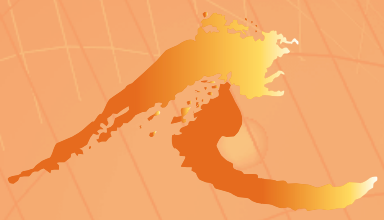
Visual Numerics



PV-WAVE® 8.0

User's Guide

Helping Customers Solve Complex Problems

PV-WAVE®

User's Guide

P/N 10017

Visual Numerics®
[www.vni.com]

Visual Numerics, Inc. – United States
Corporate Headquarters
2000 Crow Canyon Place, Suite 270
San Ramon, CA 94583

PHONE: 925.807.0138
FAX: 925.807.0145
e-mail: info@vni.com

Westminster, Colorado
10955 Westmoor Drive, Suite 400
Westminster, CO, 80021

PHONE: 303.379.3040
FAX: 303.379.2140
e-mail: info@vni.com

Houston, Texas
2500 Wilcrest, Suite 200
Houston, TX 77042

PHONE: 713.784.3131
FAX: 713.781.9260
e-mail: info@vni.com

Visual Numerics S. A. de C. V.
Florencia 57 Piso 10-01
Col. Juarez
Mexico D. F. C. P. 06600
MEXICO

PHONE: +52-55-5514-9730 or 9628
FAX: +52-55-5514-5880
e-mail:avadillo@mail.internet.com.mx

Visual Numerics International Ltd.
SoanePoint
6-8 Market Place
Reading, Berkshire RG1 2EG
UNITED KINGDOM

PHONE: +44 118.925.5910
FAX: +44 118.925.5912
e-mail: info@vniuk.co.uk

Visual Numerics, Inc.
7/F, #510, Sect. 5
Chung Hsiao E. Road
Taipei, Taiwan 110
ROC

PHONE: +88 622-727-2255
FAX: +88 622-727-6798
e-mail: info@vni.com.tw

Visual Numerics International GmbH
Zettachring 10
D-70567 Stuttgart
GERMANY

PHONE: +49-711-13287-0
FAX: +49-711-13287-99
e-mail: info@visual-numerics.de

Visual Numerics Korea, Inc.
HANSHIN BLDG. Room 801
136-1, MAPO-DONG, MAPO-GU
SEOUL, 121-050
KOREA SOUTH

PHONE: +82-2-3273-2632 or 2633
FAX: +82-2-3273--2634
e-mail: info@vni.co.kr

Visual Numerics SARL
Immeuble le Wilson 1
70, avenue du General de Gaulle
92058 Paris La Defense, Cedex
FRANCE

PHONE: +33-1-46-93-94-20
FAX: +33-1-46-93-94-39
e-mail: info@vni.paris.fr

Visual Numerics Japan, Inc.
GOBANCHO HIKARI BLDG. 4TH Floor
14 GOBAN-CHO CHIYODA-KU
TOKYO, JAPAN 102-0076

PHONE: +81-3-5211-7760
FAX: +81-3-5211-7769
e-mail: vnijapan@vnij.co.jp

The Visualizaton Toolkit

Table of Contents

Preface

Welcome to PV-WAVE! PV-WAVE is a comprehensive software environment that integrates state-of-the-art graphical and numerical analysis techniques into a system that is easy to use, easy to extend, easy to apply, and easy to learn. PV-WAVE gives you the tools you need to find solutions to, and build applications for, complex technical problems.

This manual explains how to use PV-WAVE to perform many kinds of visual data analysis (VDA) — 2D and 3D plotting, image processing, volume rendering, and mapping techniques are discussed. In addition, this manual discusses how to manage your PV-WAVE session, use color to enhance displayed data, create tables of data, and incorporate date/time data into your plots.

What's in this Manual

This manual covers the following topics:

- **Chapter 1, *Learning PV-WAVE*** — Provides an overview of the topics discussed in this manual.
- **Chapter 2, *Getting Started: UNIX and OpenVMS*** — Discusses some of PV-WAVE's basic operations under UNIX and OpenVMS, such as starting and stopping the software, using the online Help and documentation systems, journaling, and saving and restoring sessions.
- **Chapter 3, *Getting Started: Windows*** — Discusses some of PV-WAVE's basic operations under Windows.
- **Chapter 4, *Displaying 2D Data*** — Covers the basics of X versus Y plotting.

- **Chapter 5, *Displaying 3D Data*** — Describes the basics of contour and surface plotting.
- **Chapter 6, *Displaying Images*** — Describes routines used for displaying images and image processing.
- **Chapter 7, *Rendering Techniques*** — Describes the routines and techniques used to render volumes.
- **Chapter 8, *Working with Date/Time Data*** — Explains how to create plots with a Date/Time axis.
- **Chapter 9, *Creating and Querying Tables*** — Discusses how to create and subset tables using SQL-like functions.
- **Chapter 10, *Using Fonts*** — Discusses how to use and format software, or vector-drawn, fonts. This chapter also discusses the difference between software and hardware fonts and how to choose between them.
- **Chapter 11, *Using Color in Graphics Windows*** — Discusses color systems and introduces the routines that control color tables and plot colors.
- **Chapter 12, *Mapping with PV-WAVE*** — Discusses mapping procedures and optimization.
- **Chapter 13, *PV-WAVE on the World Wide Web*** — Describes features that allow you to process and present data across the Internet or your intranet.
- ***User's Guide Index*** — A subject index with hypertext links to information in this manual.

Conventions Used in this Manual

You will find the following conventions used throughout this manual:

- Code examples appear `in this typeface`. For example:

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

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

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

- 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 GUI development routines are shown in mixed case (WwMainMenu).
- A `$` at the end of a line of PV-WAVE code indicates that the current statement is continued on the following line. By convention, use of the continuation character (`$`) 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.
```

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

Technical Support

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

Office Location	Phone Number
Office Location	**Phone Number**
North American PV-WAVE Family Technical Support Westminster, Colorado	303-939-8920
Visual Numerics Corporate Headquarters San Ramon, California	925-807-0138
North American IMSL Family Technical Support Houston, Texas	713-784-3131
France	+33-1-46-93-94-20
Germany	+49-711-13287-0
Japan	+81-3-5211-7760
Korea	+82-2-3273-2633
Mexico	+52-5-514-9730
Taiwan	+886-2-727-2255
United Kingdom	+44-1-118-925-5910

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- The name and version number of the product. For example, PV-WAVE 7.0.
- The type of system on which the software is being run. For example, SPARC-station, IBM RS/6000, HP 9000 Series 700.

- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.3.
- A detailed description of the problem.

FAX and E-mail Inquiries

Contact Visual Numerics Technical Support staff by sending a FAX to:

Office Location	FAX Number
North American PV-WAVE Family Technical Support Westminster, Colorado	303-245-5301
Visual Numerics Corporate Headquarters San Ramon, California	925-807-0145
North American IMSL Family Technical Support Houston, Texas	713-781-9260
France	+33-1-46-93-94-39
Germany	+49-711-13287-99
Japan	+81-3-5211-7769
Korea	+82-2-3273-2634
Mexico	+52-5-514-4873
Taiwan	+886-2-727-6798
United Kingdom	+44-1-118-925-5912
Taiwan	+886-2-727-6798

---

or by sending E-mail to:

Office Location	E-mail Address
North American PV-WAVE Family Technical Support Westminster, Colorado	`support@boulder.vni.com`

Visual Numerics Corporate Headquarters San Ramon, California	`info@vni.com`
North American IMSL Family Technical Support Houston, Texas	`support@vni.com`
PV-WAVE Mailing List	`Majordomo@vni.com`
France	`support@vni-paris.fr`
Germany	`support@visual-numerics.de`
Japan	`vda-sprt@vnij.co.jp`
Korea	`support@vni.co.kr`
Mexico	`avadillo@mail.internet.com.mx`
Taiwan	`support@vni.com.tw`
United Kingdom	`support@vniuk.co.uk`

Electronic Services

Service	Address
General e-mail	`info@boulder.vni.com`
Support e-mail	`support@boulder.vni.com`
World Wide Web	`http://www.vni.com`
Anonymous FTP	`ftp.boulder.vni.com`
FTP Using URL	`ftp://ftp.boulder.vni.com/VNI/`
PV-WAVE Mailing List:	`Majordomo@boulder.vni.com`
To subscribe include:	`subscribe pv-wave YourEmailAddress`
To post messages	`pv-wave@boulder.vni.com`

CHAPTER 1

Learning PV-WAVE

This chapter discusses some of the PV-WAVE learning aids that are available. Use this chapter to find the best place for you to begin exploring PV-WAVE.

Using the Tutorial

The *PV-WAVE Tutorial* helps you begin using PV-WAVE Foundation and the companion technologies — PV-WAVE Visual Exploration, PV-WAVE:IMSL Mathematics, PV-WAVE:IMSL Statistics — as well as some of the specialized optional toolkits. The logical approach used in the tutorial gets you started in a focused and productive way, so that you can have immediate results.

We recommend that new users start with the *PV-WAVE Tutorial*.

Using Online Help

PV-WAVE has an easy-to-use online help facility that allows you to find and display information on many PV-WAVE features.

Using Online Help on UNIX and OpenVMS

Help from the Command Line

At the `WAVE>` prompt, enter:

```
WAVE> HELP
```

This command starts PV-WAVE's online help system with the main Help Table of Contents displayed by default.

You can also display help on a particular PV-WAVE command. For example, for help on the REBIN command, you can type:

```
WAVE> HELP, 'REBIN'
```

VDA Tools Help

Context sensitive help is provided with all VDA Tools (Visual Data Analysis). Each VDA Tool has a menu bar with a Help menu. The Help menu contains the following functions:

- **On Window** — Displays the Help viewer with the Table of Contents for information on the VDA Tool.
- **On PV-WAVE** — Brings up the Table of Contents for PV-WAVE online help. This is the full PV-WAVE reference.
- **On Help** — Displays detailed information on how to use the Help system.
- **On Version** — Displays the PV-WAVE version number and information on electronic services.

Help is also available from VDA Tool dialog boxes. Most dialog boxes have a **Help** button in the lower right-hand corner. When you click this button, the Hyperhelp viewer appears displaying information on the dialog box.

Printing from Online Help

To print information from the help system, select **File=>Print** from the online help viewer menu. This sends output directly to the default printer.

To configure the print driver (e.g., to specify a printer name, output filename, page orientation, scale, or number of copies), select **File=>Printer Setup** from the online help viewer menu.

For detailed instructions on how to use the print functions and how to set up the print driver, select **Help=>How to Use Help** from the online help viewer, and then select the **How To** help topics: **Print a Help Topic**, **Install a New Printer**, or **Configure a Printer**.

NOTE If you select the **File** option in the Printer Setup dialog box, the default location in which the help system looks for files is `$HOME`, and the default filename is `xprinter.eps`. To specify a different path and filename, enter them in the **File Name** text field.

If you select the **Printer** option in the Printer Setup dialog box, you must have

write access to the file `$HHHOME/xprinter/Xpdefaults` and you must install a printer for your site. The Hyperhelp help file describes how to install a printer.

Using Online Help on Windows

Help from the Command Line

At the `WAVE>` prompt, enter:

```
WAVE> HELP
```

This command starts PV-WAVE's online documentation system with the main Help Table of Contents displayed by default.

You can also display help on a particular PV-WAVE command. For example, for help on the REBIN command, you can type:

```
WAVE> HELP, 'REBIN'
```

Help from the Program Manager/Start Menu

You can start the main Help Table of Contents by clicking the PV-WAVE Help icon in the PV-WAVE program group, or by selecting PV-WAVE Help from the Start=>Programs=>PV-WAVE menu on Windows 95.

VDA Tools Help

Context sensitive help is provided with all VDA Tools (Visual Data Analysis). Each VDA Tool has a menu bar with a Help menu. The Help menu contains the following functions:

- **On Window** — Displays the Help viewer with the Table of Contents for information on the VDA Tool.
- **Index** — Brings up the Table of Contents for PV-WAVE online help. This is the full PV-WAVE reference.
- **On Help** — Displays detailed information on how to use the Help system.
- **On Version** — Displays the PV-WAVE version number and information on electronic services.

Help is also available from VDA Tool dialog boxes. Most dialog boxes have a **Help** button in the lower right-hand corner. When you click this button, the online help viewer appears displaying information on the dialog box.

Using Manuals Online

Using Manuals Online on UNIX and OpenVMS

Accessing Manuals Online from the PV-WAVE Command Line

If you have the online manuals installed on your system, you can start an interactive table of contents window by entering the following command at the WAVE> prompt:

```
HELP, /Doc
```

Accessing Manuals Online from the UNIX/VMS Prompts

From a C shell:

(UNIX) `source $VNI_DIR/wave/bin/wvsetup`

then type `wavedoc`

where VNI_DIR is the directory in which PV-WAVE is installed.

From a Korn or Bourne shell:

(UNIX) `. $VNI_DIR/wave/bin/wvsetup.sh`

then type `wavedoc`

where VNI_DIR is the directory in which PV-WAVE is installed.

(OpenVMS) `@VNI_DIR:[WAVE.BIN]WVSETUP.COM`

then type `wavedoc`

where VNI_DIR is the directory in which PV-WAVE is installed.

Using Manuals Online on Windows

Manuals Online from the Program Manager/Start Menu

You can start the main Manuals Online Table of Contents by selecting PV-WAVE Manuals Online from **Programs=>PV-WAVE 8.0 Product Family=>PV-WAVE 8.0** menu.

or

You can start main Manuals Online Table of Contents by entering the following command at the WAVE> prompt:

```
HELP, /Doc
```

Copying , Pasting and Printing from Manuals Online

The **PV-WAVE** Manuals Online include a main contents page from which you can display information from any PV-WAVE manual. Tables of Contents and indexes are available for all books. You can copy and paste examples directly into PV-WAVE from the manuals, or you can print all or part of any manual.

For more information on copying, pasting or printing from a PDF document see http://www.adobe.com/support/database.html.

For more detailed information about the online manuals, see the "Introduction to Manuals Online" — online.

The Printed Documentation Set

The Standard PV-WAVE Documentation Set

PV-WAVE Tutorial

An instructional series of lessons designed to get you off to a quick and successful start with PV-WAVE.

PV-WAVE User's Guide

Detailed information about how to use the numerous features of PV-WAVE.

PV-WAVE Programmer's Guide

Documents the PV-WAVE command language, which contains all the familiar features of typical 4GL languages, such as FORTRAN, Pascal, and BASIC.

PV-WAVE GUI Application Developer's Guide

Discusses how to develop applications in PV-WAVE that have a Graphical User Interface (GUI). This manual discusses inter-application communication, building VDA Tools, using WAVE Widgets (Ww) and Widget Toolbox (Wt) routines, and the Option Programming Interface (OPI).

PV-WAVE Reference (Volumes 1, 2, and 3)

A three-volume set that describes the PV-WAVE functions and procedure, keywords, system variables, fonts, executive commands and device drivers.

Documentation for Optional PV-WAVE Products

PV-WAVE IMSL Mathematics Toolkit

Descriptions of the routines that provide focussed, powerful tools for mathematical, statistical, and scientific computing. Many examples are included, so you can easily see how to apply these routines to your own work.

PV-WAVE IMSL Statistics Toolkit

Descriptions of the routines that provide focussed, powerful tools for mathematical, statistical, and scientific computing. Many examples are included, so you can easily see how to apply these routines to your own work.

PV-WAVE:GTGRID

Powerful interpolation and extrapolation techniques provided by PV-WAVE:GTGRID are used in PV-WAVE to produce technically superior gridded data sets. Even if your data set is large, sparse, faulted, noisy, or non-uniform, PV-WAVE:GTGRID provides you with the best in a wide choice of traditional and state-of-the-art algorithms for the gridding process.

PV-WAVE:Signal Processing Toolkit

Signal processing is widely used in engineering and scientific research and development for representing, transforming, and manipulating signals and the information they contain. The PV-WAVE:Signal Processing Toolkit is a collection of digital processing functions that work in conjunction with PV-WAVE Advantage. These functions are designed for easy use by the beginning signal processor, while providing the advanced signal processor with many options for solving difficult problems.

PV-WAVE:Image Processing Toolkit

Image Processing is used in a number of different fields, including biomedicine, microscopy, remote sensing, and scientific research. Common image processing operations include image visualization, transformation, filtering, and analysis. These types of functions and more are provided in the Image Processing Toolkit. The easy-to-use functions are available from the PV-WAVE command line as well as through a graphical user interface designed using the PV-WAVE Visual Data Analysis (VDA) technology.

PV-WAVE:Database Connection

PV-WAVE is the only Visual Data Analysis product to let you directly connect, query, and extract data from possibly your most valuable corporate asset — your formal SQL database. Database Connection uses standard SQL select statements to let you extract the data you need from any SQL database in your network. In combination with PV-WAVE's integrated table tools for managing and manipulating tabular data, there is no better way to extract meaning – and value – from your data.

PV-WAVE:ODBC Connection

PV-WAVE:ODBC Connection functions let you import data from ODBC compliant data sources into PV-WAVE for Windows. Once the data is imported, you can use PV-WAVE to analyze, manipulate, and visualize the data.

JWAVE

JWAVE lets you create Java client applications that communicate directly with PV-WAVE running on a remote server. On the server side, PV-WAVE code is used to analyze data and generate graphics. On the client side, a Java applet (or application) lets users interact with the PV-WAVE session and display the graphics returned from PV-WAVE.

Using the Gallery

The PV-WAVE Gallery is a suite of sample PV-WAVE applications. The entire Gallery program is written using PV-WAVE to display a wide range of application areas appropriate for Visual Data Analysis. The application code and data files are provided so you can extract parts of them and use them within your own applications.

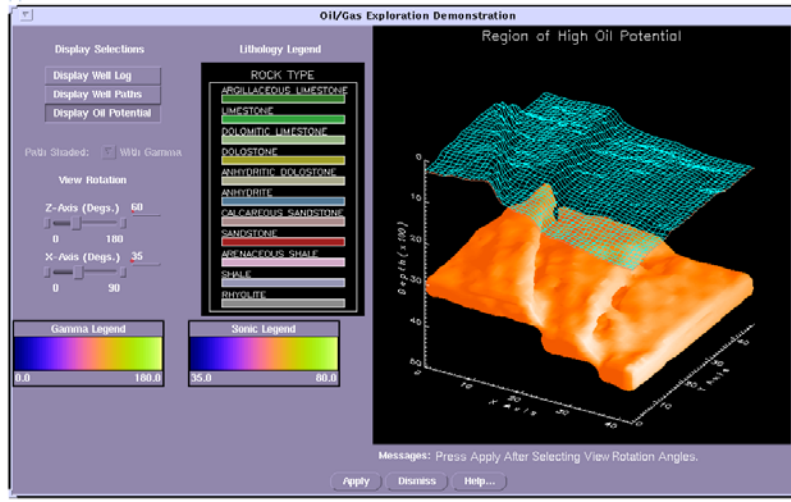


Figure 1-1 Advanced rendering techniques are used to display a region of high oil potential in this example from the PV-WAVE Gallery.

The objective of the Gallery is to highlight the performance and flexibility of PV-WAVE. While your own applications may be more or less elaborate, the Gallery helps you understand how PV-WAVE helps users discover and understand the trends, anomalies, and relationships in their data.

PV-WAVE Gallery Setup Under UNIX and OpenVMS

Starting the PV-WAVE Gallery is similar to starting up PV-WAVE. First, set up the environment variables and then start the Gallery:

From a C shell:

(UNIX)

```
source $VNI_DIR/wave/bin/wvsetup
wave_gallery
```

where `VNI_DIR` is the directory in which PV-WAVE is installed.

From a Korn or Bourne shell:

(UNIX)

```
source $VNI_DIR/wave/bin/wvsetup.sh
wave_gallery
```

where `VNI_DIR` is the directory in which PV-WAVE is installed.

(OpenVMS)

```
@VNI_DIR:[WAVE.BIN]WVSETUP.COM
```

```
@wave_gallery.com
```

where VNI_DIR is the directory in which PV-WAVE is installed.

You can run the Gallery interactively by selecting the Gallery's menu buttons.

PV-WAVE Gallery Setup Under Windows

You can start the PV-WAVE Gallery from the WAVE> prompt.

First, start up PV-WAVE.

Next, at the WAVE> prompt, press the F2 key,

— OR —

type the command for starting the Gallery.

