuse to display a pixmap that is wider than the physical width of the screen. For this case, the above routine would have to be rewritten to either use a shorter, taller pixmap or to use several pixmaps.

24-bit Visual Classes Supported by PV-WAVE

If you do not request a visual class (by entering a DEVICE command prior to opening the first graphics window), PV-WAVE uses the first class the device supports, regardless of the root visual class. PV-WAVE queries the device about the following visual classes, in order, until a supported visual class is found:

1. DirectColor, 24-bit
2. TrueColor, 24-bit
3. PseudoColor, 8-bit, then 4-bit
4. StaticColor, 8-bit, then 4-bit
5. GrayScale, any depth
6. StaticGray, any depth

PV-WAVE can operate in either of the two 24-bit visual classes. But because of the search order, if your workstation is configured to run in a 24-bit mode, PV-WAVE will choose DirectColor by default.

For more discussion concerning the search order for X visual classes, refer to *Selecting a Visual Class* on page A-78.

DirectColor Mode

In DirectColor mode, color tables are always in effect and can be loaded. PV-WAVE loads the color tables using the same LOADCT and TVLCT commands you use when using PV-WAVE in 8-bit PseudoColor mode.

In this mode, color data is *decompressed*, meaning there are still only 256 slots in the PV-WAVE color table while operating in DirectColor mode. When color data is decompressed, 8 of the bits map to one of the 256 reds, 8 of the bits map to one of the 256 greens, and 8 of the bits map to one of the 256 blues that have been loaded into the 256 element color table (with LOADCT or TVLCT).

For more information about which bits are mapped to red, green, and blue, refer to *Specifying 24-bit Colors in Hexadecimal Notation* on page A-91.

TrueColor Mode

In TrueColor mode, color tables cannot be loaded. However, PV-WAVE can load a translation table. You load the translation table using either the LOADCT or TVLCT commands. The translation table works just like a color table except that it takes effect at drawing time (TV, PLOT, CONTOUR, etc.) rather than when you enter the LOADCT or TVLCT command.

Example — Using PV-WAVE in TrueColor Mode

In TrueColor mode, the color map takes effect when the graphics command (PLOT, SURFACE, TVSCL, etc.) is entered. An intriguing benefit of this behavior is that several PV-WAVE color tables loaded into translation tables (using LOADCT or TVLCT) may be used simultaneously during the same PV-WAVE session, as shown in the following group of commands:

```
DEVICE, True_Color=24
```

Define a PV-WAVE TrueColor graphics window.

```
data = DIST(512)
```

Create a sample data set that can be displayed as an image.

```
WINDOW, 0
LOADCT, 0
TVSCL, data
```

Display the image in window 0 using grayish hues.

```
WINDOW, 1
LOADCT, 1
TVSCL, data
```

Display the image in window 1 using bluish hues.

```
WINDOW, 2
LOADCT, 4
TVSCL, data
```

Display the image in window 2 using shades of red, yellow, and green.

If the root window is also running in the 24-bit TrueColor visual class (just like PV-WAVE in this example), you will not see any "flashing", even though you are entering LOADCT and TVSCL commands. For more information about the condition known as "window flashing", refer to *Private Colormaps* on page A-81.

PV-WAVE Does Not Inherit the X Visual Class

PV-WAVE does not inherit the visual class of the X Window System root window. Thus, booting a machine with the root window set to any specific visual class has no effect on the visual classes that are available for use by PV-WAVE (see the Note later in this section for an exception to this statement). For example, you could edit the `/etc/ttys` file of a DEC Ultrix workstation (if it supported 24-bit visual classes) such that it included the line:

```
0: window = '/usr/bin/Xtm -class StaticGray'
```

and then proceed to simultaneously (and successfully) run two PV-WAVE sessions – one in DirectColor mode and the other in PseudoColor mode.

Note Windows that are created with calls to WAVE Widgets or Widget Toolbox functions and procedures are treated differently. These windows *do* inherit the X visual class from their top-level shell, which in turn, inherits the X visual class from the root window. In this sense, windows that are part of a graphical user interface (GUI) are different from ordinary PV-WAVE graphics windows.

Understanding 24-bit Graphics Displays

A 24-bit raster image is actually made up of three component 8-bit images – a red, a green, and a blue image, which combine to create a "true" color picture. With 24-bit graphics displays, each pixel on the screen can be displayed in any one of a possible 16.7 million (2^{24}) colors.

The video memory in a 24-bit machine is capable of addressing each pixel on the screen with 8 bits assigned to each of the red, green, and blue intensity values for that pixel. These sets of 8-bit values are known as *color planes*, e.g., the "red plane", the "green plane", and the "blue plane". The three red-green-blue intensity values for each 24-bit pixel in video memory are translated directly into color intensities for the red, green, and blue electron guns in the CRT.

Specifying 24-bit Colors in Hexadecimal Notation

Colors in PV-WAVE graphics windows can be specified using the *Color* keyword. In the range of {0...16,777,216}, the color magenta has a decimal value of 16,711,935. But when using 24-bit color, the convention is to represent this value not as a decimal value, but as a 6-digit hexadecimal value. For example, the color magenta can be passed to one of the graphics routines using the construct:

```
Color = 'ff00ff'x
```

The first two digits in this hexadecimal value correspond to the blue intensity, the middle two digits correspond to the green intensity, and the right two digits correspond to the red intensity. The interpretation of the various digits is shown in Figure A-5.

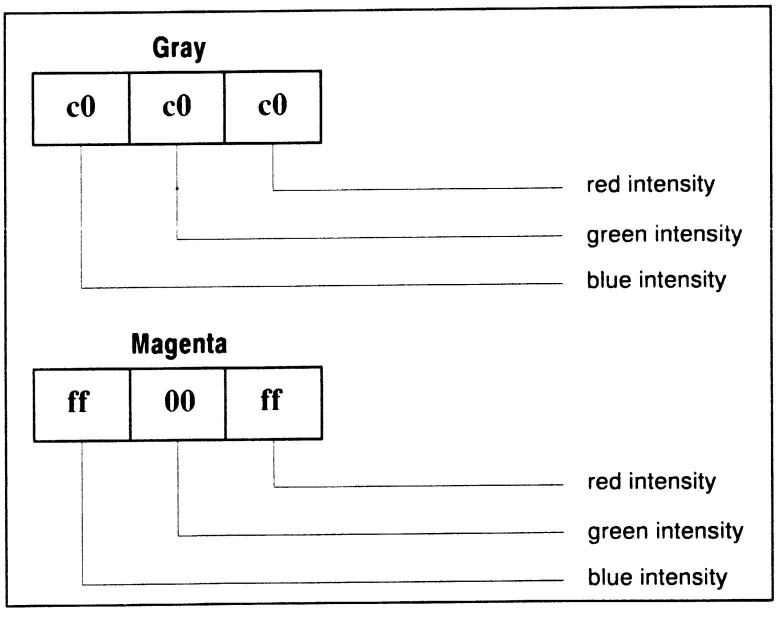


Figure A-5 Hexadecimal notation can be used to represent 24-bit numbers; each two digits describes either the red, green, or blue intensity. The top color shown in this figure is a light gray, since the red, green, and blue components are all set to an equal intensity. The lower color shown is the color magenta, where red and green are both set to full intensity (ff), but the color green is essentially turned "off" by setting it equal to zero (00).

The hexadecimal notations for some of the most commonly-used colors are shown in Table A-22.

Table A-22: Hexadecimal Notation for Commonly-used Colors

Color	Hexadecimal Notation
Black	'000000'x
White	'ffffff'x
Red	'0000ff'x
Green	'00ff00'x
Blue	'ff0000'x
Cyan	'ffff00'x
Magenta	'ff00ff'x
Yellow	'00ffff'x
Medium Gray	'7f7f7f'x

Note When programming with WAVE Widgets or Widget Toolbox, you can only enter colors using colornames. The hexadecimal form is not recognizable by the WAVE Widgets or Widget Toolbox routines. For more information about how to select colors for widgets in a graphical user interface (GUI), refer to *Setting Colors* on page 463 of the *PV-WAVE Programmer's Guide.*

Specifying 24-bit Plot Colors

In the TrueColor visual class, raster graphics colors go through the translation table, but vector graphics colors do not. This means that vector graphics colors in PV-WAVE plots can be specified explicitly despite any translation table that has been loaded.

For more information about the translation table, refer to *PV-WAVE's Color Translation Table* on page A-84.

Example — Plotting with 24-bit Colors

The following example plots a line chart showing five data sets, each one plotted in a different color. (Assume that `mydata1`, `mydata2`, `mydata3`, `mydata4`, and `mydata5` have all been defined as integers or floating-point vectors prior to entering the commands shown below.)

```
DEVICE, True_Color=24
```

Define a PV-WAVE TrueColor graphics window.

```
PLOT, mydata1, Color='00ff00'x
```

Draw mydata1 using a green line.

```
OPLOT, mydata2, Color='0000ff'x
```

Draw mydata2 using a red line.

```
OPLOT, mydata3, Color='ff0000'x
```

Draw mydata3 using a blue line.

```
OPLOT, mydata4, Color='00ffff'x
```

Draw mydata4 using a yellow line.

```
OPLOT, mydata5, Color='007fff'x
```

Draw mydata5 using a orange line.