```
WAVE> wave_gallery
```

You can run the Gallery interactively by selecting the Gallery's menu buttons.

Using the Demo Files

A great many demonstration programs are distributed with PV-WAVE. Most of these programs are located under:

(UNIX) <wavedir>/demo

(OpenVMS)<wavedir>:[DEMO]

(Windows) <wavedir>\demo

Where <wavedir> is the main PV-WAVE directory.

This demonstration area is freely available to you to explore. All of the demonstration programs are written in the PV-WAVE language. You can use any of the code from these programs in your own applications. In many of the directories, README files describe the demonstration programs in detail. Additional information can be found as comments in the program files themselves.

2

CHAPTER

Getting Started: UNIX and OpenVMS

This chapter explains how to get started using PV-WAVE if you are running under UNIX or OpenVMS.

Starting PV-WAVE

Before running PV-WAVE, the `wvsetup` command (UNIX) or `WVSETUP.COM` command (OpenVMS) must have been executed one time after installation. These commands are described in detail in your installation guide.

Starting PV-WAVE Interactively

To start PV-WAVE, at the operating system prompt type `wave` and press <Return>. The PV-WAVE prompt appears:

```
WAVE>
```

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

UNIX USERS If you use a Korn shell note the following: If you use either `set -o nounset` or `set -u` in your shell, entering the `wave` command without a parameter causes an error. These `ksh` commands tell the shell to treat unset parameters as an error when substituting. After running `wvsetup.sh`, PV-WAVE expects `$*` to contain the parameters to the `wave` call. Since `set -u` or

`set -o nounset` tell `ksh` to treat unset parameters as an error, calling PV-WAVE without parameters will cause that error.

OpenVMS USERS You may need to increase your process defaults to run PV-WAVE, especially if you use large datasets. Your OpenVMS system administrator can help you set your process defaults properly. The *Guide to OpenVMS Performance Management* provides a discussion of process limits and parameters. For example, the following process limits work well for the PV-WAVE Gallery:

```
WSdef:     5000
WSquo:    20000
Pgflquo: 100000
```

Executing a Command (Batch) File at Startup

A command file, or "batch 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 the PV-WAVE Programmer's Guide.

You can also set the environment variable (or OpenVMS logical) `WAVE_STARTUP` to execute a command file when you enter the command that starts PV-WAVE. See the PV-WAVE Programmer's Guide for more information.

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

CAUTION When you exit PV-WAVE, you are returned to the operating system prompt. Variable assignments are lost unless you saved them yourself by saving the session; however, data that is buffered for open output files is flushed to these files before exiting is complete.

Exiting on a UNIX System

If you type `EXIT` or `QUIT` at the `WAVE>` prompt, you will 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 an OpenVMS System

If you type `EXIT` or `QUIT` at the `WAVE>` prompt, you will 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 an OpenVMS System

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

Interrupting the Current PV-WAVE Command

<Control>-C is the interrupt character. Typing the interrupt character generates a *keyboard interrupt*. Under OpenVMS, <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 executive command. For more details about using executive commands such as .CON to control programs, see the PV-WAVE Reference.

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>-\.

CAUTION 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.

After aborting PV-WAVE in a UNIX environment, you may find that your terminal is set up improperly. You can restore the proper settings for your terminal by issuing the UNIX command:

```
% reset
```

or

```
% stty echo -cbreak
```

Aborting on an OpenVMS System

As with any OpenVMS program, PV-WAVE may be aborted by typing <Control>-Y.

CAUTION 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 OpenVMS to run a new program is issued prior to the `CONTINUE` command, the PV-WAVE session is totally and irreversibly lost.

Entering Commands at the Command Line

When the `WAVE>` prompt is visible, you are located at the PV-WAVE command line. The command line gives you immediate access to all the data analysis and graphics display commands and procedures that are part of PV-WAVE.

For example, the following command produces an XY plot of the integers 0 to 99:

```
WAVE> PLOT, INDGEN(100)
```

NOTE The commands PLOT and INDGEN are PV-WAVE system routines. There are many such routines, which are all documented in the *PV-WAVE Reference*.

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. Using PV-WAVE commands, you read in the data and, within seconds, you can begin manipulating it, discovering what trends and patterns it holds. Here are some additional examples of commands entered directly at the PV-WAVE command line.

```
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
        ; This statement calculates the square of the numbers 1 through 3.
```

Function and Procedure Libraries

Some functions and procedures come from an area known as the Standard Library. These are routines that have been written using the PV-WAVE language and are fully documented and supported by Visual Numerics, Inc.

You can also use functions and procedures from the Users' Library. This library contains many useful routines that have been written and submitted by PV-WAVE users; however, the routines in this area are not officially supported by Visual Numerics, Inc.

The source code for Standard Library routines is in:

(UNIX) `<wavedir>/lib/std`

(OpenVMS) `<wavedir>:[LIB.STD]`

The source code for Users' Library routines is in:

(UNIX) `<wavedir>/lib/usr`

(OpenVMS) `<wavedir>:[LIB.USR]`

Where `<wavedir>` is the main PV-WAVE directory.

For more information on the Users' Library, see the PV-WAVE Programmer's Guide.

Using Keywords to Modify Commands

Keywords are optional parameters that modify PV-WAVE commands. The *PV-WAVE Reference* lists every keyword associated with each command. For example, the PLOT command, which is used to create 2D line plots has dozens of keywords associated with it. These keywords can be used to add titles, change the color,

thickness, and style of lines, modify the way axis tick marks appear, change the data range, add symbols, and many more.

In the following example, the keyword used to modify the plot is shown in bold type:

```
PLOT, INDGEN(100), Title = 'Hello World'
```

Keywords are normally assigned either a numerical or string value. Some keywords are Boolean in nature and can either be on or off. To turn such a keyword "on", set it equal to 1 or precede the keyword by a / (backslash). Preceding a keyword by a backslash is equivalent to setting it equal to 1. To turn it "off" set it to 0. For example:

```
PLOT, INDGEN(100), INDGEN(100), /Polar
```

and

```
PLOT, INDGEN(100), INDGEN(100), Polar=1
```

are equivalent statements. They both activate the *Polar* keyword, which creates a polar plot instead of a Cartesian X-Y plot.

Most keywords have default values. The default for the *Polar* keyword is 0, or inactive. The default values for some keywords are determined by system variables.

Relationship Between Keywords and System Variables

For some keywords, the default values are derived from system variables, which are a special class of predefined variables available to all PV-WAVE applications. All system variables are denoted with an initial exclamation point (!). For example, the system variable !P.Color contains the default setting for the keyword *Color*.

When using many plotting functions and procedures, the keyword *Color* can be used to change the value of !P.Color. Here's an easy way to use a system variable to change the value of !P.Color to a bright purple color:

```
TEK_COLOR
    ; This command loads 32 predefined, unique, highly
    ; saturated colors into the bottom 32 indices of the
    ; color table.

!P.Color = 6
    ; Changes !P.Color to purple (the color identified by
    ; the color index 6). The change is temporary — it only lasts
    ; until some other color table is loaded or until you end
    ; your PV-WAVE session. For more permanent results,
    ; you can save your session, and then this setting is
```

```
; available for later use.
```

For more information about system variables, see the PV-WAVE Programmer's Guide. For more information about saving sessions, see *Saving and Restoring Sessions* on page 18 in this manual.

Using Command Recall

PV-WAVE saves the last 20 command lines you entered. 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.

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. The command `INFO, /Memory` reports the amount of dynamic memory in use and the number of times it has been allocated and de-allocated. For more information about the INFO procedure, see *Getting Session Information* in the PV-WAVE Programmer's Guide.

Saving and Restoring Sessions

The SAVE and RESTORE procedures are used together to save the state of user-generated variables, system variables, and compiled 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 *Modifying Your Environment* in the PV-WAVE Programmer's Guide for more details.
- The state of a 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 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.
- When used with the *Wavepoint* keyword, SAVE saves PV-WAVE variables so that they can be read into PV-WAVE Point & Click and PV-WAVE Personal Edition. This keyword is disabled for the Digital Alpha Digital UNIX platform.

Using the SAVE Procedure

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

Saving for Future 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. In addition, you can use keywords with SAVE. For a description of these keywords, see 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 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 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.

NOTE Visual Numerics, Inc., reserves the right to change the structure of 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.

Printing Your Work

PV-WAVE supports a number of output devices and formats, such as PostScript printers, HPGL and PCL plotters, and Computer Graphics Metafiles (CGM). These output device drivers are described in detail in the *PV-WAVE Reference*.

The five steps you take to produce graphics output are the same no matter which output device or format you use. The steps are:

- Select the graphics output device or format. (This automatically opens an output file.)
- Configure the output device to your specifications.
- Enter the PV-WAVE commands to display your graphics
- Close the output file.
- Use a UNIX or OpenVMS system command to send the output file to a printer or plotter.

For example:

```
SET_PLOT, 'ps'
    ; Select the graphics device.
DEVICE, Filename='myplot.ps', /Eps
    ; Configure the output device. This command specifies
    ; the output filename and the type of file — Encapsulated
    ; PostScript (EPS).
PLOT, INDGEN(100), Title='Hello World'
    ; Enter the graphics commands.
DEVICE, /Close
    ; Close the device.
$lpr myplot.ps
    ; Print command on a UNIX system. The dollar
    ; sign ($) is used to issue an operating system command
    ; from PV-WAVE.
$print/queue=post_q myplot.ps
    ; Print command on an OpenVMS system. The dollar
    ; sign ($) is used to issue an operating system command
    ; from PV-WAVE.
```

Each step is described in the following sections.

Selecting the Output Device with SET_PLOT

Select a graphics output device with the SET_PLOT command. The command is:

SET_PLOT, '*string*'

where *string* can be any one of the following letter codes:

Device Driver Codes

Code	Output Device
CGM	Computer Graphics Metafile format

Device Driver Codes (Continued)

Code	Output Device
HP	HPGL device
PCL	PCL device
PS	PostScript 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.

The DEVICE keywords for each output device supported by PV-WAVE are listed in the *PV-WAVE Reference*.

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 an OpenVMS 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:

(UNIX) `WAVE> $lpr peak.ps`

(OpenVMS) `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.

3

CHAPTER

Getting Started: Windows

This chapter explains how to get started using PV-WAVE if you are running under Microsoft Windows.

Starting PV-WAVE

You can start PV-WAVE in one of two "modes": Console mode or Home window mode.

In Console mode, you have access to the `WAVE>` prompt only. Home window mode provides additional features, such as menus and a tool bar, to help you manage your session.

NOTE Refer to online help for information on the features found in the Home window. Start by selecting **On Window** from the Home window Help menu.

Under Windows

When PV-WAVE was installed on your system, a Program Group was created. You can start PV-WAVE from an icon in the PV-WAVE 8.0 Product Family or by typing one of the following startup commands in MS DOS window:

- `<install_directory>\wave\bin\bin.i386nt\wave` — Starts PV-WAVE in Console mode.
- `<install_directory>\wave\bin\bin.i386nt\wavewin2` — Starts PV-WAVE in Home window mode.

After a brief pause, the PV-WAVE Console or Home window appears displaying the PV-WAVE prompt:

```
WAVE>
```

When you see this prompt, PV-WAVE is ready for you to enter commands.

Summary of PV-WAVE Startup Commands

You can start PV-WAVE in Home window mode, Console window mode, or DDE server mode. The command syntax for each mode is presented in the following table.

Command Line Syntax for Running PV-WAVE

Command	Mode
`wave`	Run PV-WAVE in a Console window. (The *options* are described in the next table.)
`wavewin2`	Run the PV-WAVE Home window.
`wavedde`	Run PV-WAVE as a DDE (Dynamic Data Exchange) server. This startup command is discussed later in this chapter.

The command line options available for the `wave` command are listed in the following table.

Command Line Options for the wave Command

Command Line Option	Meaning
filename	Execute a command file during startup.
–r *or* –rt plus *filename*	Start PV-WAVE in runtime mode. The previously compiled application stored in *filename* starts automatically.

Note: Command line options are not case-sensitive. In other words, they can be entered in either lower, mixed, or upper case.

NOTE Several of PV-WAVE's command line options can be combined on one command line.

Standard I/O and Error Redirection

Previous versions of PV-WAVE on Windows (before Version 6.0), allowed command line flags for standard I/O and error redirection. With Version 6.0, these

flags are no longer supported. Instead you can use standard I/O redirection on the command line. For example, the previous command line flags:

```
wave -i infile -o outfile -e errfile
```

can be replaced with:

```
wave < infile > outfile 2> errfile
```

in PV-WAVE 6.0.

Only the `wave` command supports I/O redirection. The `wavedde` supports redirection of standard output and standard error, but not standard input (since input is accepted only from DDE clients). The command `wavewin` does not support any I/O or error redirection.

Stopping PV-WAVE

The simplest way to stop PV-WAVE is to type `EXIT` or `QUIT` at the `WAVE>` prompt. You can also interrupt the current PV-WAVE command and then resume with the .CON command, as explained in this section.

Exiting PV-WAVE

Entering an `EXIT` or `QUIT` command at the `WAVE>` prompt causes PV-WAVE to exit unconditionally, and you are returned to the operating system prompt. The same thing happens if you enter <Control>-D or <Control>-Break; for more details, refer to *Control Characters that Interrupt or Stop PV-WAVE* on page 28.

CAUTION When you exit unconditionally, variable assignments are lost and any customizations made to PV-WAVE, such as changing the font used in the windows, are lost unless you have explicitly saved them yourself by saving the session. However, data that is buffered for open output files is flushed to these files before exiting is complete.

Interrupting the Current PV-WAVE Command

If you are running PV-WAVE in Console mode, <Control>-C is the *interrupt character*. Typing the interrupt character generates a PV-WAVE *keyboard interrupt*. When you enter the interrupt character at the `WAVE>` prompt, the following message is displayed:

```
% Interrupt encountered.
```

When the interpreter regains control (there may be a noticeable delay), you are returned to the WAVE> prompt. After interrupting PV-WAVE, you can continue with the .CON command. For more details about using executive commands such as .CON to control programs, see the PV-WAVE Reference.

Control Characters that Interrupt or Stop PV-WAVE

This section describes individual characters that can be entered in conjunction with the <Control> key to interrupt or stop PV-WAVE. These characters are summarized in the following table.

Character	Action
<Control>-C	Keyboard interrupt; enter .CON to continue.
<Control>-D	Signifies EOF; causes PV-WAVE to exit.
<Control>-Break	Abort.

NOTE The control characters listed in the previous table are only recognized in the Console window. Control characters are not recognized in any auxiliary windows, such as graphics windows. (Exception: <Control>-C is a standard Windows accelerator, so it is recognized in any window where a Copy to Clipboard operation is meaningful.)

Executing a Command (Batch) File at Startup

A command file, or "batch file," is a file that contains PV-WAVE commands. When a command file is executed, each command in it is executed. When the end of the command file is reached, control reverts to interactive mode. In other words, the Console window and the WAVE> prompt are displayed, and you can make menu selections and type commands from the keyboard. Alternatively, 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 command at the prompt in an MS-DOS window:

`wave` *filename*

NOTE The filename must be a correctly constructed command file. It cannot be a PV-WAVE procedure file or a file created with the SAVE procedure. Command files are explained in more detail in the PV-WAVE Programmer's Guide.

You can also set the environment variable `WAVE_STARTUP` to execute a command file when you enter the command that starts PV-WAVE. For more details, see the PV-WAVE Programmer's Guide.

DDE Runtime Mode — wavedde

The DDE runtime mode initializes PV-WAVE as a DDE (Dynamic Data Exchange) server. The PV-WAVE DDE server runtime mode is a non-interactive version of PV-WAVE that serves DDE requests from client applications that are able to access PV-WAVE functionality. In other words, when PV-WAVE has been initialized as a DDE server, another application can enter the commands to control this version of PV-WAVE.

Textual output and messages are displayed in the shell window from which the PV-WAVE DDE Server was launched. If the server was launched from an icon (or **Start** button), then a separate console window is created on the desktop to display the output.

For more information about how to start PV-WAVE as a DDE (dynamic data exchange) server, see the *PV-WAVE GUI Development Guide*.

For more information on starting PV-WAVE in runtime mode, see the PV-WAVE Programmer's Guide.

Windows Used by PV-WAVE

Under Microsoft Windows, PV-WAVE uses different types of windows for different tasks. The different classes of windows PV-WAVE uses are listed in the following table:

Windows PV-WAVE Uses

Window	Function
Home	Displays the `WAVE>` prompt for entering PV-WAVE commands. In addition, provides menus and a tool bar to help you manage your session. Refer to online help for information about the Home window.
Console	Displays the `WAVE>` prompt for entering PV-WAVE commands.
Graphics	Displays PV-WAVE graphics.

Windows PV-WAVE Uses

Window	Function
Help	Displays PV-WAVE online help.

Home Window

For detailed information on the Home window, refer to online help—select **Help=>On Window**.

Console Window

The Console window is where commands are entered and where PV-WAVE displays its messages and textual output.

TIP To use cut and paste in the Console window under Windows 95, you need to disable the **Fast Paste** function. To do this, click on the **Properties** icon in the MS-DOS window where you are running PV-WAVE. **Fast Paste** is listed under the **Misc** tab.

NOTE The Console window differs slightly between Windows NT and Windows 95. The Windows 95 version contains a row of icons used for editing text. For information on these functions, refer to Windows online help. The following figures show both the Windows NT and Windows 95 versions of the Console window for PV-WAVE.

Graphics Windows

PV-WAVE graphics windows are used to display data in a variety of ways. The type of graphics window you choose to use depends on the dimensions of the variables you have to display and the type of analysis you wish to perform. For example, if you have imported 8-bit image data, you would probably use the TV or TVSCL commands to view your data as an image. The image is then displayed in a graphics window.

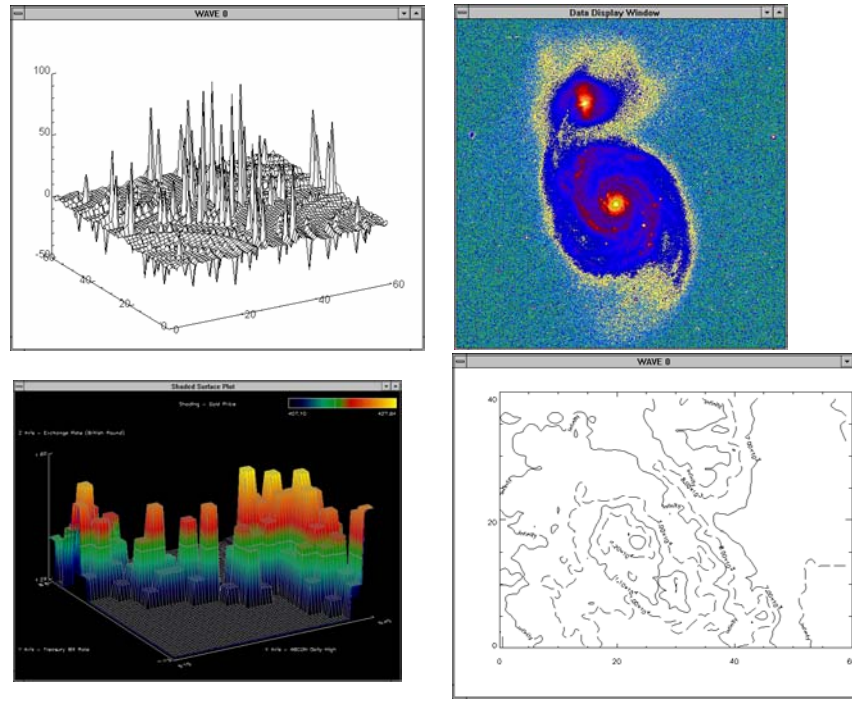


Figure 3-1 PV-WAVE graphics windows

Help Window

The PV-WAVE Help window displays information from PV-WAVE's online help system. The Help window includes controls that you can use to access the information in a variety of ways. Refer to the online help topic, **How to Use Help**, for more detailed information about the Help window, and *Using Online Help on Windows* on page 3 of this manual.

You can access this help topic by selecting **Help=>How to Use Help** from virtually any Windows application, including PV-WAVE. An example of a PV-WAVE Help window is shown in *Figure 3-2*.

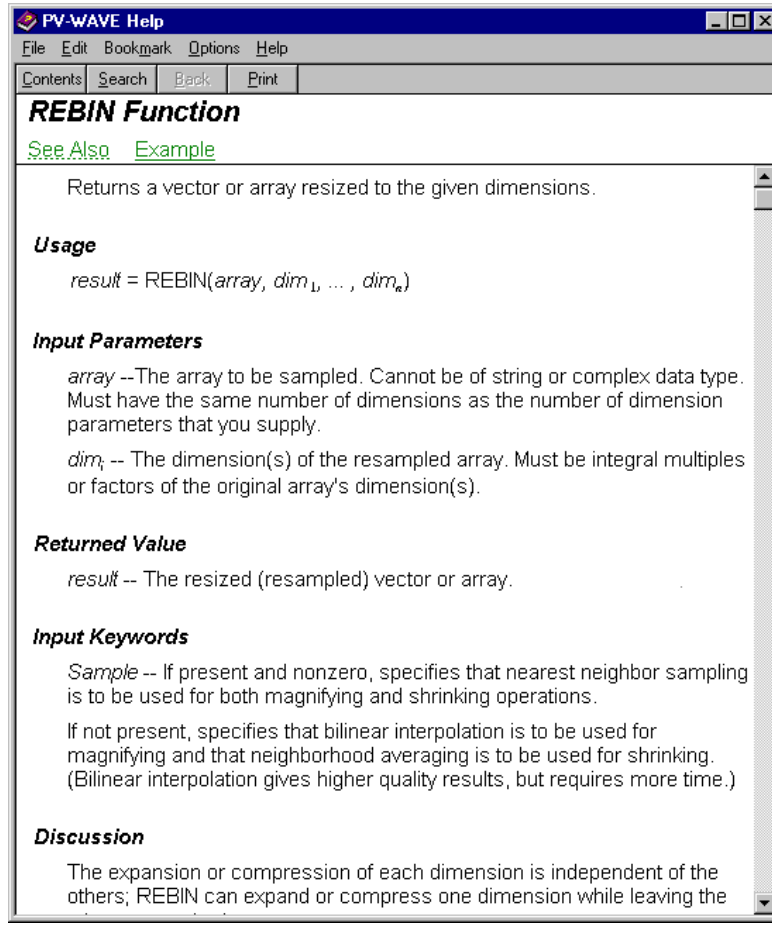


Figure 3-2 PV-WAVE Help window (Windows version).

Entering Commands at the Command Line

This section discusses ways to communicate with PV-WAVE.

When the `WAVE>` prompt is visible, you are located at the PV-WAVE command line. The command line gives you immediate access to all the data analysis and graphics display commands and procedures that are part of PV-WAVE.

NOTE The *PV-WAVE Reference* describes all of the PV-WAVE commands (functions and procedures).

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. Using PV-WAVE commands, you read in the data and, within seconds, you can begin manipulating it, discovering what trends and patterns it holds.

The following statements can be entered directly at the `WAVE>` prompt. They create and initialize the variables `VERBM`, `VERBF`, `MATHM`, and `MATHF`, which contain the verbal and math SAT scores for males and females:

```
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. The following figure is produced using the statement:

```
PLOT, YEAR, VERBM
```

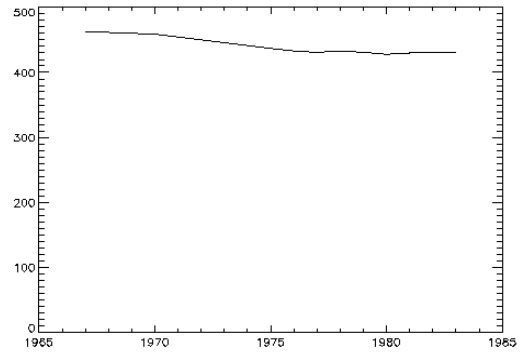


Figure 3-3 Initial 2D plot.

Function and Procedure Libraries

Some functions and procedures come from an area known as the Standard Library. These are routines that have been written using the PV-WAVE language and are fully supported by Visual Numerics, Inc. The source code for Standard Library routines can be found in:

(Windows) `<wavedir>\lib\std`

You can also use functions and procedures from the User Contributed Library; this area is in

(Windows) `<wavedir>\lib\user`

Where `<wavedir>` is the main PV-WAVE directory.

Remember, however, that the routines in this area are not officially supported by Visual Numerics, Inc.

For more information about how the Users' Library is maintained, see the PV-WAVE Programmer's Guide.

Using Keywords to Modify Commands

Keywords are optional parameters that modify PV-WAVE commands. The *PV-WAVE Reference* lists every keyword associated with each command. For example, the PLOT command, which is used to create 2D line plots, has dozens of keywords associated with it. These keywords can be used to add titles, change the color, thickness, and style of lines, modify the way axis tick marks appear, change the data range, add symbols, and many more.

The following example plots the values in the variable *y*. The keywords used to modify the plot are shown in bold type:

```
PLOT, y, XRange = [200, 600], $
     YRange = [-40, 40], Color = 36, $
     Background = 110, XTitle = 'Index', $
     Title = 'PV-WAVE 2D Plot', /Normal
```

Notice that keywords are normally assigned either a numerical or string value. Some keywords are Boolean in nature and can either be on or off. To turn such a keyword "on", set it equal to 1 or precede the keyword by a / (slash). Preceding a keyword by a slash is equivalent to setting it equal to 1. To turn it "off" set it to 0. For example:

```
PLOT, y, /Polar
```

and

```
PLOT, y, Polar = 1
```

are equivalent statements. They both activate the *Polar* keyword, which creates a polar plot instead of a Cartesian X-Y plot.

Most keywords have default values. The default for the *Polar* keyword is 0, or inactive. The default values for some keywords are determined by system variables.

Relationship Between Keywords and System Variables

For some keywords, the default values are derived from *system variables*. System variables are a special class of predefined variables available to all PV-WAVE applications. All system variables are denoted with an initial exclamation point (!). For example, the system variable !P.Color contains the default setting for the keyword *Color*.

When using many plotting functions and procedures, the keyword *Color* can be used to change the value of !P.Color. Here's an easy way to use a system variable to change the value of !P.Color to a bright purple color:

```
TEK_COLOR
    ; This command loads 32 predefined, unique,
    ; highly-saturated colors into the bottom 32 indices
    ; of the color table.

!P.Color = 6
    ; Changes !P.Color to purple (the color identified by the
    ; color index 6). The change is temporary — it only lasts until
    ; some other color table is loaded or until you end your
    ; PV-WAVE session. For more permanent results, you can
    ; save your session, and then this setting is available for later use.
```

For more information about system variables, see the PV-WAVE Programmer's Guide. For more information about saving sessions, see *Saving and Restoring PV-WAVE Sessions* on page 37 of this manual.

Using Command Recall

To recall previously entered commands, use the arrow keys as shown in the following table.

To access a previous command	Press these keys
Move “up”	Up arrow (↑)
Move “down”	Down arrow (↓)

The command recall buffer “remembers” the last 20 commands that you have entered.

Function Keys

By default, the following keys are assigned to actions:

Keyboard Accelerator Function Keys

Function Key	Action
F1	Begins the PV-WAVE Gallery, an automated demonstration.
F2	Invokes PV-WAVE’s online help system.
F3	Invokes PV-WAVE’s INFO command and prints the current session status to the screen.

TIP These function keys can be easily redefined, either by you or by someone else at your site. This topic is discussed further in the next section.

Assigning Commands to Function Keys

Function keys may be equated to a character string using the DEFINE_KEY procedure. This allows frequently used strings and commands to be entered with a single keystroke. For example, the <F10> key on your keyboard can be equated to the string PLOT, as shown in the example below.

```
SETUP_KEYS
    ; Load predefined function key definitions.
DEFINE_KEY, 'F10', 'PLOT'
    ; Enter the text “PLOT” at the WAVE> prompt when
    ; the F10 function key is pressed.
```

For detailed information on how to customize the behavior of your function keys using the DEFINE_KEY procedure, see its description in the *PV-WAVE Reference*.

To see how your function keys are presently defined, enter this command:

```
INFO, /Keys
```

TIP A natural place to put your key definitions is in your startup file so that the function keys are defined when PV-WAVE is initialized. The defaults for the key definitions are established by the SETUP_KEYS procedure that gets called from the `wavestartup` file. For more information about startup files, see the PV-WAVE Programmer's Guide.

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,

```
WAVE> INFO, /Device
```

provides information about the current graphics device being used by PV-WAVE, and

```
WAVE> INFO, /Memory
```

reports the amount of dynamic memory in use and the number of times it has been allocated and de-allocated. For more information about the INFO procedure, see *Getting Session Information* in the PV-WAVE Programmer's Guide.

Saving and Restoring PV-WAVE Sessions

You can enter the SAVE and RESTORE commands at the `WAVE>` prompt. These functions are used to save and later restore the state of user-generated variables, system variables, and compiled procedures and functions.

CAUTION If you run PV-WAVE in a Console window, you will not be prompted to save your session when you close the Console window or when you exit Windows.

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 *Modifying Your Environment* in the PV-WAVE Programmer’s Guide.
- 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 color table 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.

NOTE For more information about the keywords that you can use when you enter the command this way, refer to the description of SAVE in Chapter 2, *Function and Procedure Reference* in the *PV-WAVE Reference*.

CAUTION Creating a new save file causes any existing file with the same name to be lost. Use the *Filename* keyword with the SAVE procedure to avoid destroying files that you want to keep.

Using the RESTORE Procedure

The RESTORE command restores the objects previously saved in a save file when you used the SAVE procedure at the `WAVE>` prompt.

If a filename is not supplied in the call to RESTORE, the filename `wavesave.dat` is used. In addition, you can use keywords with RESTORE.

NOTE For a description of these keywords, see the description of RESTORE in Chapter 2, *Function and Procedure Reference* in the *PV-WAVE Reference*.

Things to Remember when Restoring Files

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.

NOTE 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.

Printing Your Work

PV-WAVE provides several methods of printing graphics. The easiest method is to print directly from the window in which the graphics are displayed, but there are other ways to print, too.

PV-WAVE supports hardcopy output to various plotters and printers, including all the hardcopy devices supported by Windows. PV-WAVE also includes some of its own hardcopy drivers; these drivers provide you with options not available when you print via Windows.

Printing the Contents of a Graphics Window

The **Print** function on the graphics window Control menu lets you print the contents of the graphics window.

The location of the graphics window Control menu button for a Windows window is shown in the following figure. On Windows, the Control menu is in the same location.

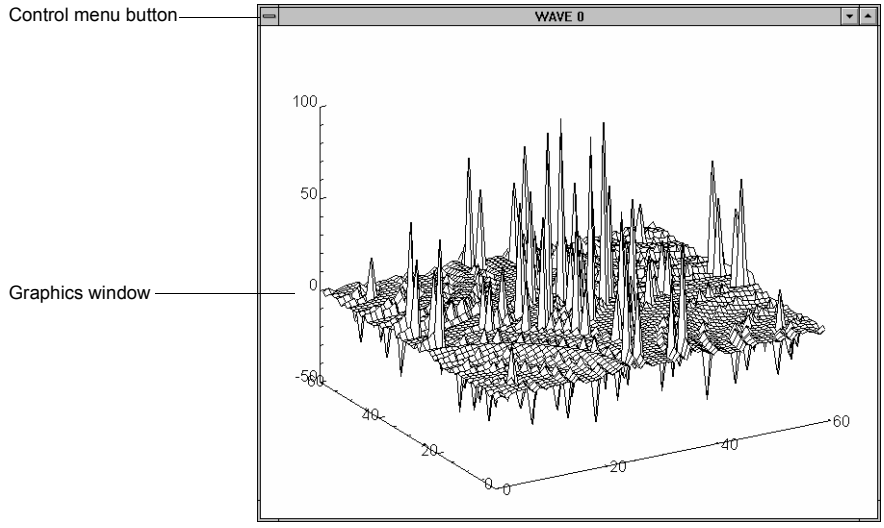


Figure 3-4 PV-WAVE graphics window (Windows version). Click the Control menu button to display this window's Control menu.

When you select **Print**, the Print dialog box appears. Use this dialog box to specify printing options and to print your graphics. Refer to online help for detailed information on using the Print dialog box.

In addition, you can print the contents of a graphics window using the WPRINT command. For detailed information on WPRINT, see its description in the *PV-WAVE Reference*.

Printing PV-WAVE Help Topics

Any help topic can be easily printed by displaying it in the Help window and then selecting **File=>Print Topic**. The help topic printout will be sent to your computer's default printer.

To change your default printer, use the Print Manager window provided by Windows.

Using the PV-WAVE Output Drivers

Output drivers allow you to output graphics in formats that can be exchanged with other applications or sent to an output device. The following table lists the output drivers supported by PV-WAVE.

Supported Output Devices and Window Systems

Device Name	Description
NULL	No graphic output
CGM	Computer Graphics Metafile generator
HP	Hewlett-Packard Graphics Language (HPGL) plotters
PCL	Hewlett-Packard Printer Control Language (PCL)
PM	Pixel map
PS	PostScript devices
REGIS	Regis graphics output devices
TEK	Tektronix or compatible terminals
WIN32	Microsoft Windows WIN32 driver
WMF	Windows metafile
X	X Window System
Z	Z-buffer device

The following steps apply no matter which output driver you select:

- Select the output device, such as PS or CGM, using the SET_PLOT command.
- Configure the output device to your specifications with keywords to the DEVICE procedure.
- Issue the commands that will display your graphic output, such as PLOT or SHADE_SURF.
- Close the graphics device using the *Close* keyword to the DEVICE procedure. For non-windowing devices, this will close the file containing your graphics output.

See the *PV-WAVE Reference* for information on the DEVICE and SET_PLOT commands and examples of their use.

Exporting Graphics to a File

Non-windowing graphics devices (PS, PM, WMF, CGM, etc.) automatically place the output of the PV-WAVE graphics command into a file. You may specify the name of this file by using Filename keyword to the DEVICE command.

Use the window Controls' menu **Export Graphics** function to save the contents of a graphics window in a file. When you choose this option, you see a dialog box that lets you select a filename and directory.

WWRITE_META and WWRITE_DIB are command line functions that also save the contents of a graphics window in a file. For information on these functions, see their descriptions in the *PV-WAVE Reference*.

Using the Clipboard

You can use the clipboard to copy graphics between PV-WAVE graphics windows and between PV-WAVE and other graphics applications.

You can use the Clipboard to exchange graphics between PV-WAVE and other applications if the other application supports the file formats:

- Device Independent Bitmap (DIB) or
- Enhanced-format metafile (EMF).

NOTE Many 16-bit Windows applications do not support enhanced metafiles.

Copying Graphics to the Clipboard

Select **Copy to Clipboard** to copy the graphics in the graphics window to the Clipboard. Graphics on the Clipboard can be pasted into another PV-WAVE graphics window, or into any graphics application that allows interaction with the Clipboard. For example, you can paste PV-WAVE graphics from the Clipboard into a Microsoft Paintbrush window.

Pasting Graphics from the Clipboard

Select **Paste from Clipboard** to paste the graphics on the Clipboard into a PV-WAVE graphics window. For example, you can copy graphics from a Microsoft Paintbrush window to the Clipboard, and then paste the graphics into PV-WAVE.

4

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, perspective plotting, and data clipping 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 5, *Displaying 3D Data*. Procedures used to display and process images are discussed in Chapter 6, *Displaying Images*.

Summary of 2D Plotting and General Graphics Routines

A list of the 2D plotting procedures described in this chapter is found in . In addition, a summary list of graphics procedures often used with the plotting procedures is listed.

Customizing Plots with Keyword Parameters

The 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 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 , and in .

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 in data, device, or normal coordinate systems. These systems are explained in the following sections.

Almost all the 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 system established by the most recent PLOT, CONTOUR, or SURFACE call. 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.

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 5, *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

Data to Normal conversion

$$N_x = \begin{cases} X_0 + X_1 D_x & \text{linear} \\ X_0 + X_1 \log D_x & \text{logarithmic} \end{cases}$$

Data to Device conversion

$$R_x = \begin{cases} V_x(X_0 + X_1 D_x) & \text{linear} \\ V_x(X_0 + X_1 \log D_x) & \text{logarithmic} \end{cases}$$

Normal to Device conversion $R_x = N_x V_x$

Normal to Data conversion

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

Device to Data conversion

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

Device to Normal conversion $N_x = R_x / V_x$

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 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 4-1* is produced using the statement:

```
PLOT, YEAR, VERBM
```

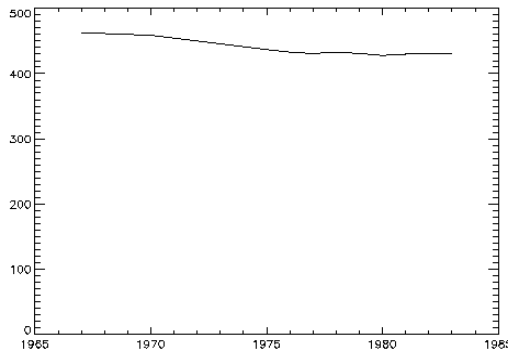


Figure 4-1 –Initial 2D plot.

TIP 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 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 4-2* 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 . 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 4-2*. This figure was produced with the statement:

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

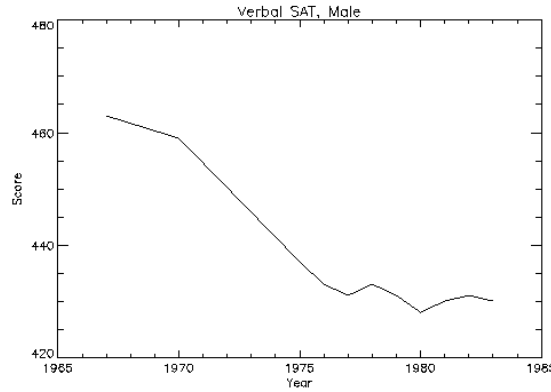


Figure 4-2 Properly scaled plot with added title annotation

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]
```

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 XStyle = 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 4-3 illustrates the result.

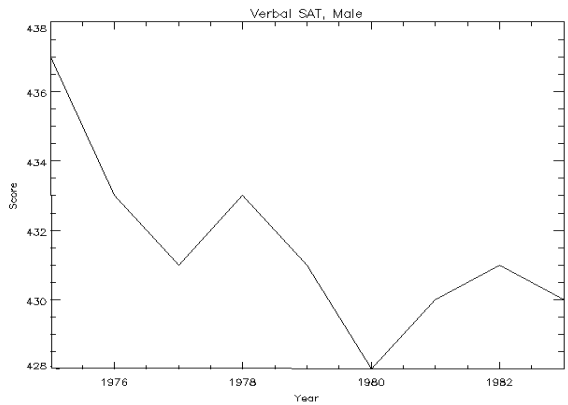


Figure 4-3 Plot with *x*–axis range of 1975 – 1983.

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 .

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

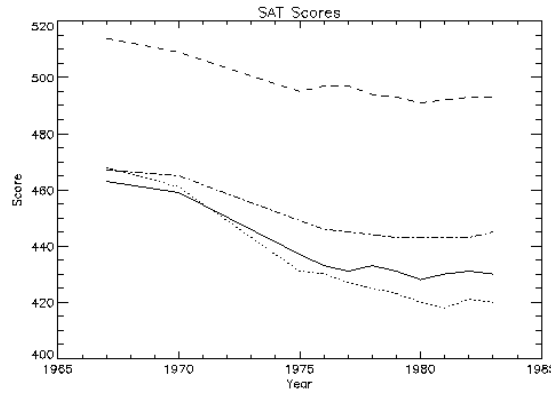


Figure 4-4 Overplotting using different line styles.

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 array operators and functions can be applied to the entire data set, rather than explicitly referencing the particular score.

Figure 4-4 is 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 first 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 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 8, *Working with Date/Time Data*.

Annotating Plots

An obvious problem with *Figure 4-4* 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 10, *Using Fonts* explains the difference between these types of fonts and the advantages and disadvantages of each.

The annotation in *Figure 4-5* uses the PostScript Times-Roman font. This is 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 the *PV-WAVE Reference*.

Using XYOUTS to Annotate Plots

You can add labels and other annotation to your plots with the XYOUTS procedure. The XYOUTS procedure is used to write graphic text at a given location (X, Y):

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 *Clipping PV-WAVE Graphics* on page 69.

Figure 4-5 illustrates one method of annotating each graph with its name. The plot is produced in the same manner as was *Figure 4-4*, with the exception that the *x*-axis range is extended to the year 1990 to allow room for the titles. To accomplish this, the keyword parameter `XRange = [1967, 1990]` is 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 4-5* is produced using the statements:

```
names = ['Female Verbal', 'Male Verbal', $
   'Female Math', 'Male Math']
       ; Vector containing the name of each score.
n1 = N_ELEMENTS(year) - 1
    ; Index of last point.
FOR i=0,3 do XYOUTS, 1984, allpts(n1,i), names(i)
       ; Append the title of each graph on the right.
```

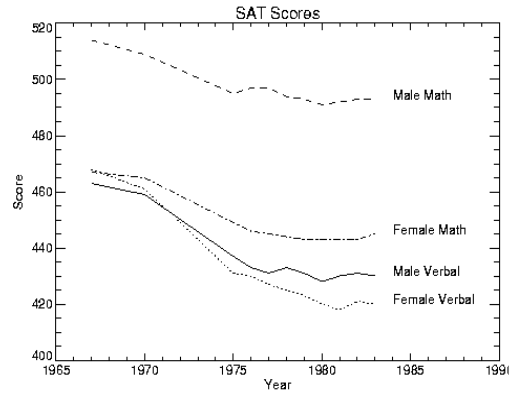


Figure 4-5 Example of annotating each line. The font used is the hardware-generated PostScript Times-Roman font.

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 4-6 illustrates this by comparing the distribution of the normally distributed random number function (RANDOMN), to the theoretical normal distribution:

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

This figure is produced using the following statements:

```
X = FINDGEN(200) / 20. - 5.
    ; Generate 200 values ranging from –5 to 5.
Y = 1 / SQRT(2. * !PI) * EXP(-X^2 / 2) *(10. / 200)
        ; Theoretical normal distribution, integral scaled to one.
H = HISTOGRAM(RANDOMN(Seed, 2000), $
      BINSIZE = 0.4, min = -5., max = 5.)/2000.
        ; Approximate a 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 (see Figure 4-6).
```

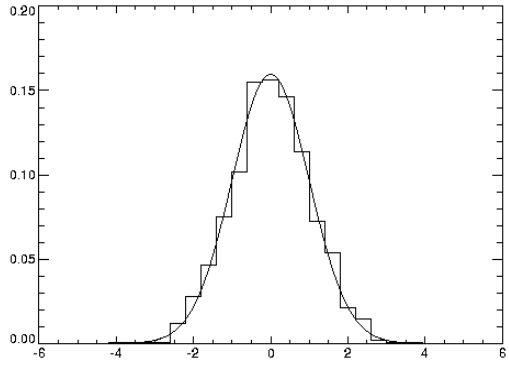


Figure 4-6 Plotting in histogram mode.

Using Different Marker Symbols

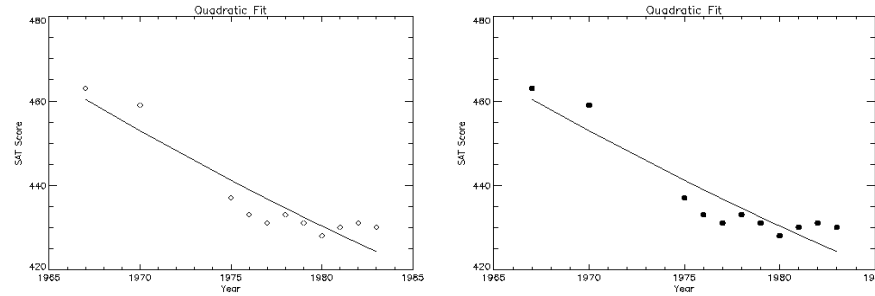
Each data point may be marked with a symbol and/or connected with lines. The value of the keyword parameter *Psym* selects the marker symbol. *Psym* is described in detail in , .

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 4-7* 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 (Figure 4-7 (a)).
OPLOT, YEAR, YFIT
    ; Overplot the smooth curve using a plain line (Figure 4-7 (b)).
```



(a) (b)

Figure 4-7 (a) Plotting with predefined marker symbols, and (b) with user-defined symbols.

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 one-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.

Figure 4-7 (b) illustrates the use of USERSYM to define a new symbol, a filled circle. It is 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 = 2πi / 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 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.