Note The PV-WAVE system variable !D.N_Colors must still be initialized properly prior to opening the graphics window. For more information about how to initialize color characteristics, refer to *When Color Characteristics are Determined* on page A-82.

Using the Write Mask and Graphics Functions to Manipulate Color

If you are using PV-WAVE in one of the 24-bit visual classes, you may want to consider using the write mask to isolate a certain group of bits, such as the red group of bits, or the green group. This is relatively easy to do, since each pixel in video memory directly references a set of three 8-bit red-green-blue intensities. For more information about how to address the various planes of a 24-bit (24-plane) workstation, refer to *Understanding 24-bit Graphics Displays* on page A-90.

In an 8-bit visual class, an analogous write mask operation is to use a write mask of 1 so that only the "lowest" plane is affected. But this is a suitable choice only if you want to force your application to display in monochrome on both color and monochrome displays.

Tip For best results when using the write mask, use a color table that uses all 256 available colors, and bypass the PV-WAVE translation table to make sure the color table starts at zero (0). Unfortunately, a side effect of letting PV-WAVE allocate all 256 colors is that you may see "window flashing" when using your application, as explained in *Private Colormaps* on page A-81.

Using the Write Mask to Create Special Effects

The write mask can be used to superimpose (overlay) one graphics pattern over another when plotting to a PV-WAVE graphics window, allowing you to create special effects. For example, some 24-bit displays allow the screen to be treated as two separate 12-bit Pseudo_Color visuals. This allows for "double-buffering", a technique useful for animation, or for storing distance data to simplify hidden line and plane calculations in 3D applications.

Another possible application for the write mask is to simultaneously manage two 4-bit-deep images in a single PV-WAVE graphics window instead of a single 8-bit-deep image. You could use the write mask to control whether the current graphics operation operates on the "top" image or the "bottom" image.

Using Graphics Functions to Manipulate Color

PV-WAVE's X driver provides two keywords for inquiring and manipulating the graphics function – *Get_Graphics_Function* and *Set_Graphics_Function.*

The value of the *Set_Graphics_Function* keyword controls the logical graphics function; this function specifies how the source pixel values generated by a graphics operation are combined with the pixel values already present on the screen. To see a complete list of graphics function codes, refer to Table A-20 on page A-75.

For example, the following code segment shows how to use the XOR graphics function to toggle the "low bit" of the pixel value that determines the color in a rectangle defined by its diagonal corners (x_0, y_0) and (x_1, y_1):

```
DEVICE, Get_Graphics=oldg, Set_Graphics=6
```

Set graphics function to exclusive or (GXxor), saving the old function.

```
POLYFILL, [[x0,y0], [x0,y1], [x1,y1], $
   [x1,y0]], /Device, Color=1
```

Use POLYFILL to select the area to be inverted, and immediately XOR a pixel value of 1 with the image currently displayed in that area. XORing every pixel with the binary equivalent of 1 ensures that only the lowest bit of the color is affected.

```
DEVICE, Set_Graphics_Function=oldg
```

Restore the previous graphics function.

The default value for the *Set_Graphics_Function* keyword is `GXcopy`, which means that the source graphics from the current operation get copied into the window, destroying any graphics that were previously displayed there.

Interaction Between the Set_Write_Mask and the Set_Graphics_Function Keywords

You use the *Set_Write_Mask* keyword to specify the planes whose bits you are manipulating or the plane you want to use for the special effects. The way the two graphics patterns are combined depends on the value you provide for the *Set_Graphics_Function* keyword. For example, you could use the following commands:

```
DEVICE, Set_Graphics_Function=6

DEVICE, Set_Write_Mask=8
```

to extract the fourth bit (the binary equivalent of the decimal value 8) of the image in the current graphics window. The extracted plane is XORed (XOR is the graphics function specified by setting the *Set_Graphics_Function* keyword equal to 6) with the source pattern (the result of the current graphics operation). After the graphics function is implemented, the result is drawn in the current graphics window using whichever color(s) in the color table match the resultant value(s).

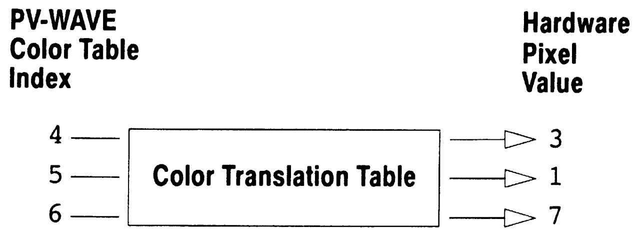
Interaction Between the Set_Graphics_Function Keyword and Hardware Pixel Values

The graphics functions specified by the *Set_Graphics_Function* keyword operate on hardware pixel values. Unless the translation table is bypassed, like it is when displays with static (read-only) visual classes and private color tables are used, a PV-WAVE color table index that represents a certain color will likely differ in value from the hardware pixel value that represents the same color. This can produce unexpected colors when you use the GX graphics functions.

An easy way to avoid getting such unexpected colors is to use PV-WAVE's Boolean OR, AND, XOR, and NOT operators, rather than the GX graphics functions. For details, see the following example.

Example — Understanding the Colors that are Produced by the GX Graphics Functions

Suppose that you are using your X display in one of the 8-bit visual classes, and a simple color translation table has been defined in the following way:



Now enter the following PV-WAVE commands:

```
data = [1, 2, 3]
```

Define a simple dataset to experiment with.

```
GXand = 1
PLOT, data, Color=5
```

The data is plotted in hardware color 1, because color 5 gets translated to hardware pixel value 1.

```
PLOT, data, Color=(5 AND 6)
```

The data is plotted in hardware color 3, because 5 ANDed with 6 equals 4, which then gets translated to hardware pixel value 3.

```
DEVICE, Set_Graphics_Function=GXand
PLOT, data, Color=6
```

The data is plotted in hardware color 1, because color 5 gets translated to hardware pixel value 1 and color 6 gets translated to hardware pixel value 7. Then, in hardware pixel values, 7 is ANDed with 1 and the result equals 1.

Tip You can check the values in the current translation table using the *Translation* keyword; this keyword specifies the name of a variable to receive the translation vector. To read the translation table, enter this command:

```
DEVICE, Translation=Transarr
```

The result is a 256-element byte vector, `Transarr`. Element zero of `Transarr` contains the pixel value allocated for the first color in the PV-WAVE colormap, and so forth.

X Window IDs

PV-WAVE provides methods for getting and setting X Window IDs for any PV-WAVE window. PV-WAVE also supports methods for setting and getting X Pixmap IDs for PV-WAVE windows.

To set the X Window ID for a PV-WAVE window, use the *Set_Xwin_Id* keyword with the WINDOW procedure. The X Window ID must be a valid, existing ID for the X server that PV-WAVE is using. When the *Set_Xwin_Id* keyword is used, PV-WAVE uses the X window associated with the ID; PV-WAVE does not create a new window. The programmer is responsible for synchronizing the use of this window by the two programs.

Note If the X Window ID is from another program, PV-WAVE color table changes may not affect the window.

To get the X Window ID for a PV-WAVE window, use the *Get_Win_ID* keyword with the WINDOW procedure. The X Window ID returned from the WINDOW procedure may be passed to another program. The other program may then write into the PV-WAVE window. The programmer is responsible for synchronizing the use of this window by the two programs.

Similarly, the *Get_Xpix_Id* keyword can be used to get the X Pixmap ID.

Caution The WDELETE procedure will delete all windows sharing a common X Window ID. For example:

```
WINDOW, 0, Get_Xwin_Id=New_Xwin_Id
WINDOW, 1, Set_Xwin_Id=New_Xwin_Id
WDELETE, 1
```

The command `WDELETE, 1` deletes both windows 1 and 0.

Z-buffer Output

The Z-buffer allows you to create complex 3D plots, image warping to polygons, and transparency effects without special hardware.

To direct graphics output to the Z-buffer, enter the command:

```
SET_PLOT, 'Z'
```

This causes PV-WAVE to use the Z-buffer driver for producing graphical output. Once the Z-buffer driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described in *Controlling Z-buffer Output with DEVICE Keywords* on page A-99.

Note Use `INFO, /Device` to view the driver's current settings.

Controlling Z-buffer Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control of the Z-buffer driver:

Close – Deallocates the memory used by the buffers. The Z-buffer device is reinitialized if subsequent graphics operations are directed to the device.

Get_Graphics_Function – See the description of the *Get_Graphics_Function* keyword in *Controlling the X Driver with DEVICE Keywords* on page A-71.

Get_Write_Mask – See the description of the *Get_Write_Mask* keyword in *Controlling the X Driver with DEVICE Keywords* on page A-71.

Set_Character_Size – A two-element vector that changes the standard width and height of the vector-drawn fonts. The first element in the vector contains the new character width, and the second element contains the height. By default, characters are approximately 8-pixels wide, with 12 pixels between lines.

Set_Colors – Sets the number of pixel values, !D.N_Colors. This value is used by a number of PV-WAVE routines to determine the scaling of pixel data and the default drawing index. Allowable values range from 2 to 256, and the default value is 256. Use this parameter to make the Z-buffer device compatible with devices with fewer than 256 color indices.

Set_Graphics_Function – See the description of the *Set_Graphics_Function* keyword in *Controlling the X Driver with DEVICE Keywords* on page A-71. The Z-buffer allows you to use all graphics functions supported by the X driver.

Set_Resolution – Two-element vector that sets the width and height of the buffers. The default size is 640-by-512. If this size is not the same as the existing buffers, the current buffers are destroyed and the device is reinitialized.

Set_Write_Mask – See the description of the *Set_Write_Mask* keyword in *Controlling the X Driver with DEVICE Keywords* on page A-71.

Z-buffer Examples

Example 1

This example demonstrates how graphics can be rendered in memory (into the Z-buffer) and then later displayed. The resulting image is shown in Figure A-6.

```
SET_PLOT, 'z'

SHADE_SURF, HANNING(23,23)

SURFACE, HANNING(23,23), /Noerase

img = TVRD(0,0,640,512)

SET_PLOT, 'x'

TV, img
```

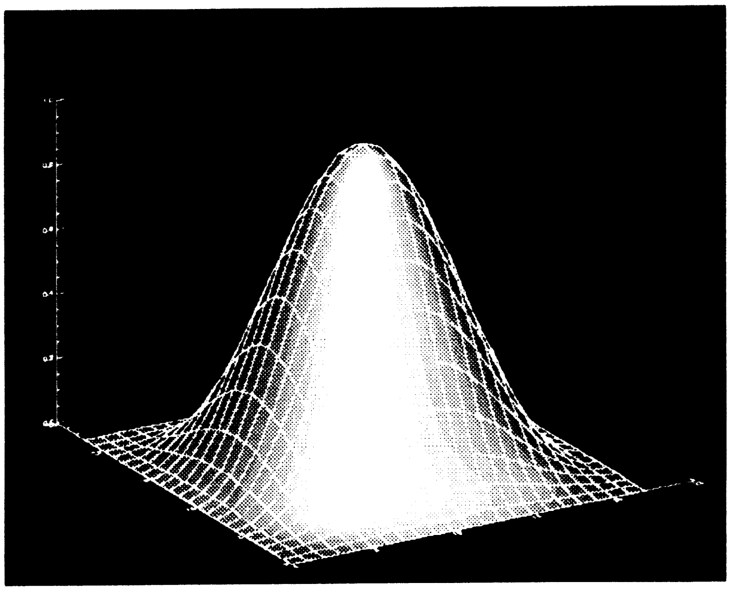


Figure A-6 Resulting image from Example 1.

Example 2

This example creates a single image composed of two intersecting objects drawn with hidden surfaces removed. This effect can only be accomplished with the Z-buffer. The resulting image is shown in Figure A-6.

```
im = BYTARR(512,512)
IF !Version.platform EQ 'VMS' THEN $
   OPENR, 1, GETENV('WAVE_DIR')+$
   '[data]photo.img', /Get_lun $
ELSE $
OPENR, 1, '$WAVE_DIR/data/photo.img', $
   /Get_lun
```

Open the file containing the image.

```
READU, 1, im
CLOSE, 1
SET_PLOT, 'z'
ERASE
```

Erase the Z-buffer in case there was something in there before.

```
SHADE_SURF, HANNING(23,23), /Noerase
```

Draw a surface.

```
verts=[ [0,0,1], [1,0,0], [1,1,0], [0,1,1] ]
```

Create a 3D quadrilateral.

```
scale3, Xrange = [-0.2,1.2], $
   Yrange = [-0.2,1.2], Zrange = [-0.2,1.2],$
   Ax=110, Az=10
```

Set up a 3D viewing transform.

```
POLYFILL, Verts, Pattern = im, $
   Image_coord=[[0,0], [511,0], [511,511],$
   [0,511] ], /T3d
```

Draw the transformed quadrilateral with an image mapped onto it.

```
res = TVRD(0,0,640,512)
SET_PLOT, 'x'
TV, res
```

Notice how the two objects intersect.

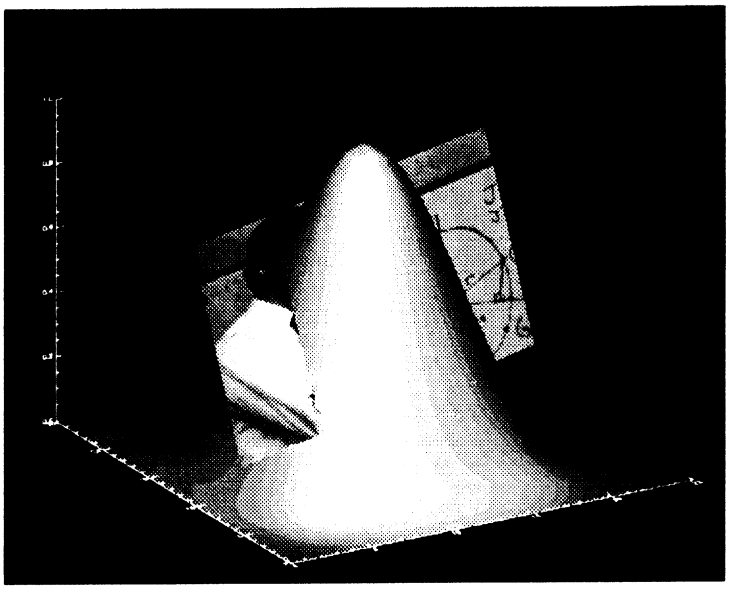


Figure A-7 Resulting image from Example 2.

Example 3

In this example, the same image from the previous example is created; however, this time maximum intensity projection is used to produce a transparency effect. The resulting image is shown in Figure A-6.

```
im = BYTARR(512,512)
IF !Version.platform EQ 'VMS' THEN $
   OPENR, 1, GETENV('WAVE_DIR')+$
   '[data]photo.img', /Get_lun $

ELSE $
```

```
OPENR, 1, '$WAVE_DIR/data/photo.img', $
   /Get_lun
```

Open the file containing the image.

```
READU, 1, im
CLOSE, 1
SET_PLOT, 'z'
ERASE
```

Erase the Z-buffer in case there was something in there before.

```
SHADE_SURF, HANNING(23,23), /Noerase
```

Draw a surface.

```
verts=[ [0,0,1], [1,0,0], [1,1,0], [0,1,1] ]
```

Create a 3D quadrilateral.

```
SCALE3, Xrange = [-0.2,1.2], $
   Yrange = [-0.2,1.2], Zrange = [-0.2,1.2], $
   Ax = 110, Az = 10
```

Set up a 3D viewing transform.

```
POLYFILL, Verts, Pattern = im, $
   Image_coord =[ [0,0], [511,0], [511,511], $
   [0,511] ], /T3d, /Mip
```

Draw the transformed quadrilateral with an image mapped onto it. Use the Mip keyword to specify maximum intensity projection (transparency).

```
res = TVRD(0,0,640,512)
SET_PLOT, 'x'
TV, res
```

Notice how the two objects interact this time. You can partially see through the surface to the image passing through it.

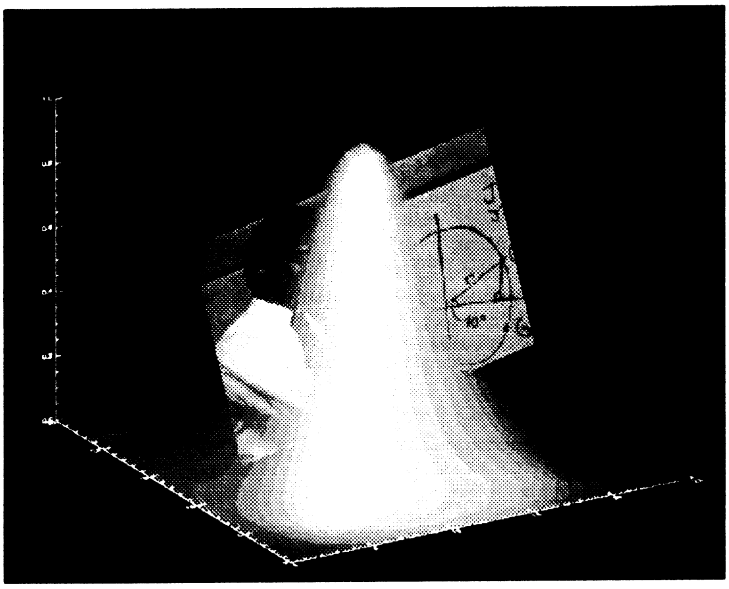


Figure A-8 Resulting image from Example 3.

B

APPENDIX

Picture Index

2D Plots

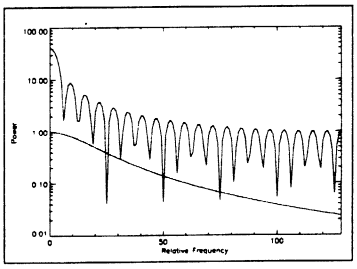


Figure 3-13 on page 85 of the *PV*‑*WAVE User's Guide*.