Windows USERS The *Pattern* keyword is not available for the POLYFILL procedure.

Figure 4-8 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.

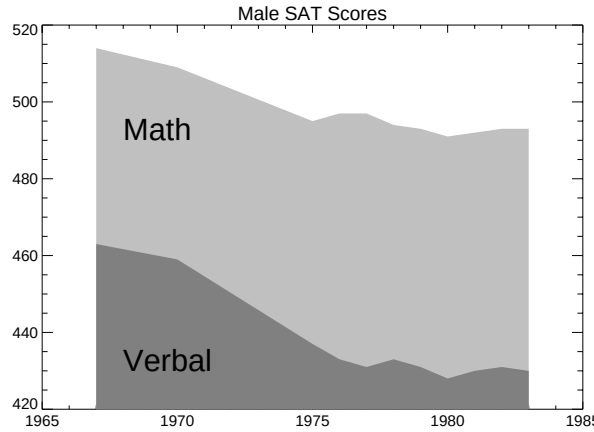


Figure 4-8 Filling regions using POLYFILL.

The following discussion describes the program that produced *Figure 4-8*. First, a plot axis is drawn 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 48), 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 containing 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. PV-WAVE can produce many types of 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 4-9* is shown below and annotated. A procedure called BOX is defined which draws a box given the coordinates of two diagonal corners.

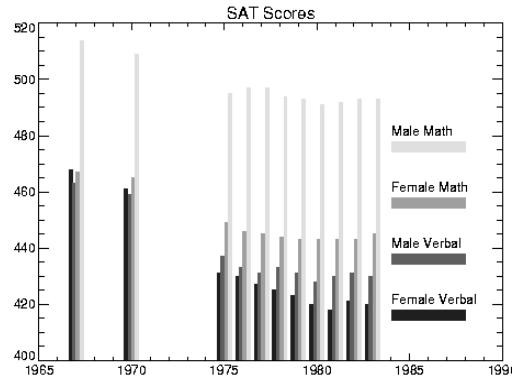


Figure 4-9 Bar chart drawn with POLYFILL.

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
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)
    ; Annotation of y–value. 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
    ; Vertical bar x–offset 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 a 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 ,.

Example 1: Specifying Tick Labels and Values

Figure 4-10 is a bar chart illustrating the direct specification of the *x*–axis tick values, number of ticks, and tick names. Building upon the BOX program described in the previous section, this program shows each of the four scores for the year 1967, the first year in the data. The BOX procedure is used to draw a rectangle for each score. Using the same data and variables from that example, the program for specifying tick labels and values is as follows.

```
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 the first year,
       ; corresponding to the names vector from the previous example.

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.
```

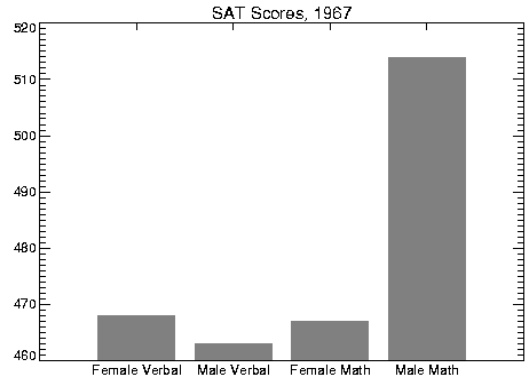


Figure 4-10 Controlling *x*–axis tick marks and their annotation.

Example 2: Specifying Tick Lengths

Figure 4-11 illustrates the 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 the 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, and setting the
       ; number, position and labels of the 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 is to create a dotted or dashed grid on the plot region. To do this, 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 , Volume 3.

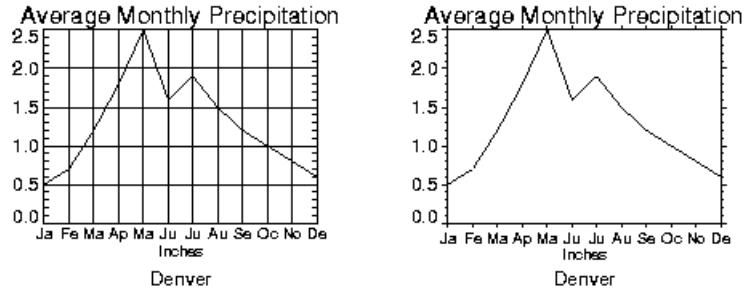


Figure 4-11 Full grid produced with tick marks (right) and outward-extending tick marks (left).

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 the *PV-WAVE Reference*.

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:

40.00 400.00 4000.00 40000.00

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-08 1.00e-06 1.00e-04 1.00e-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 stack two plots vertically on each page specify the following value for !P.Multi.

```
!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 4-12 illustrates the two rows and two columns format. Use the following command to reset the display to the default setting of one plot per page.

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

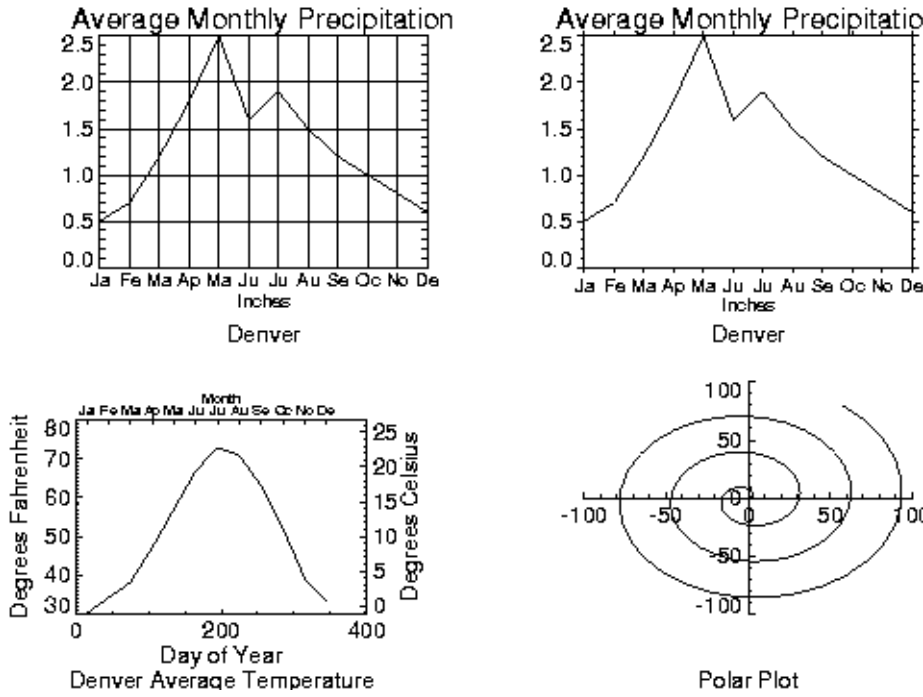


Figure 4-12 Multiple plots per page.

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 (l*i*near *x*–axis, l*o*g *y*–axis), PLOT_OI (l*o*g *x*, l*i*near *y*), or PLOT_OO (l*o*g *x*, l*o*g *y*). The OPLOT procedure uses the same scaling and transformation as did the most recent plot.

Figure 4-13 illustrates the use of PLOT_IO to make a linear-log plot. It is produced using the following statements:

```
X = FLTARR(256)
    ; Create data array.
X(80:120) = 1
    ; Make a step function.
FREQ = FINDGEN(256)
FREQ = FREQ < (256-FREQ)
```

```
    ; Make a filter symmetrical about x = 64.
FIL = 1. / (1+(FREQ / 20) ^2)
    ; A 2nd order Butterworth, with acutoff frequency = 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
```

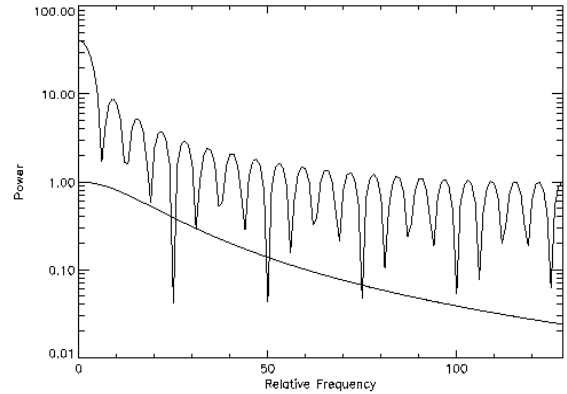


Figure 4-13 Logarithmic scaling of a second order Butterworth filter.

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 4-14* illustrates the relationship of the plot data window, plot region, and the entire device area (or window if using a windowing device).

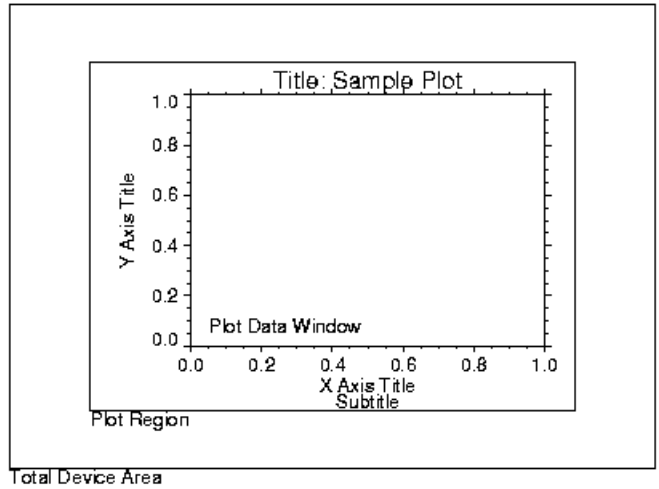


Figure 4-14 Relationship of the plot data window, plot region, and the device area.

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 and .

- 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 4-15 illustrates using AXIS to draw axes with a different scale, opposite the main *x*– and *y*–axis.

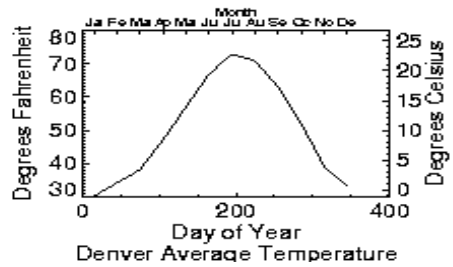


Figure 4-15 Plot containing axes with different scales, created with the AXIS procedure.

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, omitting the right and top axes.

AXIS, XAxis = 1, XTicks = 11, XTickv = day, $
      XTickname = 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 the right y–axis. Scale the 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 4-16 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', $
      XStyle = 4, YStyle = 4, /Polar
        ; Plot the data, suppressing the axes by setting their styles to 4.
```

```
AXIS, XAxis = 0, 0, 0
AXIS, YAxis = 0, 0, 0
    ; Draw the x– and y–axes through (0,0).
```

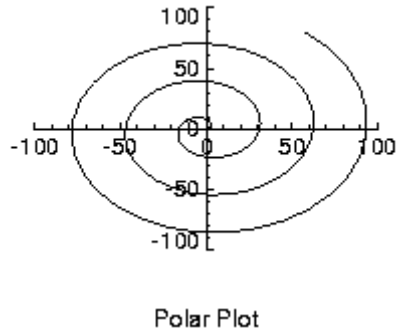


Figure 4-16 A polar plot.

Clipping PV-WAVE Graphics

Clipping removes data from a specified region of the display device. Keywords provided with the PV-WAVE graphics commands let you specify how clipping is done.

The clipping concept can be described in terms of a "clipping rectangle." Graphics that fall inside the clipping rectangle are displayed; graphics that fall outside the rectangle are not — they are clipped.

For example, the following commands produce the graphics shown in *Figure 4-17*:

```
PLOT, HANNING(100,100)
    ; Produces the graphic on the left (a). Data is not clipped.
PLOT, HANNING(100,100), Clip=[0,0, 5000,1]
    ; Produces the graphic on the right (b). Data is clipped outside the
    ; clipping rectangle, which is defined with the Clip keyword.
```

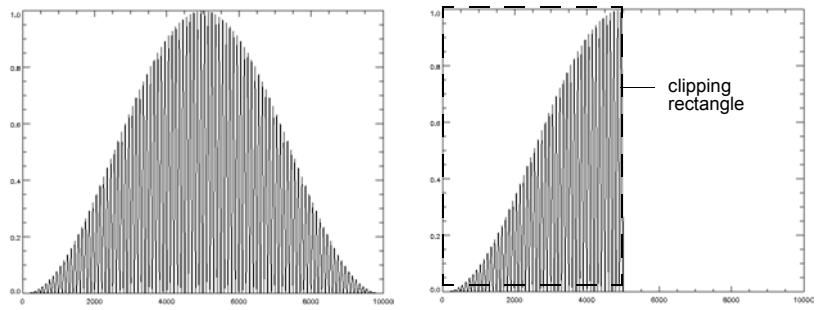


Figure 4-17 The graphic in (a) is displayed without clipping. The same data is plotted in (b), but clipping is used so that only half of the data is shown. (The dashed line shows the clipping rectangle.)

Defining a Clipping Rectangle

The following illustration shows a plot that is clipped. The dashed line shows the boundary of the clipping rectangle. (This dashed line does not appear on the actual PV-WAVE plot.) Note that all of the data that falls outside this rectangle are clipped. The data that fall inside the rectangle are displayed normally.

The *Clip* keyword is used to define the clipping rectangle as follows:

```
PLOT, INDGEN(100), Clip=[25,25,75,75]
```

The *Clip* keyword specifies the lower-left and upper-right corners of a rectangle:

$$Clip = [X_0, Y_0, X_1, Y_1]$$

By default, the *Clip* keyword accepts data coordinates.

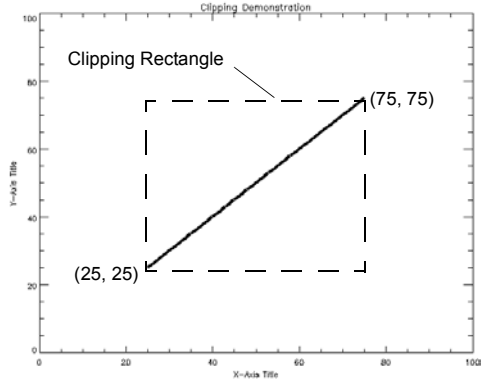


Figure 4-18 Graphics outside the boundary of the clipping rectangle are not displayed — they are clipped.

The following illustration shows the same plot with a different clipping region defined:

```
PLOT, INDGEN(100), Clip=[50,50,100,100]
```

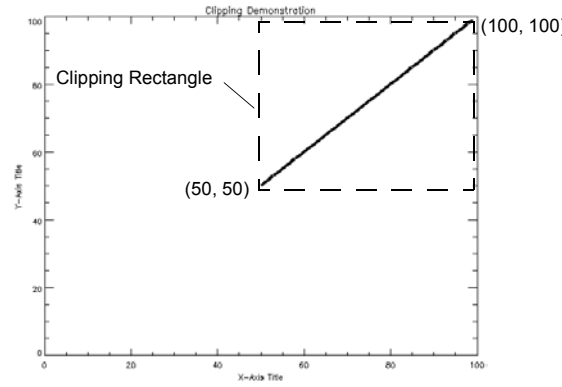


Figure 4-19 Another clipping rectangle defined by the *Clip* keyword.

The next illustration shows the same plot once again, but this time the clipping rectangle is defined in such a way that none of the data are displayed.

```
PLOT, INDGEN(100), Clip=[20,60, 40,90]
```

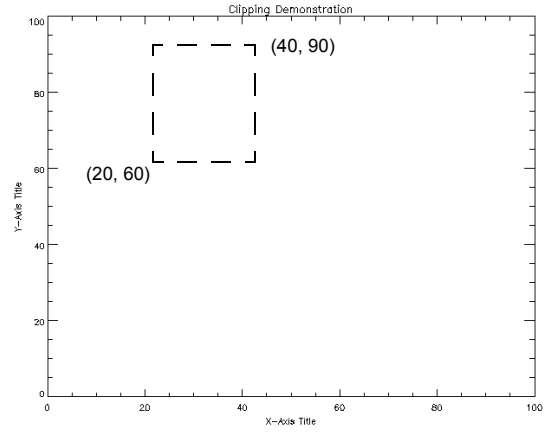


Figure 4-20 The clipping rectangle is defined so that none of the data are displayed. All of the data fall outside the clipping rectangle.

How is Clipping Controlled in PV-WAVE?

The following graphics keywords and system variables control clipping. They are listed here in their order of precedence. The first keyword in the list, *NoClip*, takes precedence over all the other keywords and system variables below it, and so on.

1. *NoClip* (graphics keyword)
2. *Clip* (graphics keyword)
3. *PClip* (graphics keyword)
4. !P.NoClip *(*system variable)
5. !P.Clip *(*system variable)

The following sections explain how these keywords and system variables are used to control clipping of PV-WAVE graphics.

For more information on these keywords and system variables, see and .

Which PV-WAVE Commands Use Clipping

The graphics procedures that use clipping are:

- CONTOUR (see Chapter 5, *Displaying 3D Data*)
- SURFACE (see Chapter 5, *Displaying 3D Data*)
- PLOT
- OPLOT
- POLYFILL
- PLOTS
- XYOUTS

The way clipping works depends on the graphics command you are using. The table in the next section breaks these commands into three groupings. Each grouping handles clipping in the same way. That is, each group has the same clipping defaults, accepts the same clipping keywords, and reads the same clipping system variables.

Notes on the Keywords and System Variables

The *Clip* keyword takes data coordinates by default. To clip in normal or device coordinates, add `/Device` or `/Normal` to the graphics command.

When you call PLOT, the value of !P.Clip is *set* to a default value. This value depends on the current device. Setting the *Clip* keyword has no effect on !P.Clip.

The default clipping rectangle for OPLOT is defined by the value of !P.Clip. !P.Clip is set by the call to PLOT, CONTOUR, or SURFACE.

Since clipping is disabled for PLOTS, POLYFILL, and XYOUTS by default, the *NoClip* keyword has little importance for these commands.

Changing the value of the !P.NoClip system variable has no effect on PLOTS, POLYFILL, and XYOUTS.

Clipping controls are summarized in the following table:

Clipping Controls in PV-WAVE

Command	Default Clipping	Clipping Options
CONTOUR (*), PLOT, SURFACE (*)	!P.Clip is set by these commands. It defines the default clipping rectangle, which is usually bounded by the coordinate axes.	Use the *Clip* keyword to specify a clipping rectangle within default clipping region. Set the *NoClip* keyword to override *Clip* and explicitly enforce the default condition. *PClip* is not a valid keyword for these commands. !P.Clip, and !P.NoClip are not recognized.
OPLOT	The clipping rectangle is defined by the coordinate axes (the plot data region).	Use the *Clip* keyword to specify a clipping rectangle. Disable clipping altogether by setting `/NoClip` or `!P.NoClip=1`. If you disable clipping, then data that falls outside the region bounded by the coordinate axes is not clipped. *PClip* is not a valid keyword for OPLOT.

Clipping Controls in PV-WAVE (Continued)

Command	Default Clipping	Clipping Options
PLOTS, XYOUTS, POLYFILL	Clipping is disabled.	Use the *Clip* keyword to specify a clipping rectangle. Set the *PClip* keyword to override the default condition. *PClip* causes the value of !P.Clip to be used to define the clipping rectangle (usually the area bounded by the coordinate axes).

* Denotes a 3D Routine, see Chapter 5, *Displaying 3D Data*.

Examples

The following examples illustrate how clipping works for specific combinations of two-dimensional plotting routines, and the clipping keywords and system variables.

OPLOT Default Clipping

In this example, PLOT plots a solid line, then OPLOT plots a dotted line. The dotted line is clipped at the boundary of the axes (the default clipping rectangle defined in the !P.Clip system variable).

```
PLOT, INDGEN(100)

OPLOT, INDGEN(100)+10, Linestyle=2
```

Figure 4-21 shows the default clipping of the OPLOT line at the at the boundaries of the axes set up for the PLOT line.

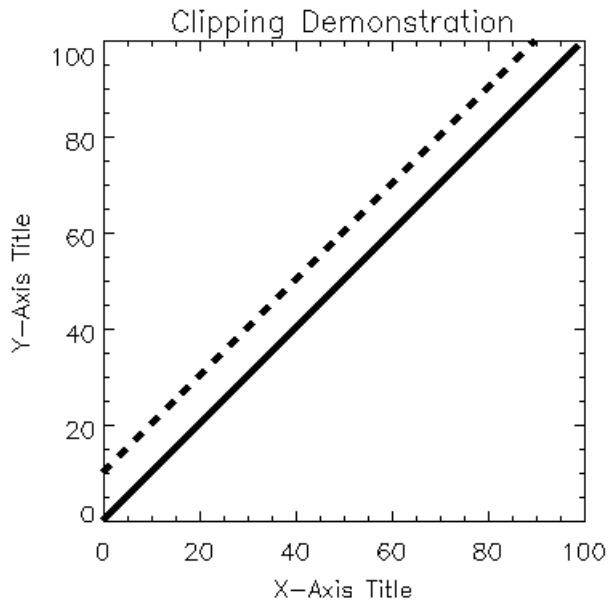


Figure 4-21 By default, graphics plotted with OPLOT are clipped by the boundary of the coordinate axes.

OPLOT with NoClip Keyword

In the next example, the *NoClip* keyword is added to the OPLOT command. This overrides the default clipping rectangle defined by !P.Clip, and the dotted line extends beyond the boundary of the coordinate axes.

```
PLOT, INDGEN(100)
OPLOT, INDGEN(100)+10, LineStyle=2, /NoClip
```

The effect of specifying the *NoClip* keyword in an OPLOT command is shown in *Figure 4-22*.