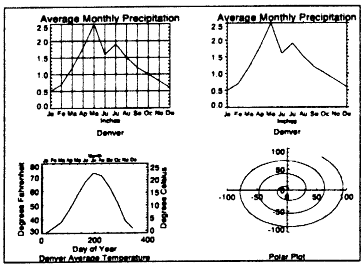


Figure 3-12 on page 84 of the *PV*‑*WAVE User's Guide*.

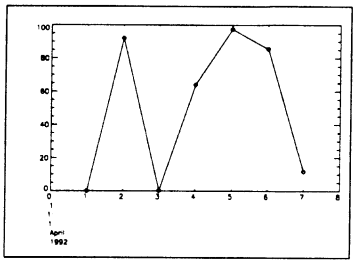


Figure 7-2 on page 245 of the *PV-WAVE User's Guide*.

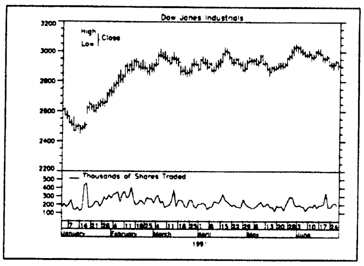


Figure 7-1 on page 222 of the *PV-WAVE User's Guide*.

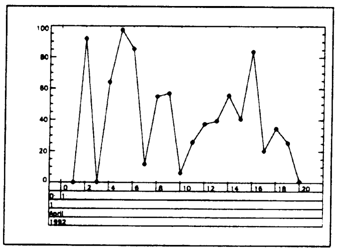


Figure 7-3 on page 246 of the *PV-WAVE User's Guide*.

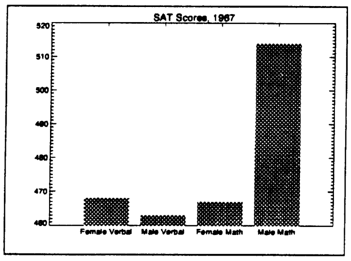


Figure 3-10 on page 80 of the *PV-WAVE User's Guide*.

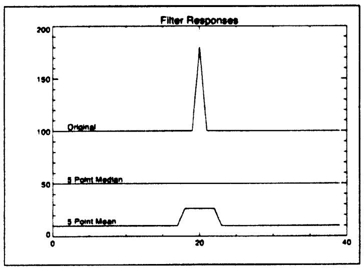


Figure 5-2 on page 160 of the *PV-WAVE User's Guide*.

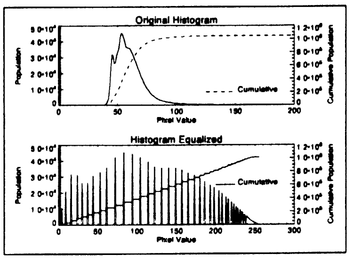


Figure 5-1 on page 157 of the *PV-WAVE User's Guide*.

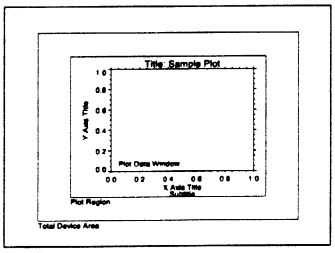


Figure 3-14 on page 86 of the *PV-WAVE User's Guide*.

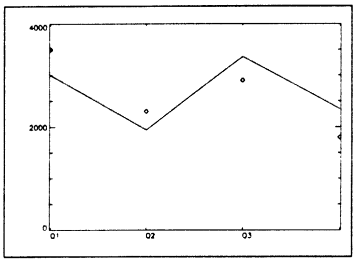
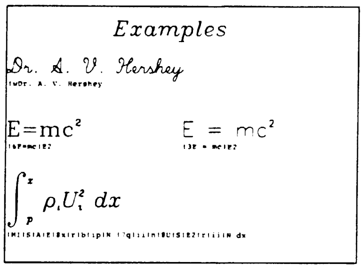


Figure 7-8 on page 253 of the *PV-WAVE User's Guide*.



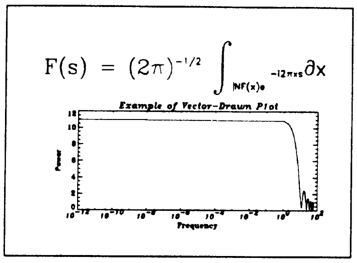
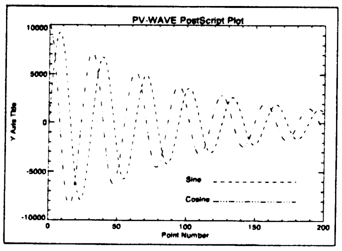
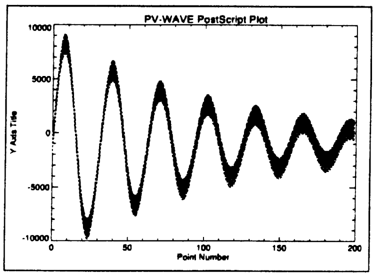


Figure 9-3 on page 300 of the *PV*-*WAVE User's Guide.*

Figure 9-4 on page 303 of the *PV*-*WAVE User's Guide.*





See page A-42 of the *PV-WAVE User's Guide.*

See page A-47 of the *PV-WAVE User's Guide.*

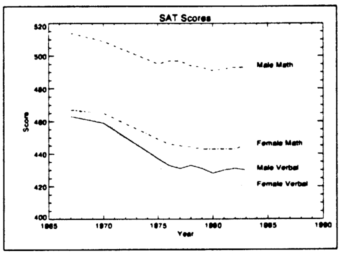


Figure 3-4 on page 67 of the *PV-WAVE User's Guide*.

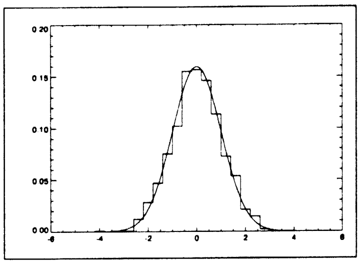


Figure 3-6 on page 71 of the *PV-WAVE User's Guide*.

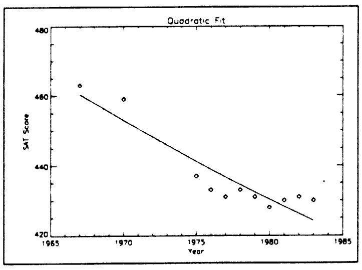


Figure 3-7 on page 73 of the *PV-WAVE User's Guide*.

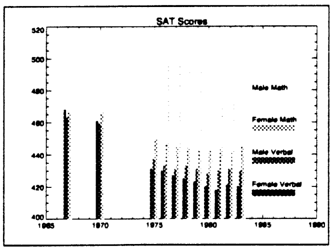


Figure 3-9 on page 77 of the *PV-WAVE User's Guide*.

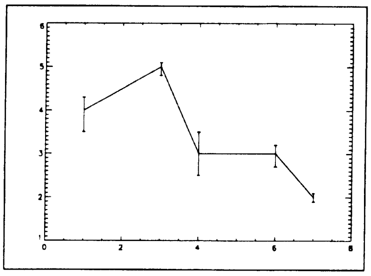


Figure 2-14 on page 308 of the *PV-WAVE Reference, Volume 1.*

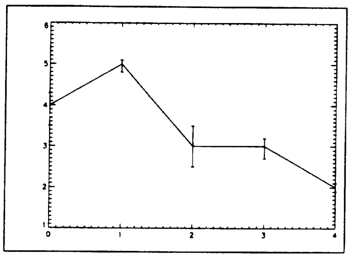


Figure 2-13 on page 307 of the *PV-WAVE Reference, Volume 1.*

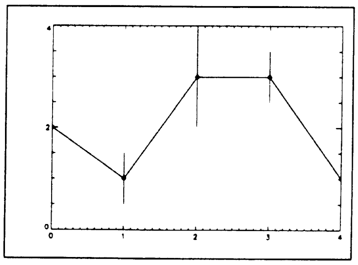


Figure 2-29 on page 506 of the *PV-WAVE Reference, Volume 1.*

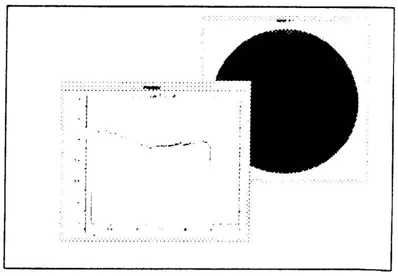


Figure 2-41 on page 586 of the *PV-WAVE Reference, Volume 1.*

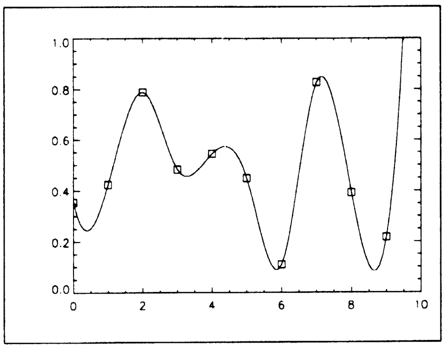


Figure 2-23 on page 490 of the *PV-WAVE Reference, Volume 1.*

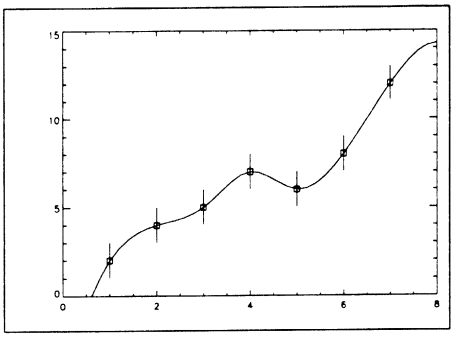


Figure 2-25 on page 493 of the *PV-WAVE Reference, Volume 1.*

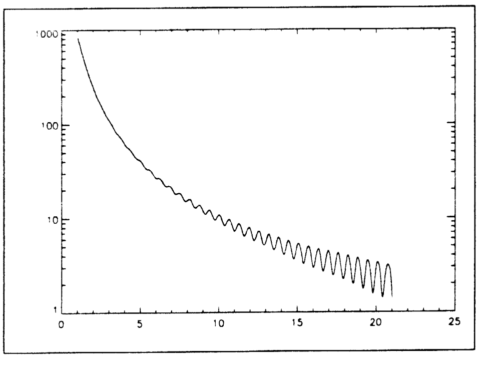


Figure 2-27 on page 501 of the *PV-WAVE User's Guide.*

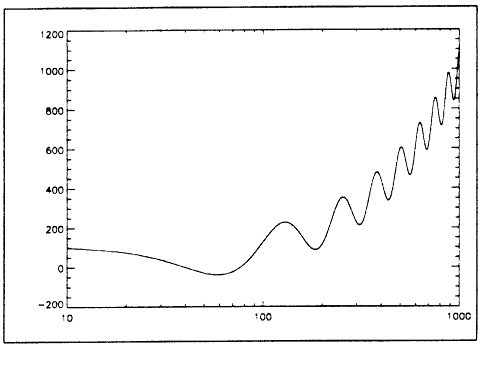


Figure 2-28 on page 502 of the *PV-WAVE User's Guide.*

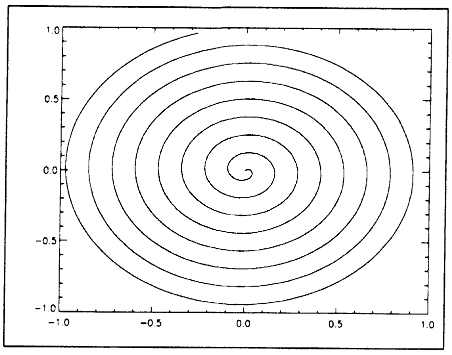


Figure 2-34 on page 513 of the *PV*-*WAVE Reference, Volume 1.*

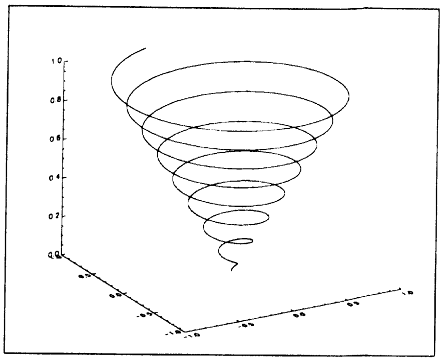


Figure 2-35 on page 514 of the *PV*-*WAVE Reference, Volume 1.*

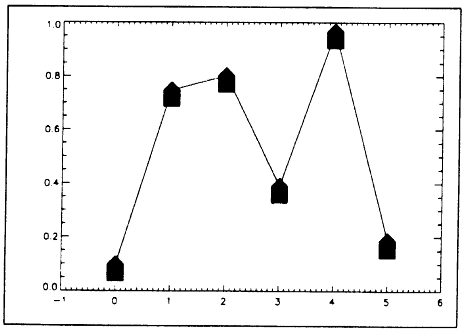


Figure 2-59 on page 237 of the *PV*-*WAVE Reference, Volume 2.*

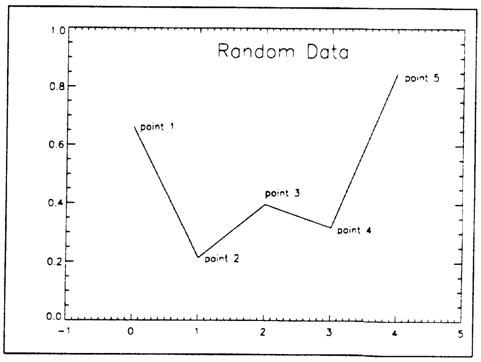


Figure 2-83 on page 492 of the *PV*-*WAVE Reference, Volume 2.*

Contour Plots

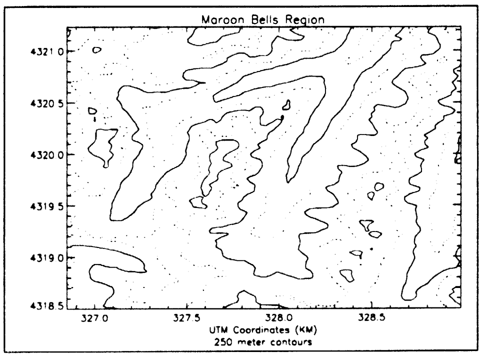


Figure 4-2 on page 100 of the *PV-WAVE User's Guide*.

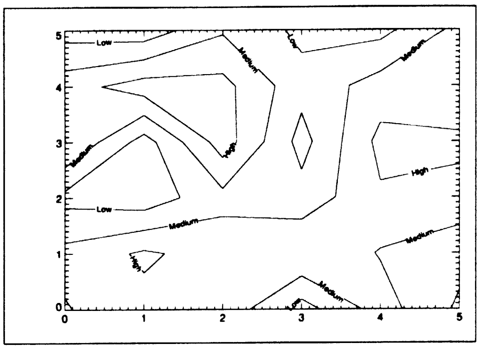


Figure 4-7 on page 108 of the *PV-WAVE User's Guide*.

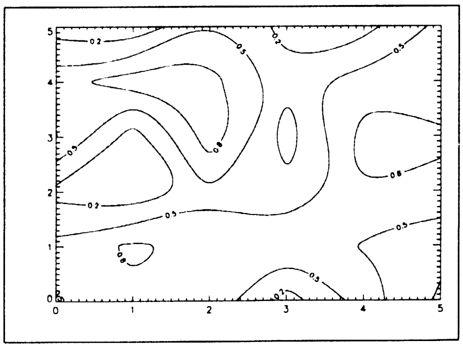


Figure 4-8 on page 109 of the *PV-WAVE User's Guide*.

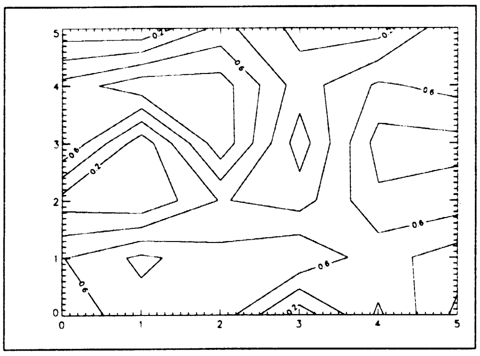


Figure 4-5 on page 106 of the *PV-WAVE User's Guide*.