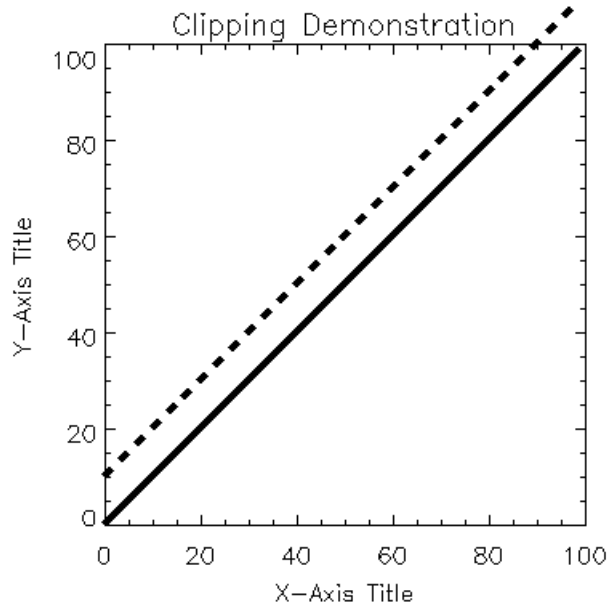


Figure 4-22 When the NoClip keyword is specified, the overplotted data is not constrained by default clipping rectangle (the region bounded by the coordinate axes).

OPLOT with Clip Keyword

Finally, the *Clip* keyword is used with OPLOT, and only the dotted line is clipped.

```
PLOT, INDGEN(100)

OPLOT, INDGEN(100) + 10, LineStyle = 2, $
      Clip = [25, 25, 75, 75]
```

Figure 4-23 shows the graphic with only the OPLOT line clipped.

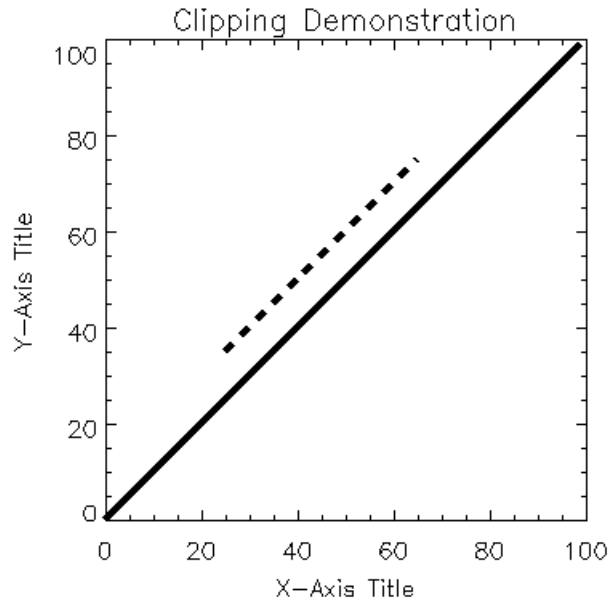


Figure 4-23 The Clip keyword is used to specify a clipping rectangle for OPLOT. Only the OPLOT data is clipped.

XYOUTS Default Clipping

When PLOTS, POLYFILL, or XYOUTS is called, clipping is disabled by default (that is, the value of the !P.Clip system variable is ignored). In this example, the text drawn by XYOUTS extends well beyond the coordinate axes into the device region.

```
PLOT, INDGEN(100)

XYOUTS, 60, 40, 'This is a Clipping Demonstration'
```

Figure 4-24 shows the result of the code. Note that the text specified by the XYOUTS procedure is not affected by any axis-boundary default clipping.

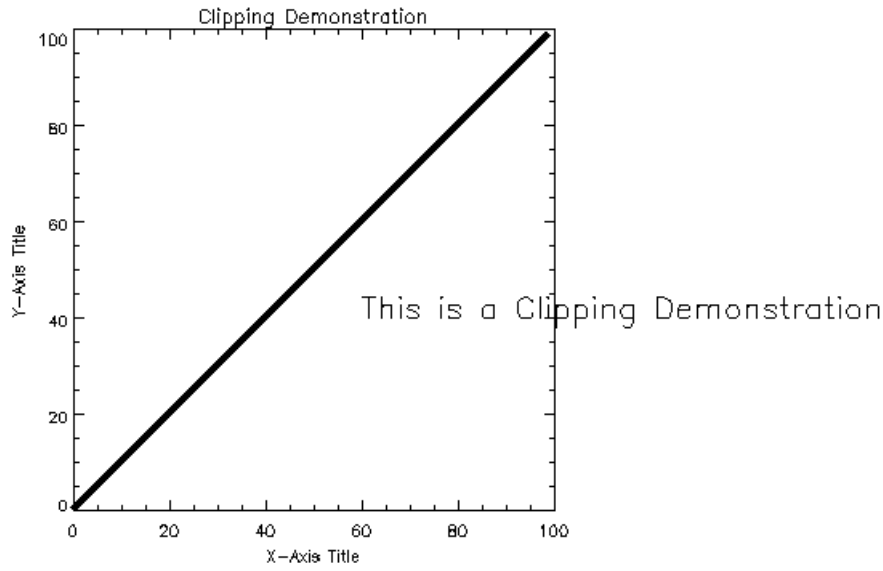


Figure 4-24 By default, clipping is disabled for PLOTS, POLYFILL, and XYOUTS. Here the text extends well past the default clipping region bounded by the coordinate axes.

XYOUTS with PClip Keyword

In the next example, the *PClip* keyword is used. This keyword overrides the default clipping condition for XYOUTS, PLOTS, and POLYFILL. *PClip* causes the value of !P.Clip to be recognized for these commands, and graphics (or text) are clipped at the boundary of the coordinate axes.

```
PLOT, INDGEN(100)
XYOUTS, 60,40, 'This is a Clipping Demonstration', /PClip
```

By using the *PClip* keyword with XYOUTS, the text is contained within the coordinate boundaries of the graphic as shown in *Figure 4-25*.

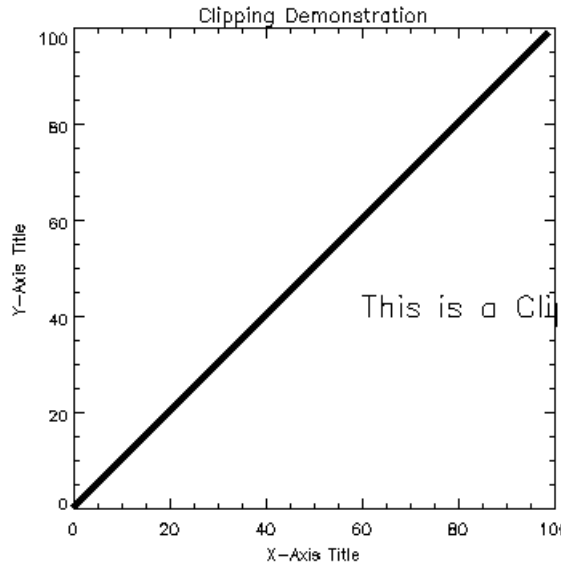


Figure 4-25 With the *PClip* keyword set, XYOUTS uses the value of the system variable !P.Clip as its clipping rectangle. In this case, !P.Clip was set by the PLOT command to the boundary of the coordinate axes.

XYOUTS with Clip Keyword

Finally, the *Clip* keyword is used with a XYOUTS. *Clip* works the same way with all graphics functions — it defines a clipping rectangle within the plot data region. In this example, the text drawn by XYOUTS is clipped everywhere outside the clipping rectangle.

```
PLOT, INDGEN(100)

XYOUTS, 60,40, Clip=[40,20,80,80], $
     'This is a Clipping Demonstration'
```

Figure 4-26 demonstrates the ability to clip the XYOUTS procedure text within the coordinate boundaries of the graphic using the *Clip* keyword.

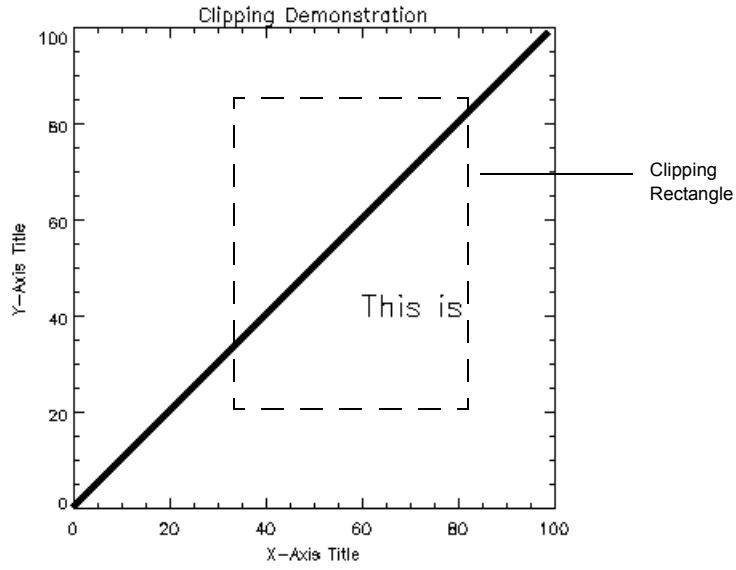


Figure 4-26 In this example, the Clip keyword is used with XYOUTS to specify a clipping rectangle. Only the text is clipped; the data is unaffected.

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.

```
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. Using
    ; the NoClip keyword ensures 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.

5

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 6, *Displaying Images*. The 3D plotting procedures include:

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

CONTOUR2, *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, CONTOUR2, and SURFACE use line graphics to depict the value of a two-dimensional array. As their names imply, CONTOUR and CONTOUR2 draw 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 4, *Displaying 2D Data*, also apply to CONTOUR, CONTOUR2, and SURFACE. The keywords and system variables discussed in this chapter are described in detail in the *PV-WAVE Reference*.

Differences Between CONTOUR and CONTOUR2

CONTOUR2 enhances PV-WAVE's contouring capabilities by adding scattered data plotting and sophisticated curve smoothing to produce more realistic contour lines than CONTOUR. These advantages are especially noticeable in smaller data sets.

When to Use CONTOUR2

- Use CONTOUR2 to plot 3D scattered data. Note that CONTOUR2 plots rectangular and irregularly gridded data using the same curve smoothing algorithm that it uses for scattered data.
- CONTOUR2 produces more "realistic" contours, especially for sparse data sets.
- CONTOUR2's *Fill* keyword simplifies the procedure for creating filled contours.

When to Use CONTOUR

- Because CONTOUR2's curve smoothing is computationally intensive, consider using CONTOUR if a shorter computing time is important. Remember that whenever you use CONTOUR, your data must define either a rectangular or curvilinear coordinate system.

When to Use either CONTOUR or CONTOUR2

- For regular or irregularly gridded data, you can use either the CONTOUR or CONTOUR2.

Drawing Contour Plots with the CONTOUR Procedure

NOTE The following sections describe how to use the CONTOUR procedure; however, most of the information applies to CONTOUR2 as well. The primary differences between CONTOUR and CONTOUR2 are listed in the previous section. For detailed information on these procedures, see the *PV-WAVE Reference*.

The CONTOUR procedures draw contour plots from data stored in a rectangular array. In their simplest form, these procedures make a contour plot given a two-dimensional array of *z* values. In more complicated forms, CONTOUR accept, 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.

Basic Usage

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*–axis 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 CONTOUR

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 CONTOURFILL procedure to shade closed contour regions with specified colors, as explained in *Filling Contours with Color* on page 96. The follow method is used if any of the following keywords is specified: *C_Annotation*, *C_Charsize*, *C_Labels*, *Follow*, *Path_Filename*, or *Spline*. In addition, the use of any of these keywords causes the contours to be labeled.

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 .

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 5-1 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.

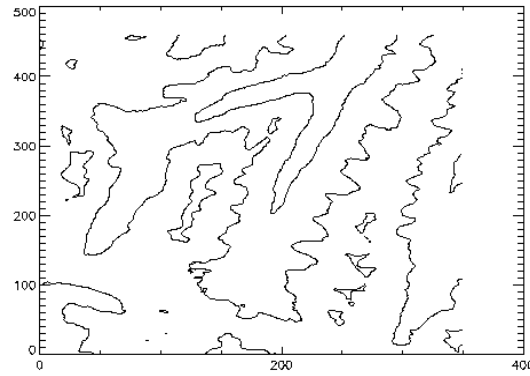


Figure 5-1 Simple contour plot of Maroon Bells.

- 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.
- Vectors *x* and *y* 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 5-2*. It was produced with the following 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.
```

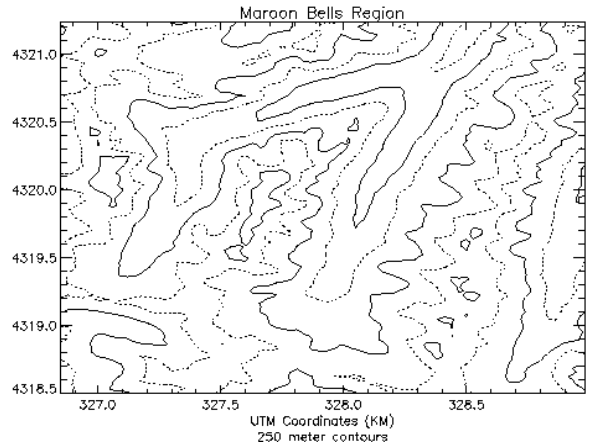


Figure 5-2 Improved contour plot.

Overlaying Images and Contour Plots

Figure 5-3 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.

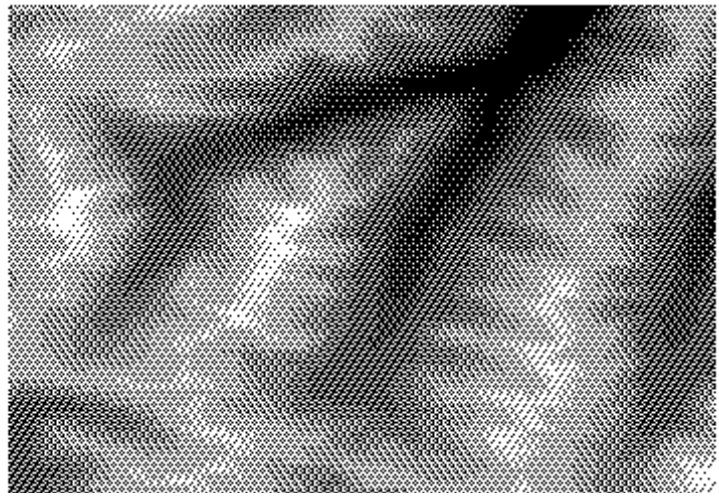


Figure 5-3 Maroon Bells data displayed as an image.

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 5-4*:

```
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.

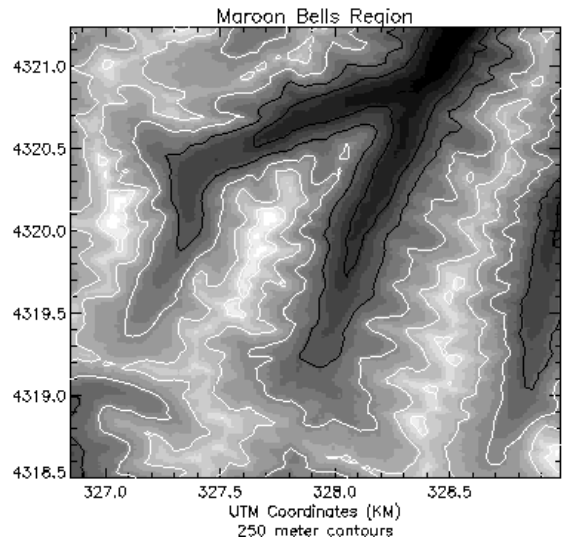


Figure 5-4 Overlay of image and contour plot.

Also, note that in *Figure 5-4* 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

There are two methods for overlaying images 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 the size of the plot window in device pixels.
py = !Y.Window * !D.Y_Vsize
sz = SIZE(a)
    ; The size of the 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 5-5* was produced using the statement:

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

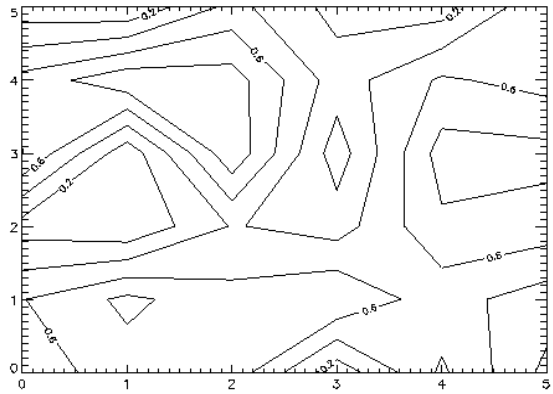


Figure 5-5 Simple labeled contour plot.

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 261. The result of this statement is shown in *Figure 5-6*.

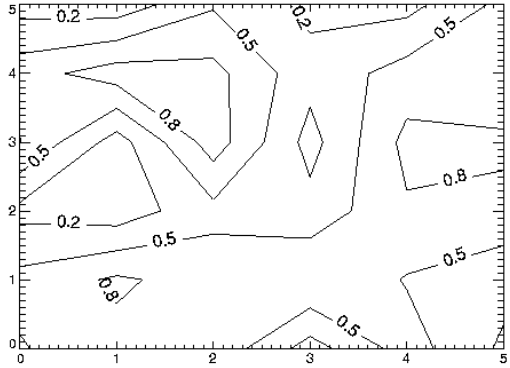


Figure 5-6 Label size and levels specified.

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 5-7*.

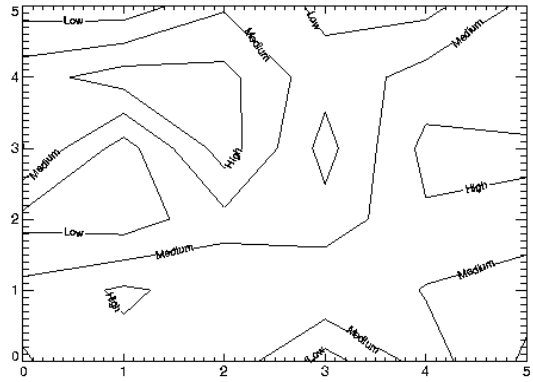


Figure 5-7 Explicitly specified labels.

Smoothing Contours

NOTE The CONTOUR2 algorithm produces smoothed contour lines by default.

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 5-8*. Compare it with the non-smoothed versions in *Figure 5-6* and *Figure 5-7*.

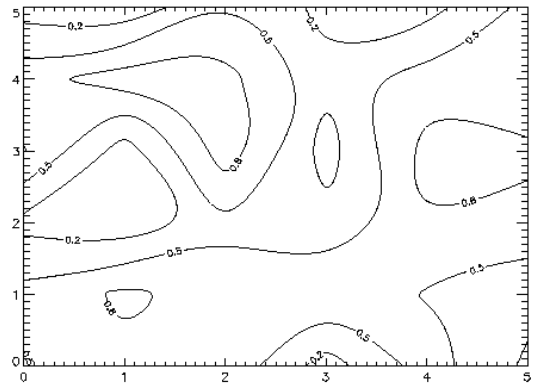


Figure 5-8 Contour plot with smoothing via cubic splines.

Filling Contours with Color

NOTE The following procedure applies primarily to CONTOUR. The CONTOUR2 procedure simplifies contour filling with a convenient *Fill* keyword. See the *PV-WAVE Reference* for information filling contours with CONTOUR2.

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 5-9*.

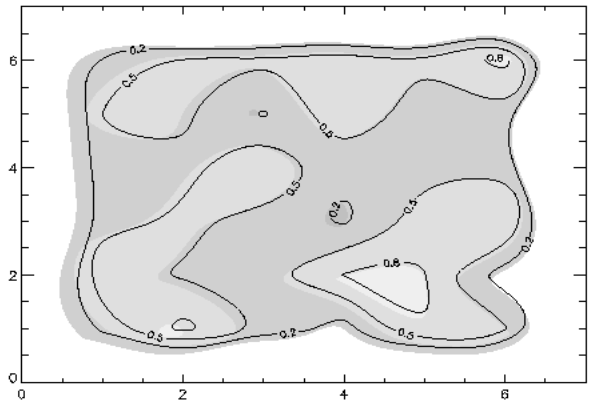


Figure 5-9 Filled contour plot with closed contours.

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 code illustrates the most basic call to SURFACE. It produces a two-dimensional Gaussian function and then calls SURFACE to produce *Figure 5-10*:

```
z = SHIFT(DIST(40), 20, 20)
    ; Create a 40-by-40 array, 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.
```

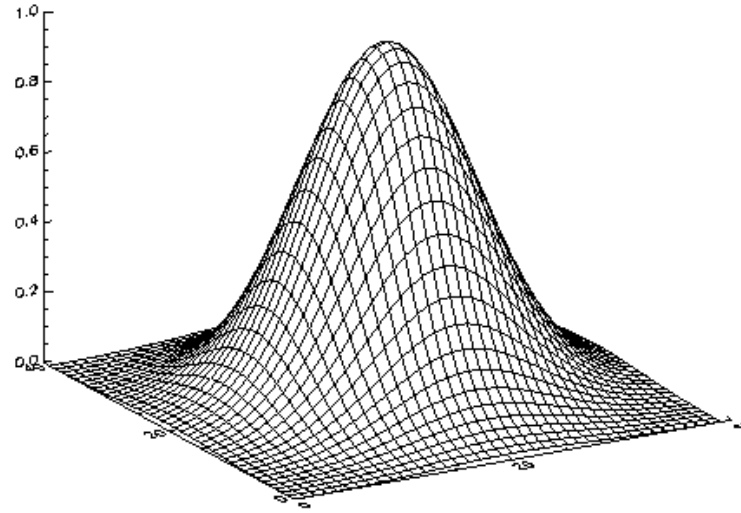


Figure 5-10 Simple SURFACE plot of a Gaussian.

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 .

Example of Drawing a Surface

Figure 5-11 illustrates the application of the SURFACE procedure to the Maroon Bells data discussed earlier in this chapter (see *Drawing Contour Plots with the CONTOUR Procedure* on page 84).

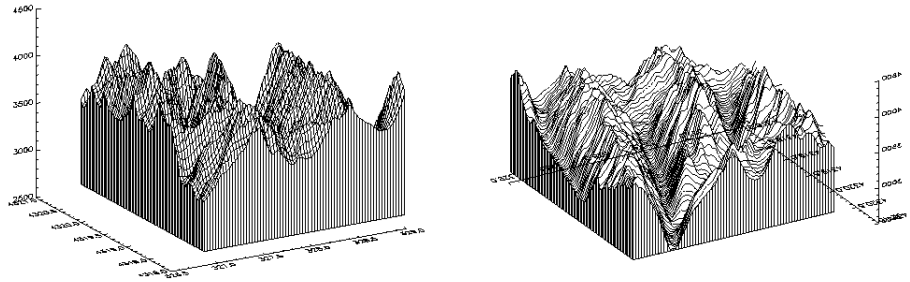


Figure 5-11 Maroon Bells surface plots.

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 86, 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.

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:

$$\begin{bmatrix} 1 & 0 & 0 & D_x \\ 0 & 1 & 0 & D_y \\ 0 & 0 & 1 & D_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotating Data

Rotation about the *x*-, *y*-, and *z*-axes is represented respectively by the three matrices:

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

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

$$R_z = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Clipping 3D Plots

Clipping consists of defining a specific region in a plot where existing data is plotted, and outside of which no data are shown. The general concept of clipping and the use of clipping for two-dimensional plots is discussed in Chapter 4, *Displaying 2D Data*. Keywords provided with the PV-WAVE graphics commands let you specify how clipping is done.

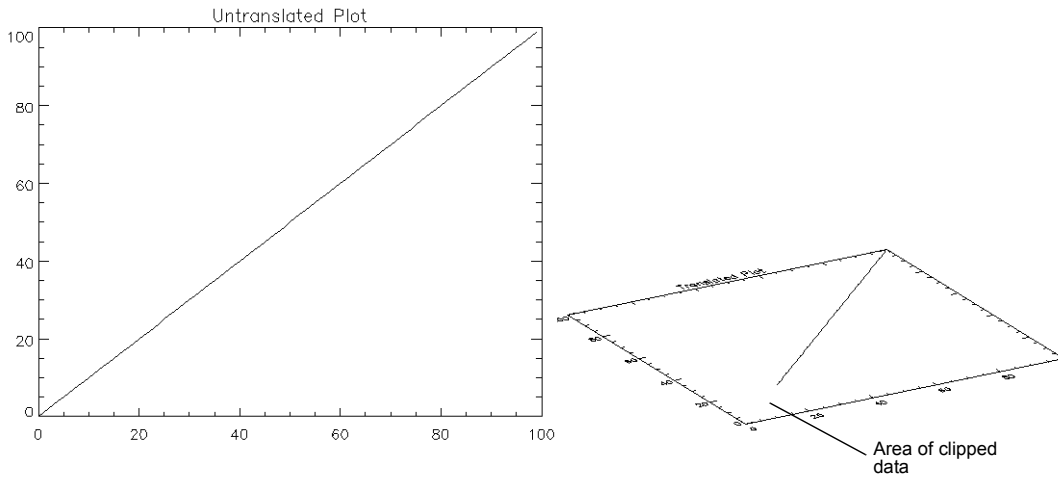
Notes on the Keywords and System Variables for 3D Clipping

When you call CONTOUR or SURFACE, the value of !P.Clip is *set* to a default value. This value depends on the current device. Setting the *Clip* keyword has no effect on !P.Clip.

Graphics keywords for controlling clipping are discussed in detail in Chapter 4, *Displaying 2D Data* of this manual, and are summarized in *Notes on the Keywords and System Variables* on page 73.

If you use clipping in a three-dimensional plot and you rotate the plot in three dimensions, you may notice some unusual clipping behavior. For instance, some part of your plot may be clipped in 3D when it was not clipped in 2D, as shown in the following example. Note the clipping "problem" encountered when these two plots are compared:

```
PLOT, INDGEN(100)
    ; Produce the first plot — a simple line plot.
SURFR
    ; Set up 3D translation.
PLOT, INDGEN(100), /T3D
    ; Produce the second plot — a simple line plot translated to 3D.
```



(a) Simple line plot, untranslated

(b) Translated line plot.

Figure 5-12 The picture on the left (a) shows a simple line plot. In (b), the same plot is shown translated in 3D. The translated plot appears to be missing some data near the origin.

The key to clipping in three dimensions is to remember that the !P.Clip system variable defines a default clipping rectangle that a) is always in device coordinates and b) cannot be translated to 3D coordinates.

The following figure shows clearly why the rotated graphic was clipped by the default clipping rectangle.

```
PRINT, !P.Clip
   90      72      613      476      0      6
```

; Show the coordinates of the default clipping rectangle defined by !P.Clip.

In this figure, a dashed line was drawn connecting the coordinates defining the corners of the default clipping rectangle. The coordinates of these corners were taken directly from !P.Clip. The dashed lines show clearly the boundary of this clipping rectangle. Part of the data (near the origin) falls outside this rectangle, and that is why it is clipped.

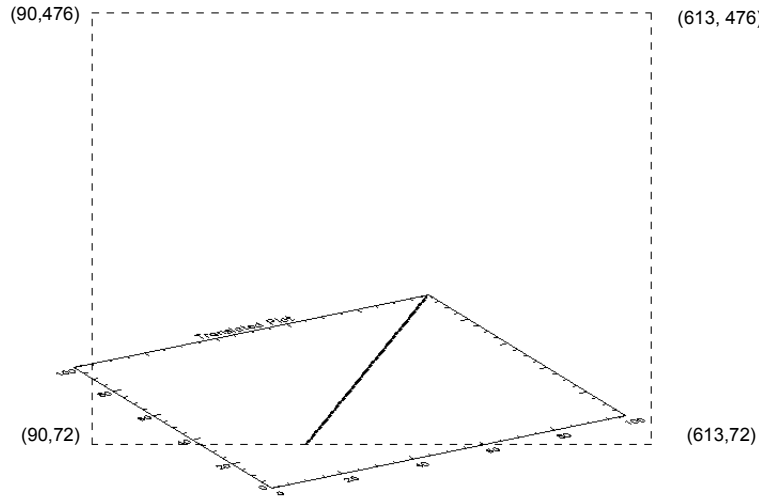


Figure 5-13 The data in a translated graphic can be clipped by the default clipping rectangle, which cannot be rotated. This unwanted clipping behavior can be avoided by adding the /NoClip keyword to the command that produces the rotated graphic.

You can avoid unwanted clipping of translated plots by adding the *NoClip* keyword to the graphics command:

```
PLOT, INDGEN(100), /T3D, /NoClip
```

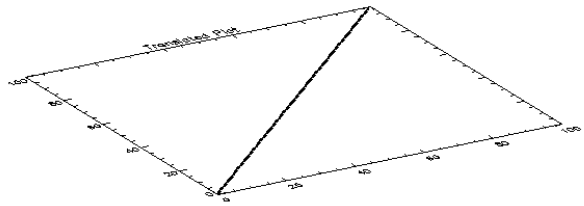
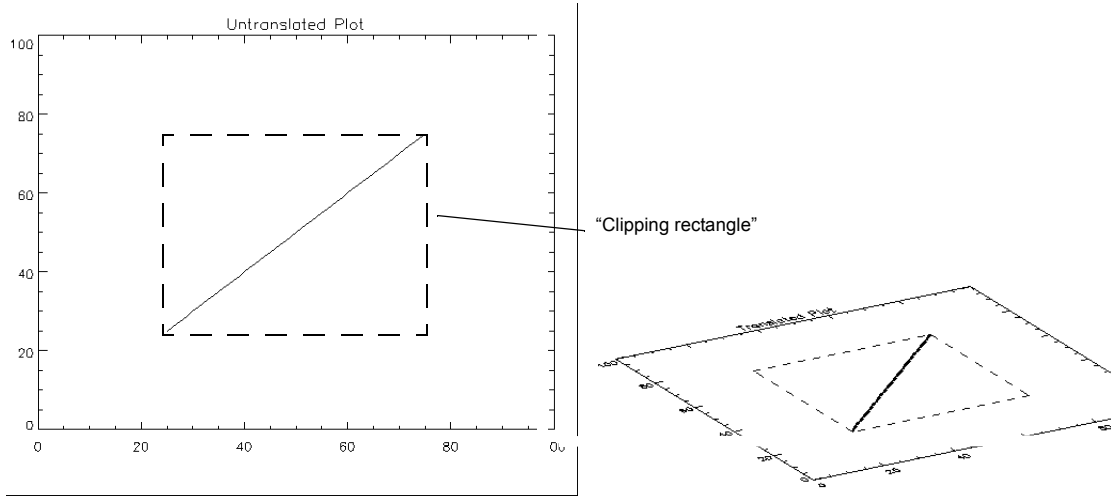


Figure 5-14 The same translated plot is produced, but this time /NoClip is specified on the command line. The data near the origin is not clipped.

The clipping rectangle defined by the *Clip* keyword is translated along with the plot for which it is defined. You can use *Clip* to modify a plot, and the clipping is preserved whenever you translate or rotate the plot in 3D, as the following example shows:

```
PLOT, INDGEN(100), Clip=[25,25,75,75]
SURFR
PLOT, INDGEN(100), Clip=[25,25,75,75], /T3D
```



(a) Untranslated, clipped plot

(b) Translated, clipped plot

Figure 5-15 The "clipping rectangle" defined by the Clip keyword is translated/rotated in 3D along with the rest of the graphic. The clipping rectangle does not appear on an actual plot; it is shown here for illustration purposes only.

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 46, 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 in *Overview of Homogeneous Coordinates* on page 101, 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 46.

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 = [ -Xmin, 1 ] / (Xmax - Xmin)
!Y.S = [ -Ymin, 1 ] / (Ymax - Ymin)
!Z.S = [ -Zmin, 1 ] / (Zmax - Zmin)
```

- 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 draws 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]
    ; The 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 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.
```

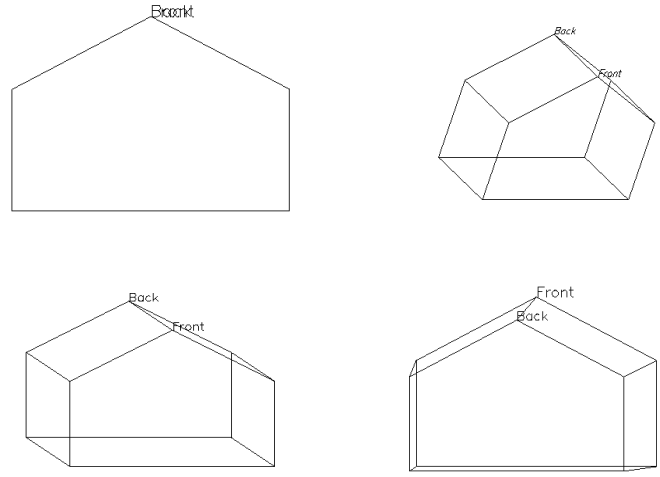


Figure 5-16 Illustration of different 3D transformations. From upper left: No rotation, plain projection; Rotation of 30 degrees about both the *x*- and *y*-axes, plain projection; Oblique projection, factor = 0.5, angle = –45; and in the bottom right, 30 degrees rotation with the eye at 50.

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.

The PLOT procedures output graphs in the XY plane at the normal coordinate *z* value given by the keyword *ZValue*. If *ZValue* is not specified, the plot is drawn at the bottom of the unit cube, at *z*=0.

CONTOUR draws axes at Z=0, and 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

You can combine the output of SURFACE with the other graphics procedures. The keyword *Save* causes SURFACE to save the graphic transformation it used in !P.T. Then, when CONTOUR or PLOT are called with the keyword *T3d*, their output is transformed with the same projection.

For example, *Figure 5-17* illustrates SURFACE combined with CONTOUR. In essence, this a combination of *Figure 5-2* and *Figure 5-11*. Using the same variables as discussed in *Drawing Contour Plots with the CONTOUR Procedure* on page 84 and *Drawing a Surface* on page 98, this figure was produced with the following statements:

```
SURFACE, c, x, y, SKIRT=2650, /Save
    ; Make the mesh as in Figure 5-11.

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 5-2. Specify T3D to align
        ; with Surface, at ZVALUE of 1.0. Suppress clipping as the plot
        ; is outside the normal plot window.
```

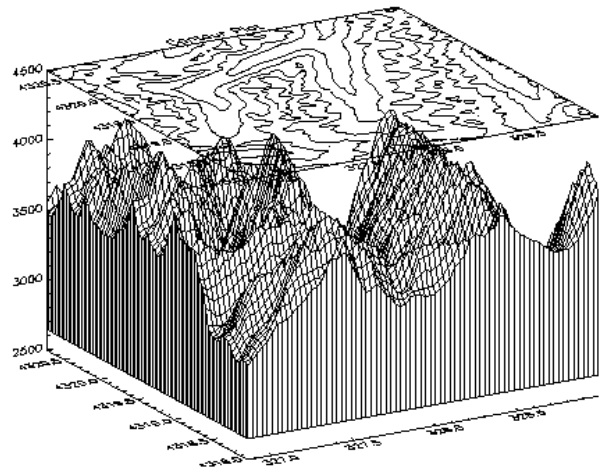


Figure 5-17 Combining CONTOUR with SURFACE, Maroon Bells data.

Even More Complicated Transformations are Possible

Figure 5-18 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.

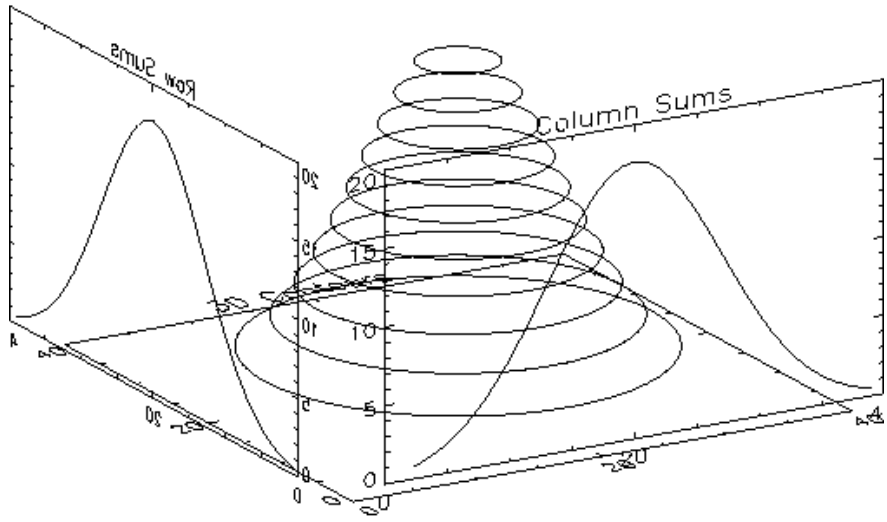


Figure 5-18 Example of using PLOT and CONTOUR with a 3D transform.

The code used to draw *Figure 5-18* 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-axes,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 5-19*, using the transformation techniques described in the previous section.

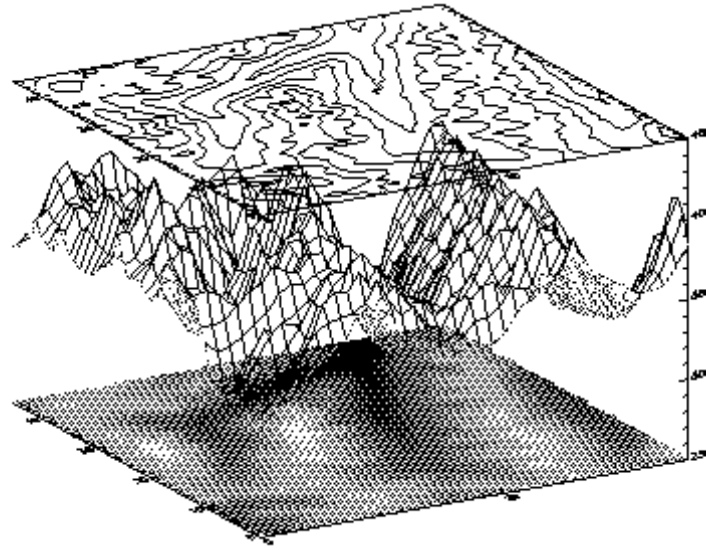


Figure 5-19 Using the SHOW3 procedure to overlay an image, surface mesh, and contour.

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 in *Converting from 3D to 2D Coordinates* on page 108, 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 98, 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 5-20* illustrates the application of SHADE_SURF, with light source shading, to the two-dimensional Gaussian, z, used to produce *Figure 5-10* on page 99. This figure was produced by the statement:

```
SHADE_SURF, z
```

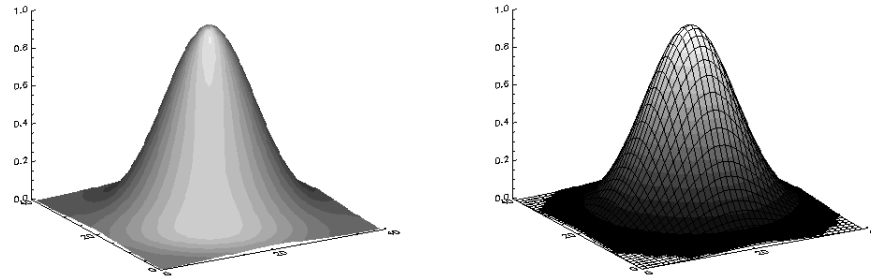


Figure 5-20 Shaded representations of two-dimensional Gaussian.

The right half of *Figure 5-20* 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 5-21 shows the Maroon Bells data, also shown in the right half of *Figure 5-11* on page 100, 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.

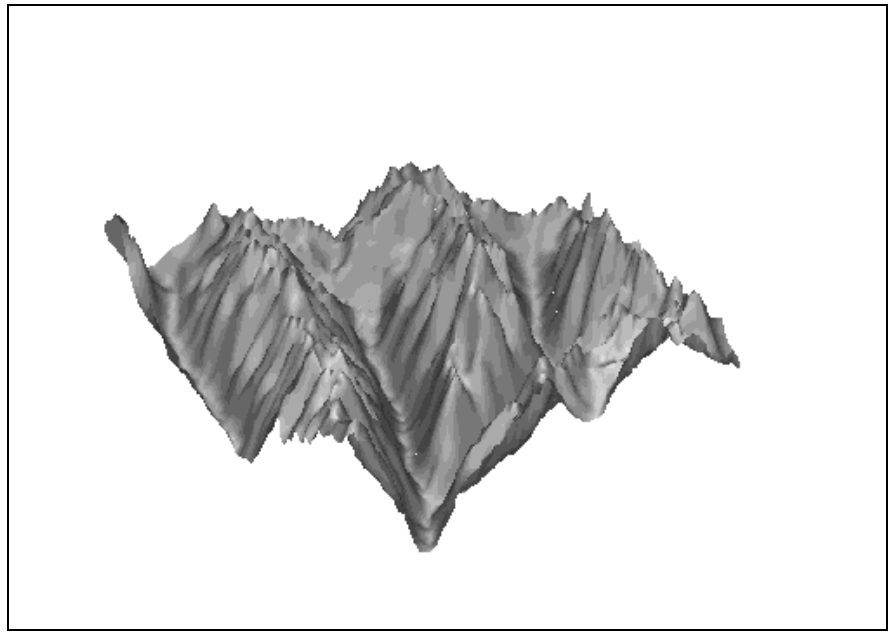


Figure 5-21 Maroon Bells data shown as a shaded surface.

6

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 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 4, *Displaying 2D Data* and Chapter 5, *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.

For details on the keywords for a particular routine, see the routine's description in the *PV-WAVE Reference*.

Windows USERS Because Windows NT reserves 20 out of the available 256 colors, you might achieve better results displaying color images with the TVSCL procedure. TVSCL automatically scales the color intensities to the full range of available colors.

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 the TV and TVSCL routines. 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

$X = X_{dim} Position_{\text{modulo} Nx}$ = 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:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

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 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 re-sampling.

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 the REBIN function, 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 re-sampled, *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 bi-linear interpolation when magnifying, and neighborhood averaging when reducing. Bi-linear 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 bi-linear interpolation:

```
TV, REBIN(area, 512, 512)
```

or without bi-linear 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 bi-linear 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 variable. One use for this capability is to build up a complex display using many statements, and then read the resulting image back as a single unit for storage in a file.

NOTE For colors to work properly on 24-bit true-color, TVRD(TRUE=3) is REQUIRED. See the PV-WAVE Reference manual for more information.

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.

Examples 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.
```

The following example copies part of an image, then resizes and displays the copied portion:

```
x=dist(300)
tvscl,x
x_rd=tvrd(0,0,150,150)
erase
tv,x_rd
tv,rebin(x_rd,300,300)
```

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 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*).

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 color table B–W Linear (standard color table number 0).

UNIX and OpenVMS USERS On a monochrome display, by default, color images are dithered. For more information about dithering, see *Displaying Images on 24-bit Devices (UNIX/OpenVMS)* on page 132.

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 .

If you are interested in seeing a more complete discussion of color systems, refer to *Understanding Color Systems* on page 275.

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 122.

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 281.

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`. These tables are located in:

  (UNIX) `<wavedir>/bin`

  (OpenVMS)`<wavedir>:[BIN]`

  (Windows) `<wavedir>\bin`

  Where `<wavedir>` is the main PV-WAVE directory.