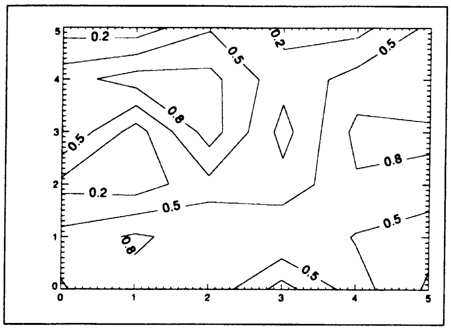


Figure 4-6 on page 107 of the *PV-WAVE User's Guide.*

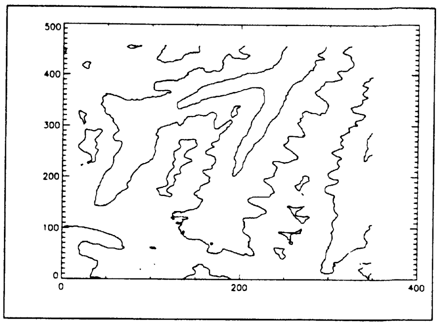


Figure 4-1 on page 97 of the *PV-WAVE User's Guide.*

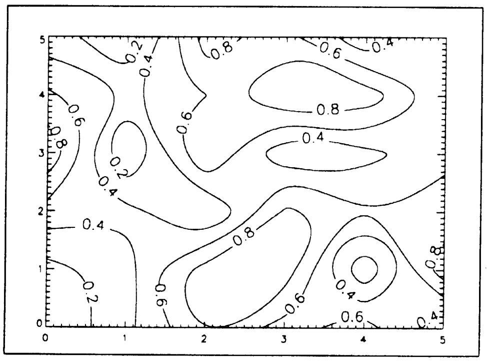


Figure 2-7 on page 122 of the *PV-WAVE Reference, Volume 1.*

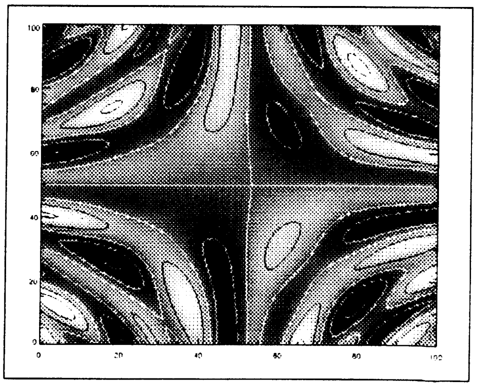


Figure 2-21 on page 404 of the *PV-WAVE Reference, Volume 1.*

3D Plots, Surface and Shaded Surface Plots

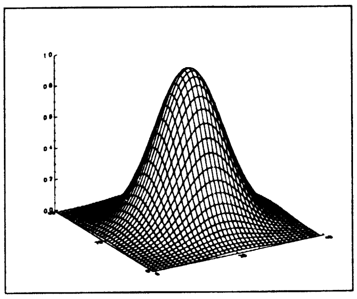


Figure 4-10 on page 113 of the *PV-WAVE User's Guide*.

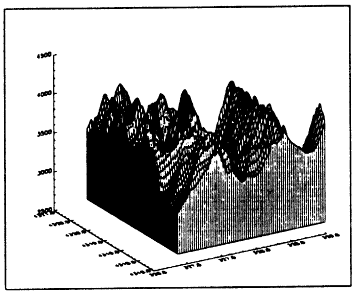


Figure 4-11 on page 114 of the *PV-WAVE User's Guide*.

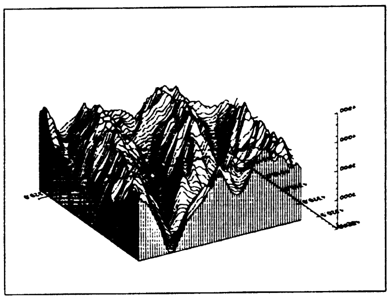


Figure 4-11 on page 114 of the *PV-WAVE User's Guide*.

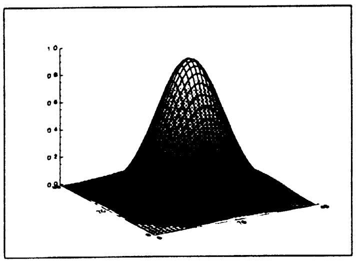


Figure 4-16 on page 133 of the *PV-WAVE User's Guide*.

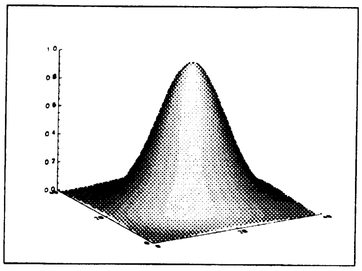


Figure 4-16 on page 133 of the *PV‑WAVE User's Guide*.

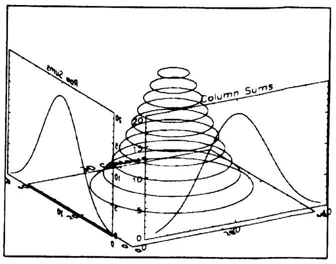


Figure 4-14 on page 127 of the *PV‑WAVE User's Guide*.

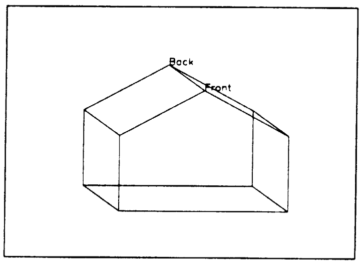


Figure 4-12 on page 124 of the *PV‑WAVE User's Guide*.

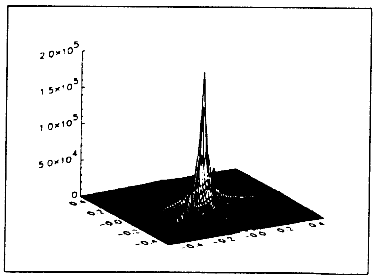


Figure 5-4 on page 166 of the *PV‑WAVE User's Guide*.

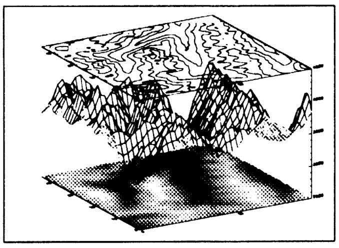


Figure 4-15 on page 129 of the *PV-WAVE User's Guide*.

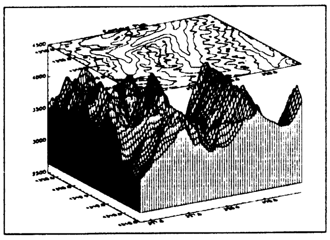


Figure 4-13 on page 126 of the *PV-WAVE User's Guide*.

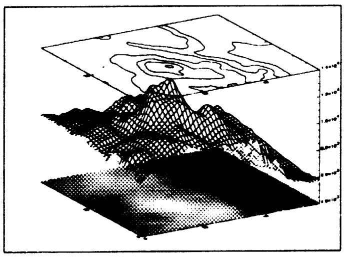


Figure 2-53 on page 106 of the *PV-WAVE Reference, Volume 2*.

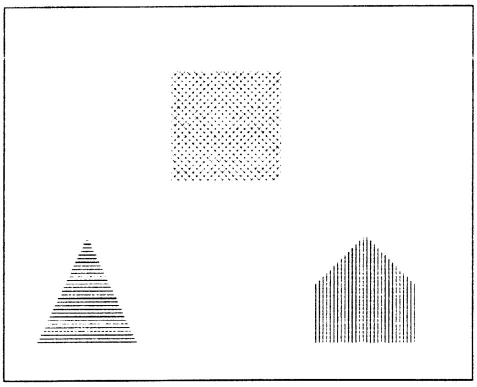


Figure 2-38 on page 545 of the *PV-WAVE Reference, Volume 1*.