- **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 280. For more information about creating custom color tables that emphasize some special trend or effect, see *Modifying the Color Tables* on page 283.

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 286.

Not all Color Images are True-color Images (UNIX/ OpenVMS)

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 the *PV-WAVE Reference*.

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.

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

NOTE 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 24-bit Devices (UNIX/OpenVMS)* on page 132.

For more information about the different ways that image data may be stored, refer to the section *Input and Output of Image Data* in Chapter 8 of the *PV-WAVE Programmer's Guide*.

Displaying Images on Monochrome Devices (UNIX/OpenVMS)

Windows USERS This section on monochrome devices does not pertain to the Windows version of PV-WAVE.

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 (UNIX/OpenVMS)

Windows USERS This section on 24-bit devices does not pertain to the Windows version of PV-WAVE.

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, see the *PV-WAVE Reference*.

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 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 (UNIX/OpenVMS)* on page 130.

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 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 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 an OpenVMS environment, as well:

```
DEVICE, Direct_Color=24
    ; Define a 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 286, or refer to the STRETCH procedure 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:

$$P_i = \sum_{j=0}^{i} H_j$$

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 6-1* 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.

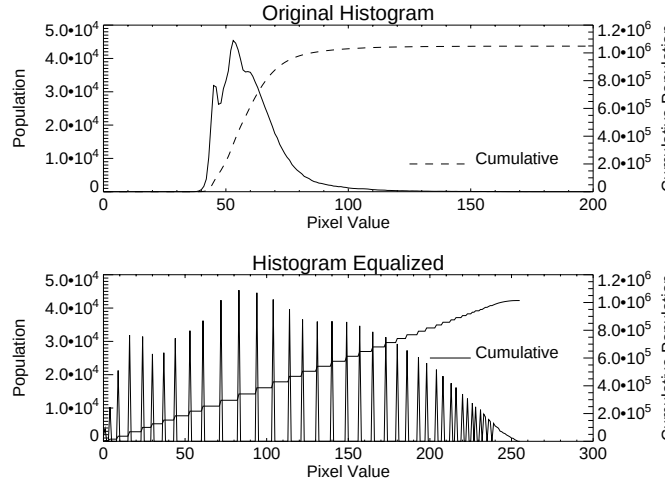


Figure 6-1 Histograms of the original and histogram-equalized images.

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 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:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

which approximates a two-dimensional triangle is easily implemented using the CONVOL function by the statement:

```
TVSCL, CONVOL(A,[[1,2,1],[2,4,2],[1,2,1]])
```

The first parameter, `A`, in the CONVOL function call, 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 notation for a 3-by-3 array containing the 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 with the MEDIAN function:

```
TVSCL, MEDIAN(A, 3)
```

Figure 6-2 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.

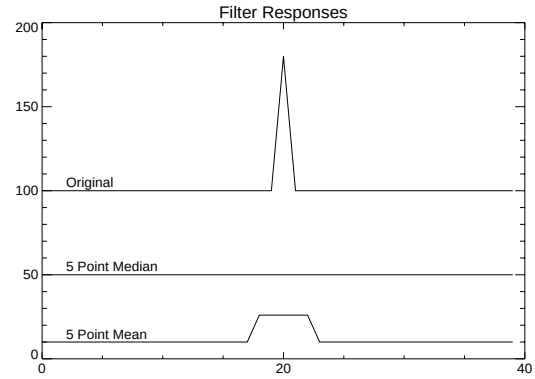


Figure 6-2 Signal response using median and mean filtering.

Image Sharpening

This section discusses some image sharpening methods. 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:

$$G(F_{x,y}) = \left|F_{x,y} - F_{x+1,y+1}\right| + \left|F_{x+1,y} - F_{x,y+1}\right|$$

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:

$$S_{x,y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

where the notation,

$$\begin{bmatrix} w_0 & w_1 & w_2 \\ w_3 & w_4 & w_5 \\ w_6 & w_7 & w_8 \end{bmatrix}$$

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:

$$\begin{vmatrix} -\left(\frac{1}{9}\right) & -\left(\frac{1}{9}\right) & -\left(\frac{1}{9}\right) \\ -\left(\frac{1}{9}\right) & \frac{8}{9} & -\left(\frac{1}{9}\right) \\ -\left(\frac{1}{9}\right) & -\left(\frac{1}{9}\right) & -\left(\frac{1}{9}\right) \end{vmatrix}$$

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 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 FFT 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:

$$D_{x,y} = \sqrt{(x'/n)^2 + (y'/n)^2}$$

where:

$$x' = \begin{cases} x & \text{if } (x \le n/2) \\ n - x & \text{otherwise} \end{cases}$$

$$y' = \begin{cases} x & \text{if } (x \le n/2) \\ n - y & \text{otherwise} \end{cases}$$

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.)

$$filter_{u,v} = \frac{1}{1 + [D_{u,v}/D_0]^{2n}}$$

```
filter = 1/(1+(D/D0) ^(2*N))
```

- Butterworth high pass filter of order *n*, cutoff at *D – 0*:

$$filter_{u,v} = \frac{1}{1 + [D_0/D_{u,v}]^{2n}}$$

```
filter = 1/ (1 + (D0/D) ^ (2*N))
```

- Butterworth bandpass filter, order *n*, center frequency is C, width of D_0:

$$filter_{u,v} = \frac{1}{1 + [(D_{u,v} - C)/D_0]^{2n}}$$

```
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 6-3* 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.

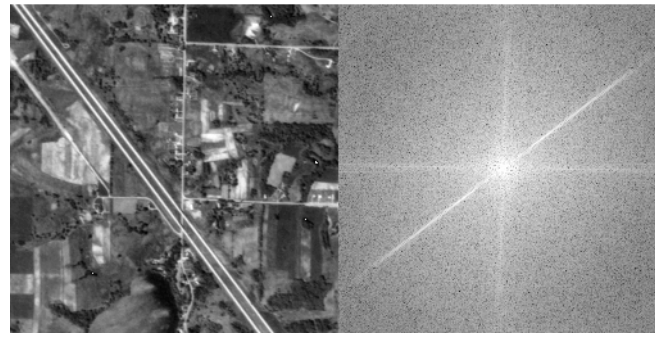


Figure 6-3 An aerial photograph and its Fourier spectrum.

The Fourier spectrum is also displayed in *Figure 6-4* as a surface plot.

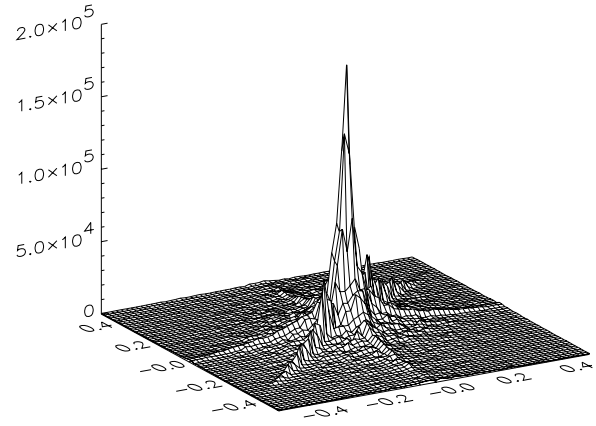


Figure 6-4 Surface plot of the aerial photograph's Fourier spectrum.

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 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 6-3* 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. The calling sequence for the ROTATE function is

ROTATE*(image, rotation)*

where *image* is the input array and *rotation* is an integer value specifying one of the eight possible combinations of axis interchange and reversal.

Example of ROTATE Function Usage

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*:

$$u = a(x, y) = \sum_{i=0}^{N} \sum_{j=0}^{N} C_{i,j} x^j y^i$$

$$v = b(x, y) = \sum_{i=0}^{N} \sum_{j=0}^{N} D_{i,j} x^j y^i$$

Either the nearest neighbor or bi-linear interpolation methods may be selected. The calling sequence and a brief description of the input parameters for the POLY_2D function is as follows.

output_image = POLY_2D(*image*, *c*, *d* [, *interp* [, dim_x, dim_y]])

Where

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 $x^i y^i$. The POLYWARP 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 bi-linear interpolation, otherwise the nearest neighbor method is used. For the linear case (i.e. $N = 1$) bi-linear 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 output keyword parameter *Missing* 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. For more detailed information on the POLY_2D function, refer to the function description in the *PV-WAVE Reference*.

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. bi-linear 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 6-5*.

The equations relating the *(u, v)* coordinates to *(x, y)* are:

$$u_i = c_0 + c_1 y_i + c_2 x_i + c_3 y_i x_i \quad , \quad i = 0, 1, 2, 3$$

$$v_i = d_0 + d_1 y_i + d_2 x_i + d_3 y_i x_i \quad , \quad i = 0, 1, 2, 3$$

We can write the four equations for u_i as:

$$U = ZC$$

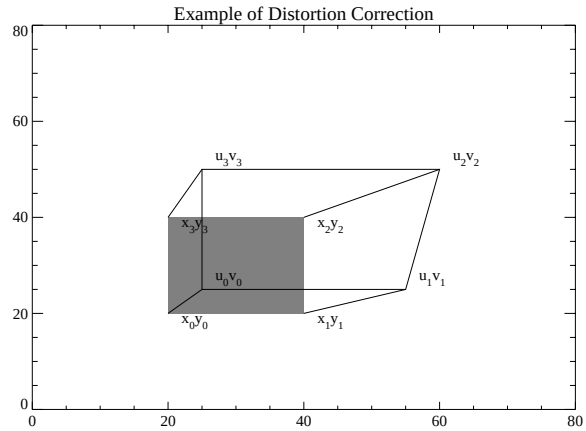


Figure 6-5 Example of geometric distortion.

where $C = [c_0, c_1, c_2, c_3]$, and

$$Z = \begin{bmatrix} 1 & y_0 & x_0 & x_0 y_0 \\ 1 & y_1 & x_1 & x_1 y_1 \\ 1 & y_2 & x_2 & x_2 y_2 \\ 1 & y_3 & x_3 & x_3 y_3 \end{bmatrix}$$

Solving for C and D:

$$C = Z^{-1}U \quad D = Z^{-1}V$$

The 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, one 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 6-6 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.

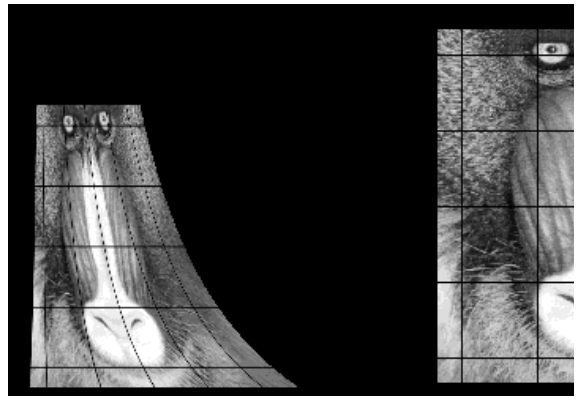


Figure 6-6 Correcting a geometric transformation.

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 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 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.

7

CHAPTER

Rendering Techniques

PV-WAVE can render 3D geometric and volumetric data through the use of your graphics hardware using PV-WAVE's integration with the Visualization Toolkit (VTK) or through software using rendering algorithms built into PV-WAVE.

Besides faster 3D graphics, VTK supports techniques such as the ability to rotate and manipulate 3D views in real-time, do real-time displays of volumetric data, and enhanced functionality for static 3D rendering in 24-bit color. The end user can gain insight about their data though interacting, visualizing and analyzing data real time with the VTK utilities.

The advanced rendering routines built into PV-WAVE use software to display geometric and volumetric data. Three rendering techniques are available including; polygon and volume rendering and ray-tracing techniques. In addition, gridding algorithms and coordinate conversion functions are available to prepare data for rendering.

Hardware Rendering

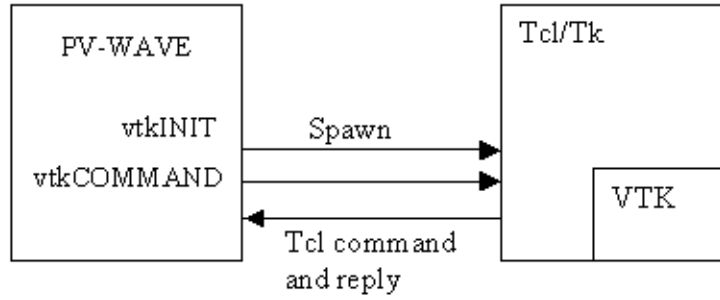
Introduction

PV-WAVE users can create high quality, interactive graphics through the use of the PV-WAVE link to the Visualization Toolkit (VTK). The Visualization Toolkit is an Open Source toolkit for creating both simple and complex visualizations in 3D using OpenGL, a low-level software interface to graphics hardware, for high-performance, accelerated graphics. In addition to convenience routines that have been written to access some of the more common VTK utilities, all of the

functionality available in the Visualization Toolkit is available to PV-WAVE users. The two products complement each other well. PV-WAVE excels at data access, data manipulation, numerical algorithms, data filtering, user interface development, and many interactive 2D graphical tasks. The Visualization Toolkit is a best-of-breed tool for creating complex 3D visualizations. Together they provide a simple and quick way to build tools for Visual Data Analysis.

The VTK toolkit was integrated into PV-WAVE through the use of a Tcl shell as an intermediary. Tcl is a popular scripting language, see *http://www.scriptics.com* for more information. The VTK toolkit already has a binding for the Tcl language. PV-WAVE spawns a Tcl shell and communicates with it using sockets, sending VTK commands formatted for Tcl.

The following diagram shows how the two routines vtkINIT and vtkCOMMAND are used to create the Tcl shell and send commands to it, including VTK commands:



The vtkCOMMAND procedure is used in PV-WAVE to send individual commands to the shell, and higher level wrappers have been built around this to allow many common PV-WAVE plotting commands to be accessed in a manner similar to existing PV-WAVE commands. Some of these commands include vtkWINDOW, vtkLIGHT, vtkPLOTS, vtkSURFACE, and vtkPOLYSHADE. The full power of the VTK visualization pipeline can still be accessed but requires specific knowledge of the VTK objects and methods.

A set of commands for packaging data into one of five supported data types in VTK is also provided, allowing data to be easily sent from PV-WAVE to VTK.

Many procedures have a "*Name*" keyword which allows you to either choose a name for the object you are creating or return a generated one. This acts as a bridge between the high-level PV-WAVE wrappers and low-level VTK functionality.

Thus you can create a vtkSURFACE object and later use low-level VTK commands to change specific properties of this object.

Currently the VTK windows can not be integrated into a user interface created with PV-WAVE widgets. However, widgets can be used to size, position and annotate an object in a VTK window by passing keywords from the widget to the vtkWINDOW.

NOTE the PV-WAVE link to the Visualization Toolkit is available on Windows and most UNIX platforms

---

Additional Information

You can download the Visualization Toolkit and reference documentation from <http:// public.kitware.com>. You can also purchase the following VTK manuals at this site:

The Visualization Toolkit User's Guide
William J. Schroeder, Lisa S. Avila,
Kenneth M. Martin, William A. Hoffman,
C. Charles Law
380 pages, CD-ROM with software/data
ISBN 1-930934-05-X
Kitware, Inc.

The Visualization Toolkit
An Object-Oriented Approach To 3D Graphics
Will Schroeder, Ken Martin, Bill Lorensen
646 pages, 40 color pages, CD-ROM with software/data
ISBN 0-13-954694-4
Prentice Hall

The Visualization Toolkit User's Guide is the best manual for details on using VTK. *The Visualization Toolkit* manual has more general information on computer graphics and scientific visualization.

Demonstration Programs

There are three sources for advanced rendering demonstration programs:

- You can run the demonstration programs and look at the code in:

(UNIX) `<vni_dir>/vtk-3_2/demo`

(Windows) `%VNI_DIR%\vtk-3_2\demo`

Where <VNI_DIR> is the main Visual Numerics directory.

These routines can be easily modified to work with your own data.

Initializing VTK and Managing VTK Windows

The routine vtkINIT is used to spawn a Tcl process through which VTK commands may be sent. The spawned process will continue executing until vtkCLOSE is called. Repeatedly calling vtkINIT will not cause multiple Tcl processes to be created; only one will be allowed to run at any given time for a PV-WAVE session. You can use the /Print parameter to vtkINIT to cause debug information from the Tcl process to be logged in the PV-WAVE console. Most VTK wrapper routines, with the exception of vtkCOMMAND and the dataset creation routines, will automatically call vtkINIT for you if you have not done so manually.

NOTE You must call vtkCLOSE before exiting PV-WAVE or else an orphaned Tcl process will be left running on your machine. This process will only go away if you manually kill it or log off of your computer.

To create a VTK window in which 3D OpenGL graphics can be rendered, use the vtkWINDOW command. It is used in a manner similar to the WINDOW command for PV-WAVE windows. You can specify a window index to be used to refer to this window, or use /Free to allow an unused index to be chosen for you. There are two important keywords for vtkWINDOW that affect how 3D objects are displayed in it: /NoRender and /NoInteract.

Normally VTK windows operate much like PV-WAVE windows, in that as you issue commands to add plots, annotation, axes and other objects to the window, the results are immediately rendered and displayed. For performance reasons you do not always want to do this for VTK windows, and would rather add all of the objects to be rendered before actually rendering and displaying the scene. If you specify /NoRender in a call to vtkWINDOW, then you are turning off automatic rendering until you explicitly call the vtkRENDERWINDOW routine. Use of the /NoRender routine does not affect calls to the low-level vtkCOMMAND routine.

By default VTK windows have a set of mouse interactions built into them. This allows you to rotate, zoom, and pan your view interactively. If you do not want these default interactions then specify /NoInteract with vtkWINDOW and any changes in camera view will be under programmatic control only. The default mouse interaction for VTK windows including the following features:

- ***Keypress j / Keypress t***: toggles between joystick (position sensitive) and trackball (motion sensitive) styles. In joystick style, motion occurs continuously as long as a mouse button is pressed. In trackball style, motion occurs when the mouse button is pressed and the mouse pointer moves.
- ***Keypress c / Keypress o***: toggles between camera and object (actor) modes. In camera mode, mouse events affect the camera position and focal point. In object mode, mouse events affect the actor that is under the mouse pointer.
- ***Button 1***: rotates the camera around its focal point (if camera mode) or rotate the actor around its origin (if actor mode). The rotation is in the direction defined from the center of the renderer's viewport towards the mouse position. In joystick mode, the magnitude of the rotation is determined by the distance the mouse is from the center of the render window.
- ***Button 2***: pans the camera (if camera mode) or translate the actor (if object mode). In joystick mode, the direction of pan or translation is from the center of the viewport towards the mouse position. In trackball mode, the direction of motion is the direction the mouse moves. (**Note**: with 2-button mice, pan is defined as <Shift>-Button 1.)
- ***Button 3***: zooms the camera (if camera mode) or scale the actor (if object mode). Zoom in/increase scale if the mouse position is in the top half of the viewport; zoom out/decrease scale if the mouse position is in the bottom half. In joystick mode, the amount of zoom is controlled by the distance of the mouse pointer from the horizontal centerline of the window.
- ***Keypress r***: resets the camera view along the current view direction. Centers the actors and moves the camera so that all actors are visible.
- ***Keypress s***: modifies the representation of all actors so that they are surfaces.
- ***Keypress w***: modifies the representation of all actors so that they are wireframe.

The following routines operate exactly like their PV-WAVE counterparts to manage windows:

vtkERASE	Erases the VTK window to its background color or specify a new background color.
vtkWSET	Makes a VTK window the current one to be drawn to in subsequent calls to VTK wrapper routines.

vtkWDELETE	Deletes a VTK window (but does not close the Tcl process, you still must call vtkCLOSE before exiting PV-WAVE).

Saving the Contents of VTK Windows

In order to save the contents of VTK windows, you can use the procedure vtkPPMWRITE. This causes the current contents of the selected or current VTK window to be saved to a file in PPM (Portable PixMap) format. This is the only format supported by VTK for saving the contents of VTK windows. The corresponding function vtkPPMREAD can be used to read a PPM file and return a 24 bit image that can be displayed in PV-WAVE. The routine vtkTVRD functions much as the PV-WAVE TVRD routine but is a wrapper to calls to vtkPPMWRITE and vtkPPMREAD.

NOTE The vtkPPMWRITE procedure will save exactly what you see on your screen for a VTK window, including the contents of any windows that are partially or fully obscuring the VTK window. You must make sure the VTK window is fully visible for vtkPPMWRITE or vtkTVRD to work properly. This is a result of the underlying VTK implementation and there is not a way around this at present.

High-level Interface Routines

The following routines are PV-WAVE wrappers which mimic the functionality of common PV-WAVE graphics routines. The source code for these routines are available as PV-WAVE procedures and act as good examples of using the low-level VTK functionality.

vtkLIGHT	Adds a light source to a VTK scene
vtkCAMERA	Adds a customized camera to a VTK scene
vtkAXES	Adds a set of 3 axes to a VTK scene
vtkPLOTS	Plots 3D lines and points
vtkTEXT	Adds text annotation to a VTK scene
vtkPOLYSHADE	Displays vertex/polygon lists which describe polygonal objects
vtkSURFACE	Plots shaded and wireframe surfaces with axes

vtkSCATTER	Plots points in 3D with axes

Specifying Color

In the PV-WAVE wrappers for VTK there are a number of ways to specify colors. The VTK windows always display in 24-bit color, although we can use the PV-WAVE color table values as we will see. For parameters that expect as input a single color value, the color can be specified in any one of the following ways (in this case for the color red):

'red'	See the file <*vni*>`/vtk-3_2/lib/vtkcolornames.pro` for a complete list of supported color names, where <*vni*> is the path to the PV-WAVE installation.
'FF0000'XL	A long integer hexadecimal value specifying the 24-bit color.
[1.0, 0.0, 0.0]	A three element vector of normalized floating point values specifying the red, green, and blue components of the color.
[1.0, 0.0, 0.0, 1.0]	A four element vector of normalized floating point values specifying the red, green, blue, and alpha components of the color. The alpha component is the transparency where 0.0 is completely transparent and 1.0 is opaque. Transparency is not supported for all color specifications and will be ignored where not available.
2	If a short byte or short integer value is passed, the RGB color is obtained from the corresponding entry in the current PV-WAVE color table. In this case when `TEK_COLOR` has been called, color index 2 is red.