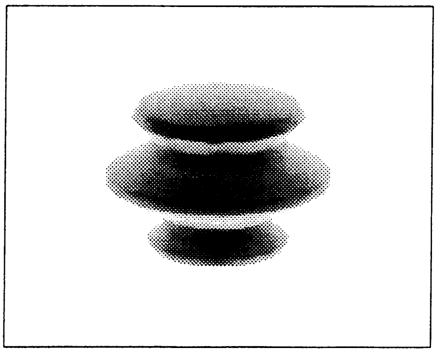


Figure 2-40 on page 567 of the *PV-WAVE Reference, Volume 1.*

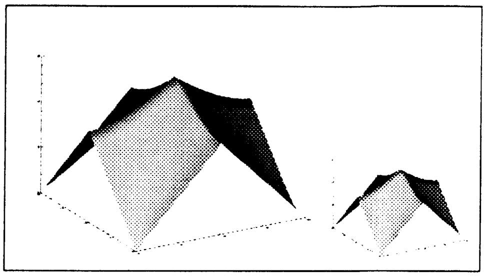


Figure 2-42 on page 23 of the *PV-WAVE Reference, Volume 2.*

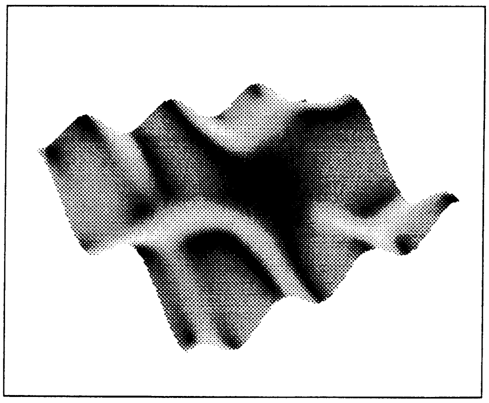


Figure 2-50 on page 95 of the *PV-WAVE Reference, Volume 2.*

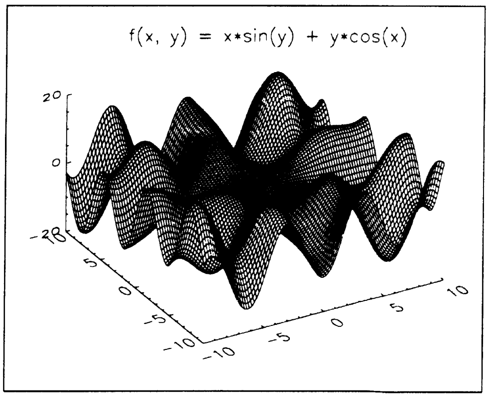


Figure 2-56 on page 171 of the *PV-WAVE Reference, Volume 2.*

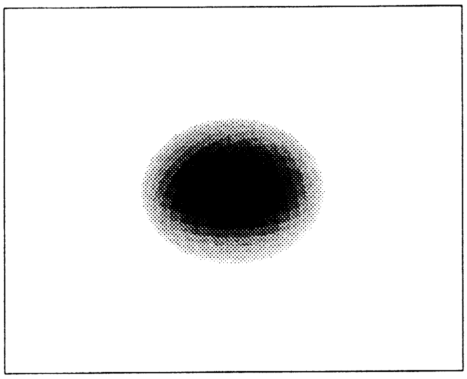


Figure 2-46 on page 72 of the *PV‑WAVE Reference, Volume 2.*

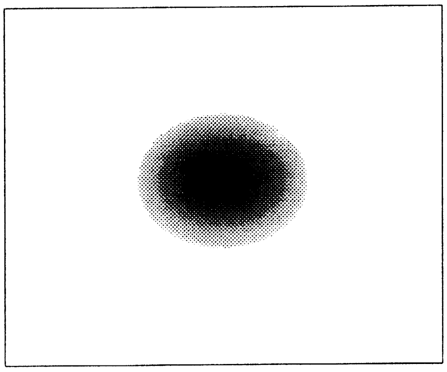


Figure 2-47 on page 73 of the *PV‑WAVE Reference, Volume 2.*

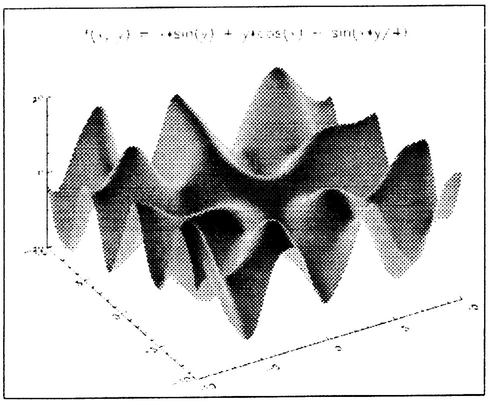


Figure 2-48 on page 87 of the *PV‑WAVE Reference, Volume 2.*

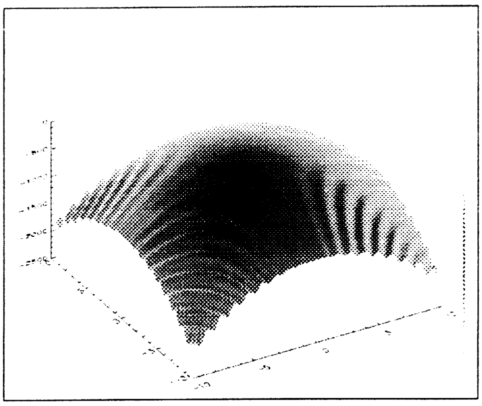


Figure 2-49 on page 90 of the *PV‑WAVE Reference, Volume 2.*

Images

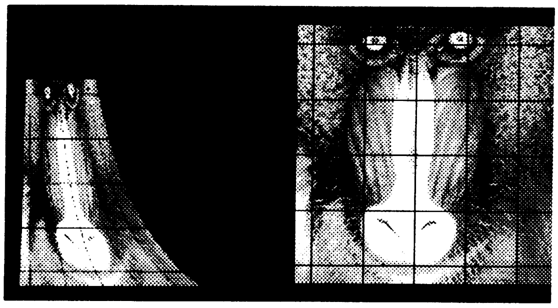
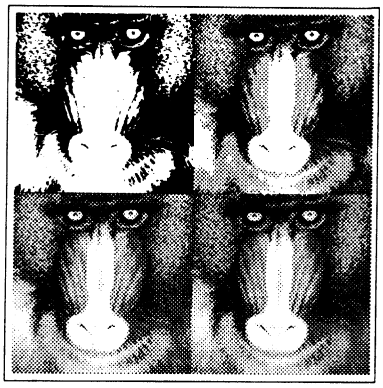


Figure 5-6 on page 172 of the *PV-WAVE User's Guide*.



See page A-43 of the *PV-WAVE User's Guide*.

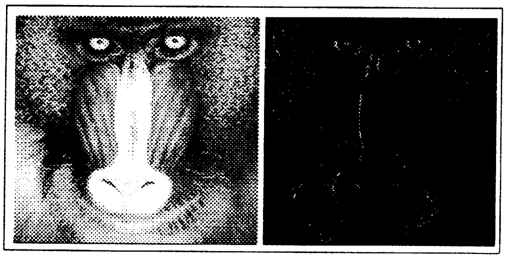


Figure 2-8 on page 128 of the *PV-WAVE Reference, Volume 1*.

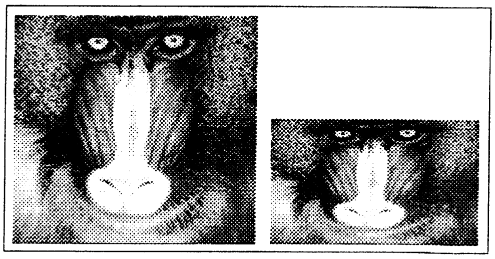


Figure 2-6 on page 117 of the *PV-WAVE Reference, Volume 1*.

Figure 2-12 on page 303 of the *PV-WAVE Reference, Volume 1.*

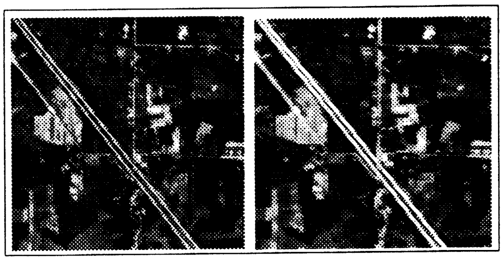


Figure 2-11 on page 258 of the *PV-WAVE Reference, Volume 1.*

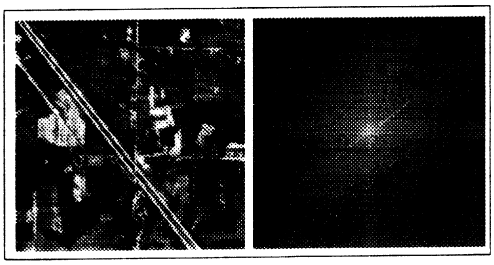


Figure 2-15 on page 330 of the *PV-WAVE Reference, Volume 1.*

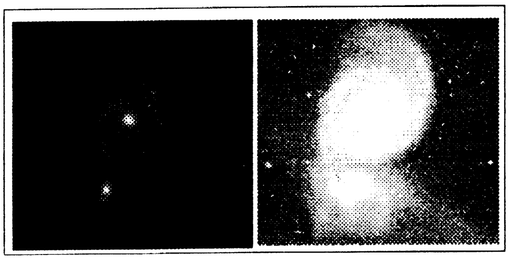


Figure 2-16 on page 385 of the *PV-WAVE Reference, Volume 1.*

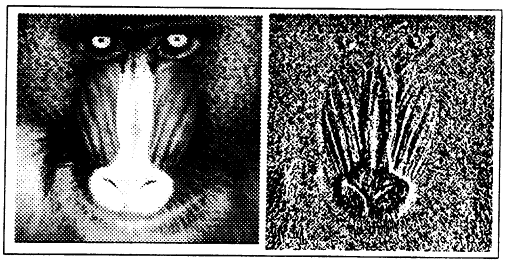


Figure 2-51 on page 102 of the *PV‑WAVE Reference, Volume 2.*

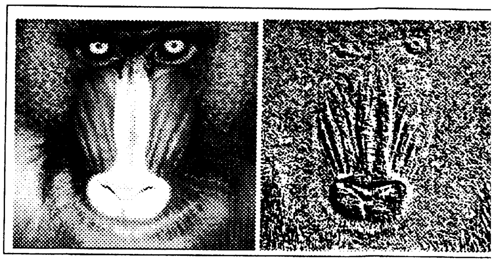


Figure 2-52 on page 103 of the *PV‑WAVE Reference, Volume 2.*

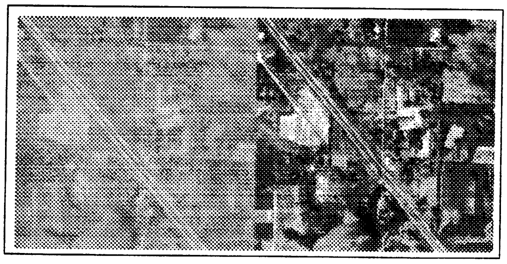


Figure 2-43 on page 47 of the *PV‑WAVE Reference, Volume 2.*

Figure 2-54 on page 121 of the *PV‑WAVE Reference, Volume 2.*

Figure 2-55 on page 124 of the *PV-WAVE Reference, Volume 2.*

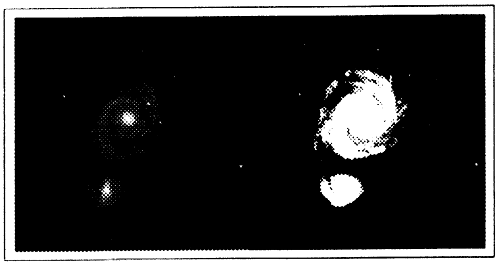


Figure 2-2 on page 76 of the *PV-WAVE Reference, Volume 1.*

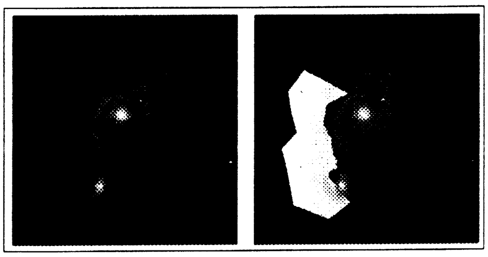


Figure 2-9 on page 234 of the *PV-WAVE Reference, Volume 1.*

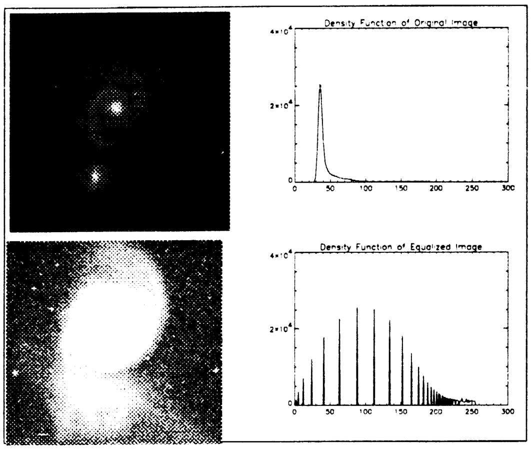


Figure 2-17 on page 392 of the *PV-WAVE Reference, Volume 1.*

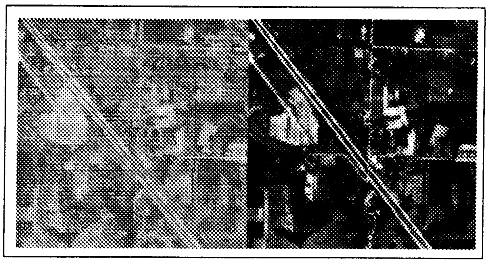


Figure 2-58 on page 229 of the *PV-WAVE Reference*, *Volume* 2.

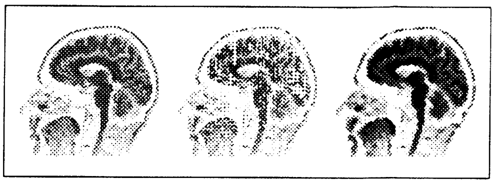


Figure 2-22 on page 456 of the *PV-WAVE Reference*, *Volume* 1.

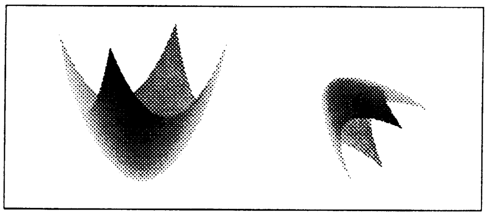


Figure 2-44 on page 50 of the *PV-WAVE Reference*, *Volume 2.*

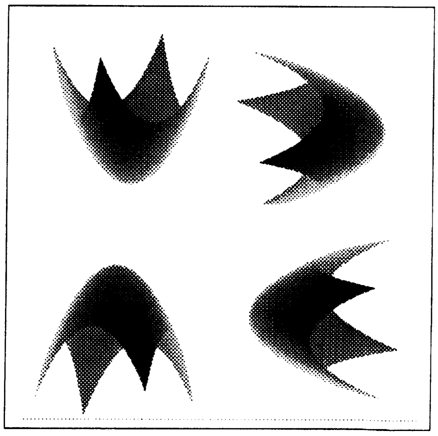


Figure 2-45 on page 53 of the *PV-WAVE Reference, Volume 2.*

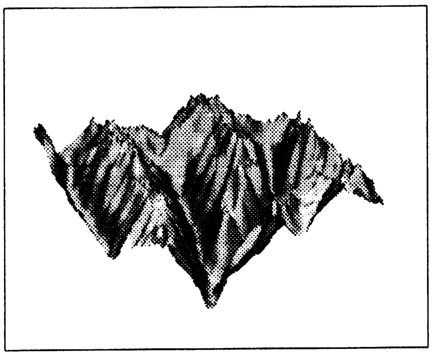


Figure 4-17 on page 134 of the *PV-WAVE User's Guide*.

Figure 4-3 on page 101 of the *PV-WAVE User's Guide*.

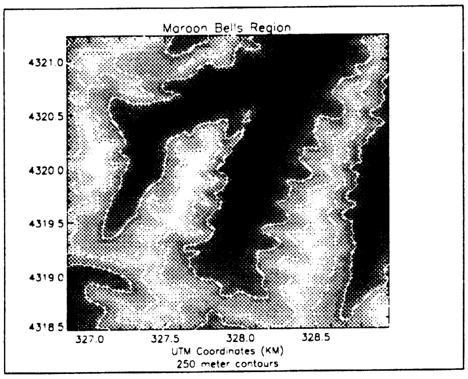


Figure 4-4 on page 103 of the *PV-WAVE User's Guide*.

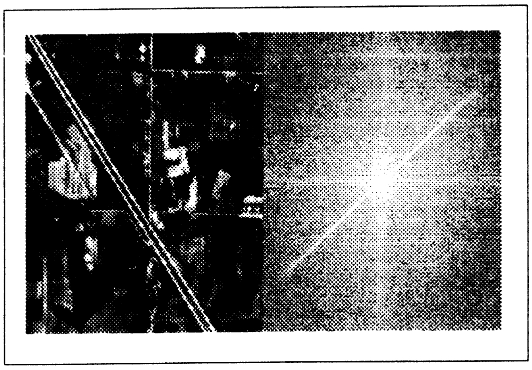


Figure 5-3 on page 166 of the *PV-WAVE User's Guide*.

Rendered Images

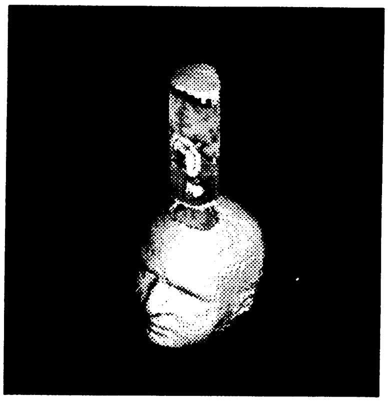


Figure 6-8 on page 216 of the *PV-WAVE User's Guide*.

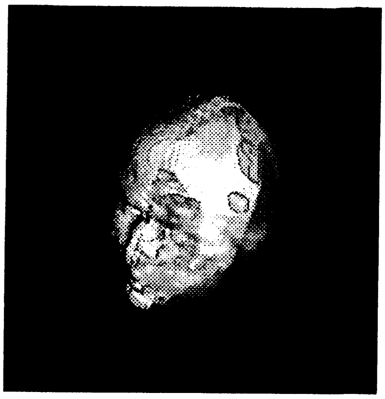


Figure 6-7 on page 214 of the *PV-WAVE User's Guide*.

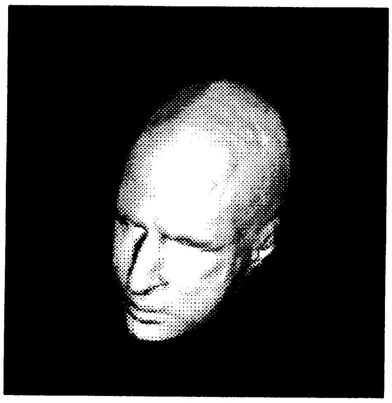


Figure 6-4 on page 205 of the *PV-WAVE User's Guide*.

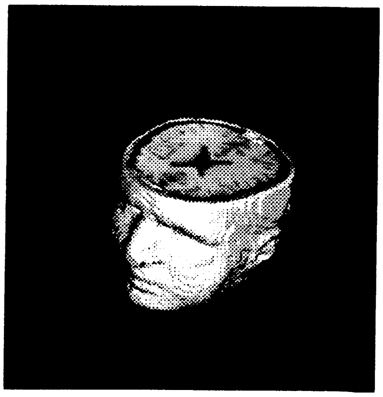


Figure 6-6 on page 212 of the *PV-WAVE User's Guide*.

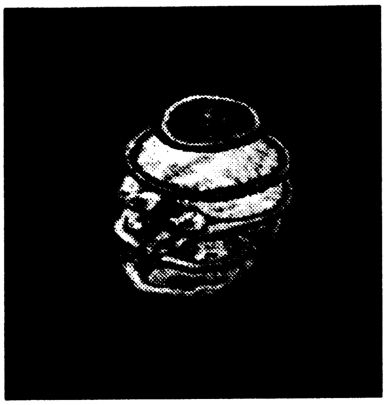


Figure 6-5 on page 210 of the *PV-WAVE User's Guide*.

Multivolume Index

This is a multivolume index that includes references to the *PV-WAVE User's Guide* (UG), *PV-WAVE Programmer's Guide* (PG), and both volumes of the *PV-WAVE Reference* (R1 and R2).