For parameters that expect as input a 1D or 2D array of color values, such as the *Shades* keyword for vtkSURFACE or *Color* keyword for vtkPOLYSHADE, the color can be specified as arrays of the above. For example for vtkSURFACE we could pass a 2D array of short integers to make use of the PV-WAVE color table, or a (3, *m*, *n*) array of float values between 0.0 and 1.0.

Low-level Interface Routines

For many users, the above high-level VTK wrapper routines will provide sufficient functionality for creating 3D charts similar to what is already available in PV-WAVE, but now using accelerated OpenGL graphics. Others may want to make use of the vast amount of functionality available in VTK including source code developed by other VTK users. All of this is possible using the low-level interface

provided to VTK through PV-WAVE. If you intend to use the low-level functionality available in VTK you will need to obtain documentation on VTK. See the references in the INTRODUCTION (page 155) for more details.

vtkCOMMAND

Most low-level VTK functionality is accessed using vtkCOMMAND, which simply sends a text string containing any valid Tcl or VTK wrapper command to the spawned Tcl process. You can send individual commands or even invoke pre-written Tcl scripts through the "include" Tcl command.

Creation and access to VTK objects via Tcl (and thus vtkCOMMAND from PV-WAVE) is made using this convention:

To create a VTK object in Tcl:

```
Vtk_class_name my_vtk_variable name
```

To call a method of a VTK object in Tcl:

```
My_vtk_variable_name method_name param_1 param_2 param_n
```

For example to create a VTK light source and set the color to red you could use these commands from PV-WAVE:

```
vtkCOMMAND, 'vtkLight my_light'
vtkCOMMAND, 'my_light SetColor 1.0 0.0 0.0'
```

The use of the Tcl wrappers for VTK commands are documented in the references mentioned in the *Introduction* on page 155 and in reference pages available with the VTK download. If you download the VTK distribution from http://public.kitware.com, there are hundreds of example Tcl scripts for creating different kinds of VTK visualizations. These scripts can be used in developing PV-WAVE wrappers to create these same visualizations from PV-WAVE.

VTK Dataset Creation

VTK supports five (5) basic dataset representations. These represent the different ways in which data can be organized for use in visualizations. This includes representations from simple points in 3D space, polygons, grids, and voxels (volume elements). The following PV-WAVE wrappers offer a way to create, store, and pass these datasets to VTK:

- vtkPOLYDATA
- vtkRECTILINEARGRID
- vtkSTRUCTUREDGRID
- vtkSTRUCTUREDPOINTS

- vtkUNSTRUCTUREDGRID

With all of the above dataset types, the most fundamental element is a point. In VTK there are attributes that can be associated with points and used in various ways for visualizations, such as for coloring points or drawing vectors associated with points. Creating these attributes is accomplished using the PV-WAVE procedure vtkADDATTRIBUTE. Attributes created with this routine can be used in calls to the above five dataset creation routines. See the references in the *Introduction* on page 155 for more details.

Simple Examples

Here are some examples of using the PV-WAVE VTK Integration routines that show how VTK can be accessed in a manner very similar to existing PV-WAVE graphic routines.

Example 1: Create a Surface Plot

```
vtkSURFACE, DIST(10), Shades='slate_blue'
```

This one command automatically invokes vtkINIT and vtkWINDOW to open a window. Since only one color was specified (in this case using a named color), the entire surface is shaded using that color.

NOTE In VTK, *X*, *Y* and Z scaling are always identical, therefore you may need to scale your raw data in order to change the scaling of one direction. For example when using vtkSURFACE multiply your *Z* array by a scale factor so that the height of the surface is appropriate.

Example 2: Display a Cube With a Different Color at Each Vertex

```
vertex_list = [[0.0, 0.0, 0.0], $
               [1.0, 0.0, 0.0], $
               [1.0, 1.0, 0.0], $
               [0.0, 1.0, 0.0], $
               [0.0, 0.0, 1.0], $
               [1.0, 0.0, 1.0], $
               [1.0, 1.0, 1.0], $
               [0.0, 1.0, 1.0]];
polygon_list = [4, 0, 1, 2, 3, $
                4, 4, 5, 6, 7, $
                4, 0, 1, 5, 4, $
                4, 2, 3, 7, 6, $
```

```
                4, 0, 4, 7, 3, $
                4, 1, 2, 6, 5]
TEK_COLOR
vertex_colors = [2,3,4,5,6,7,8,9]
vtkPOLYSHADE, vertex_list, polygon_list, Color=vertex_colors
```

This example shows how we can use a vertex/polygon list to create a plot in much the same way as with POLYSHADE. Since the colors we specify are short integers, the colors used are from the current PV-WAVE color table, which was loaded using `TEK_COLOR` in this case.

Example 3: Adding an Annotation to a Scene

```
vtkWINDOW, /Free, Background='000077'XL, /NoRender
vtkSURFACE, HANNING(20,20)*20.0, Shades=[1.0, 0.0, 0.0, 0.5]
vtkTEXT, 'Transparent Surface', Position=[10, 10, 20], $
         /Follow, Color='green'
vtkRENDERWINDOW
```

NOTE that we use /NoRender to suppress rendering until we have added everything to the scene (the surface and text annotation). Also note that we specify colors in three different ways:

‘000077’XL	A long integer specifying a dark blue color
[1.0, 0.0, 0.0, 0.5]	A four element array specifying the color red with a 50% transparency
'green'	A string specifying a named color, as defined in `vtkcolornames.pro`.

Example 4: Debugging VTK

```
vtkINIT, /Print
vtkWINDOW, /NoRender
vtkSCATTER, RANDOMN(seed, 3, 100), Color=’blue’
vtkRENDERWINDOW
HAK, /Mesg
vtkWDELETE
vtkCLOSE
```

In this example we explicitly call vtkINIT so that we can turn on logging of all Tcl commands. We also manage the window creation and deletion and rendering ourselves.

More examples are provided with the PV-WAVE distribution.

For additional examples see the procedures in the directory `<vni>/vtk-3_2/demo` (where <*vni*> is the path to your top level VNI directory).

Software 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 additional information on the rendering, gridding, and coordinate conversion functions discussed in this chapter, see the *PV-WAVE Reference*.

In PV-WAVE, advanced rendering 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-tracer 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 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.)

4 a volume slicer for interactive subsetting and display of volumetric data

4 a view tool to graphically set the X, Y, Z position

4 “rubber sheet” mapping of an image onto a sphere

Demonstration Programs

There are three sources for advanced rendering demonstration programs:

- You can run the demonstration programs and look at the code in:

(UNIX) `<wavedir>/demo/arl`

(OpenVMS)`<wavedir>:[DEMO.ARL]`

(Windows) `<wavedir>\demo\arl`

Where `<wavedir>` is the main PV-WAVE directory.

- You can run the demonstration programs and look at the code in:

(UNIX) `<wavedir>/demo/arl`

(OpenVMS)`<wavedir>:[DEMO.ARL]`

(Windows) `<wavedir>\demo\arl`

Where `<wavedir>` is the main PV-WAVE directory.

- You can run the PV-WAVE Demonstration Gallery and look at Gallery code examples.

Demonstration Programs in the Examples Directory

You can find most of the Advanced Rendering Library demonstration programs in:

(UNIX) `<wavedir>/demo/arl`

(OpenVMS)`<wavedir>:[DEMO.ARL]`

(Windows) `<wavedir>\demo\arl`

Where `<wavedir>` is the main PV-WAVE directory.

UNIX USERS 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:

q Change to the `examples` directory and start PV-WAVE.

q At the WAVE> prompt, enter the name of the program you want to run. For example:

```
WAVE> vol_demo1
```

q If you want to run another program, 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)

For a quick demonstration of ray tracing applications, CD to the following directory:

(UNIX) <wavedir>/demo/render

(OpenVMS)<wavedir>:[DEMO.RENDER]

(Windows) <wavedir>\demo\render

Where <wavedir> is the main PV-WAVE directory.

Then, enter the following commands:

```
WAVE> show_anim
    ; Quadric animation - Shows earth revolving for several revolutions.
    ; 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.)

```
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 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/Aerodynamics** demos 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/ Aerodynamics** demos of the PV-WAVE Demonstration Gallery.

Tables of Demonstration Programs

The following tables summarize the demonstration programs and list the rendering routines that are used in these programs.

The demonstration programs listed in the tables are located in the `render` and `arl` directories in:

(UNIX) `<wavedir>/demo`

(OpenVMS) `<wavedir>:[DEMO]`

(Windows) `<wavedir>\demo`

Where <wavedir> is the main PV-WAVE directory.

Polygon Rendering

Demonstration Programs	Routines Used
poly_demo1 Displays a perspective view of a surface from a viewpoint within the data.	SET_VIEW3D POLY_SURF POLY_NORM POLY_TRANS POLY_DEV POLY_C_CONV POLY_PLOT
grid_demo4 Shows 4D gridding and a cut-away view of a block of volume data.	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.	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.	SHADE_VOLUME MESH, RENDER
sphere_demo1 Displays an image warped onto a sphere.	POLY_SPHERE CENTER_VIEW POLYSHADE

Polygon Rendering (Continued)

Demonstration Programs	Routines Used
sphere_demo2 Displays data warped onto an irregular sphere.	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.	GRID_SPHERE POLY_SPHERE POLY_COUNT POLY_TRANS POLY_MERGE CENTER_VIEW POLYSHADE POLY_NORM POLY_DEV POLY_PLOT
grid_demo5 Shows spherical gridding.	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.	SPHERE RENDER

Volume Rendering

Demonstration Programs	Routines Used
vec_demo1 Displays a 3D vector field from X-Y-Z data.	VECTOR_FIELD3
vec_demo2 Displays a 3D vector field from the volumetric data with specified starting points for the vectors.	CONV_TO_RECT VECTOR_FIELD3

Volume Rendering (Continued)

Demonstration Programs	Routines Used
vol_demo1 Displays a 3D fluid flow vector field with random starting points for the vectors.	CONV_TO_RECT VECTOR_FIELD3
gen_slic_head show_slic_head Demonstrates the rendering of selected slices through some volume data.	VOLUME RENDER
gen_flat_head show_flat_head Renders a diffuse iso-surface with voxel values.	VOLUME RENDER
gen_tran_head show_tran_head Renders both a diffuse iso-surface together with a partially transparent iso-surface.	VOLUME RENDER
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.	VOLUME RENDER

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.	VOL_PAD CENTER_VIEW VOL_MARKER SHADE_VOLUME POLYSHADE VOL_TRANS VOL_REND

Polygon and Volume Rendering (Continued)

Demonstration Programs	Routines Used
vol_demo3 Displays 3D fluid data using two display techniques.	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.	GRID_4D VOL_PAD CENTER_VIEW SHADE_VOLUME POLYSHADE POLY_NORM POLY_TRANS POLY_DEV POLY_COUNT POLY_PLOT VOL_TRANS VOL_REND

Gridding

Demonstration Program	Routines Used
f_gridemo2 Shows 2D gridding with dense data input.	FAST_GRID2
f_gridemo3 Shows 3D gridding with dense data input.	FAST_GRID3
f_gridemo4 Shows 4D gridding with dense data input.	FAST_GRID4
grid_demo2 Shows 2D gridding with sparse data input.	GRID_2D
grid_demo3 Shows 3D gridding with sparse data input.	GRID_3D
grid_demo4 Shows 4D gridding with sparse data input.	GRID_4D

Gridding (Continued)

Demonstration Program	Routines Used
grid_demo5	GRID_SPHERE
Shows spherical gridding.	

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, you may need 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 174 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 178 for details.
- Set up your data for viewing. See the section *Setting Up Data for Viewing* on page 181 for details.
- Use one of the rendering routines to render the image. The rendering routines are:

  POLY_PLOT

  POLYSHADE

  VECTOR_FIELD3

  VOL_MARKER

  VOL_REND

  RENDER

  See the sections *Rendering with Standard Techniques* on page 181 and *Ray-tracing* on page 182.
- Display data. See the section *Displaying Rendered Images* on page 202.

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 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 183, 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 7-1*, there are only six vertices in the resulting *vertex_list* because two vertices are shared by both polygons.

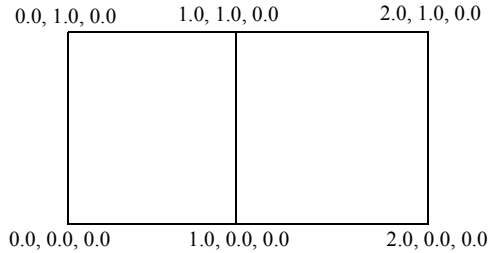


Figure 7-1 Vertices of two square polygons. Six vertices define both polygons.

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 189, *Example 2: Polygonal Mesh (Flat-shaded Polygons)* on page 189, and *Example 3: Polygonal Mesh (Many Polygons)* on page 190.

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 7-2*. This function is often used to process volumes before using the VOL_TRANS and SLICE_VOL function.

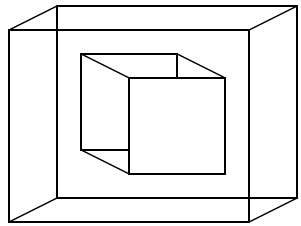


Figure 7-2 The VOL_PAD function pads the space between two volumes with zeros. Without VOL_PAD, data values from the outer edges of the small volume fill the empty space between the two volumes.

- **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 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 This procedure is used by `sphere_demo3`, `vec_demo1`, `vec_demo2`, and `vol_demo1`. These demonstration programs are located in:

  (UNIX) `<wavedir>/demo/arl`

  (OpenVMS)`<wavedir>:[DEMO.ARL]`

  (Windows) `<wavedir>\demo\arl`

  This procedure is also used by all of the `gen_` demonstration programs in:

  (UNIX) `<wavedir>/demo/render`

  (OpenVMS)`<wavedir>:[DEMO.RENDER]`

  (Windows) `<wavedir>\demo\render`

  Where `<wavedir>` is the main PV-WAVE 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

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 a standard list of vertices and polygons that are described in *Vertex Lists and Polygon Lists* on page 175.

  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 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 a

$$Color\ (D + T + A)$$

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*.

• 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 191.

Setting Object and View Transformations

The view that is automatically generated by RENDER is depicted in *Figure 7-3*. (You can retrieve this view with the *Info* keyword; for details, see the *PV-WAVE Reference*.)

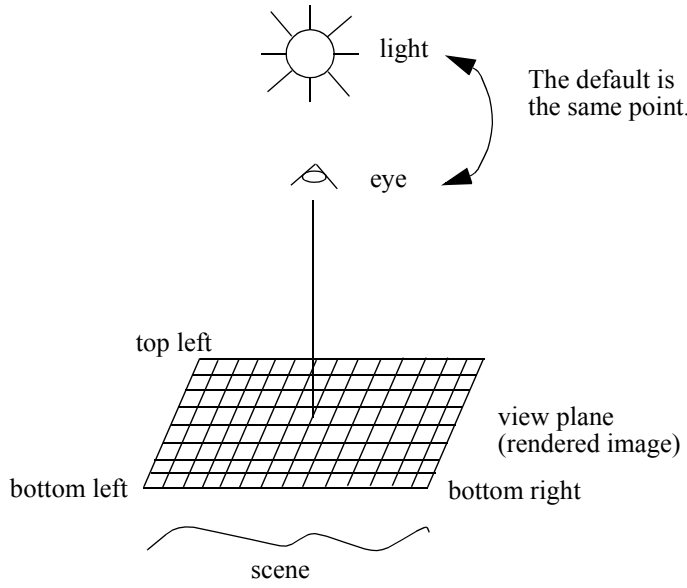


Figure 7-3 The default view used in RENDER positions the observer's eye on the positive *z*-axis, looking towards the origin into the scene with a slight perspective. All objects are visible in this default view.

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:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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* on page 188.

For more information, see the section *Geometric Transformations* on page 148.

Invoking RENDER

RENDER is the function that generates the image from the objects you have specified. The general format is:

$$result = \text{RENDER}(object_1, ..., object_n)$$

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 7-3* on page 187, 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 in:

(UNIX) `<wavedir>/demo/render`

(OpenVMS) `<wavedir>:[DEMO.RENDER]`

(Windows) `<wavedir>\demo\render`

The data and image files used are in:

(UNIX) `<wavedir>/data`

(OpenVMS)<wavedir>:[DATA]

(Windows) <wavedir>\data

Where <wavedir> is the main PV-WAVE directory.

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 7-4*, 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.

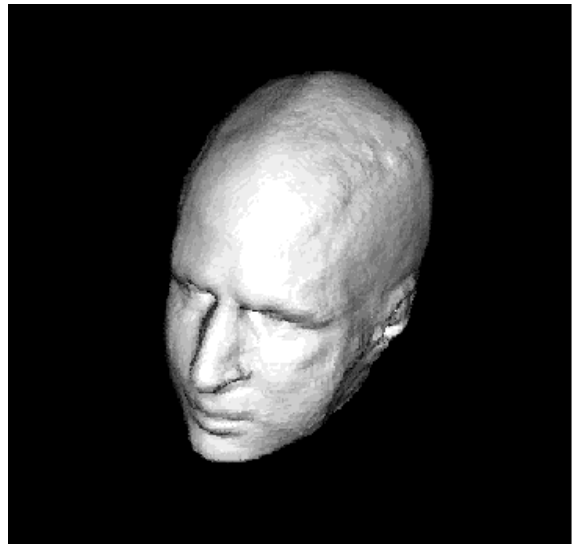


Figure 7-4 An example of polygonal meshes showing 52,000 polygons generated as an iso-surface and ray traced using the RENDER function.

Program Listing

```
PRO gen_iso_head
   volx = 115 & voly = 75 & volz = 105
   ; Volume data dimensions
   band = 5
   ; The neighborhood size of the 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.

   SHADE_VOLUME, head, 18, vertex_list, polygon_list, /Low
   m = MESH(vertex_list, polygon_list)
   ; Generate iso-surface.

   T3D, /Reset, Rotate = [60.0,0.0,-60.0]
   im = RENDER(m, x = 512, y = 512, Transform = !P.T)
   TVSCL, 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
   TVSCL, 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

   decal = BYTARR(720, 360)
   OPENR, 1, !Data_Dir + 'world_map.dat'
   READU, 1, decal
   CLOSE, 1
   ; Load the decal to apply.

   dif = FLTARR(256)
   amb = FINDGEN(256)/255.
   ; Set shading to correspond directly to image values.
   T3D, /Reset, Rotate = [-90.0, 90.0, 0.0]
   c = SPHERE(Decal = decal, Kamb = amb, Kdiff = dif, $
        Transform = !P.T)

   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)
   ; Create an animation by orbiting view around the sphere.
   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

   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)))
    ; Create a color lookup table.
    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

    frames = BYTARR(256, 256, 72)
```

```
    OPENR, 1, !Data_Dir + 'world_anim.img'
    READU, 1, frames
    CLOSE, 1
    ; Load the previously generated animation.

    MOVIE, frames, Order=0
    RETURN, frames
    ; Display the animation.
END
```

Example 5: Slicing a Volume

This example renders selected slices from a large amount of volume data. The resulting image, shown in *Figure 7-5*, is saved to a file and displayed using `show_slic_head`.

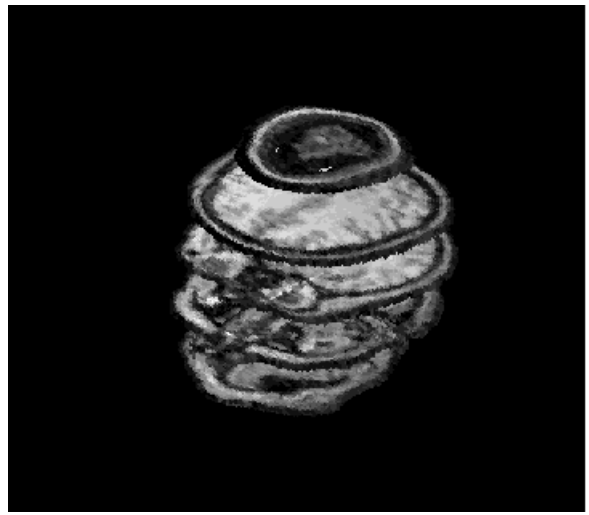


Figure 7-5 After slices have been rendered from a large quantity of volume data for this example, the resulting pixel intensity values show the actual density values of the voxel data.

Program Listing

```
PRO gen_slic_head
    width = 125 & height = 85 & depth = 115
    load_seg_head, head, skull
    ; 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.
    vox = BYTARR(width, height, depth)
    FOR i=0,depth-2,20 DO BEGIN
vox(*,*,i) = head(*,*,i)
vox(*,*,i+1) = head(*,*,i+1)
    ; Generate the slices of segmented data we wish to view.
    ENDFOR
    v = VOLUME(vox)

    T3D, /Reset, Rotate = [60.0,0.0,-45.0]
    im = RENDER(v, x = 512, y = 512, Transform = !P.T, /Scale)
    TVSCL, 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
    TVSCL, 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 7-6*, are saved to a file and displayed using `show_flat_head`.

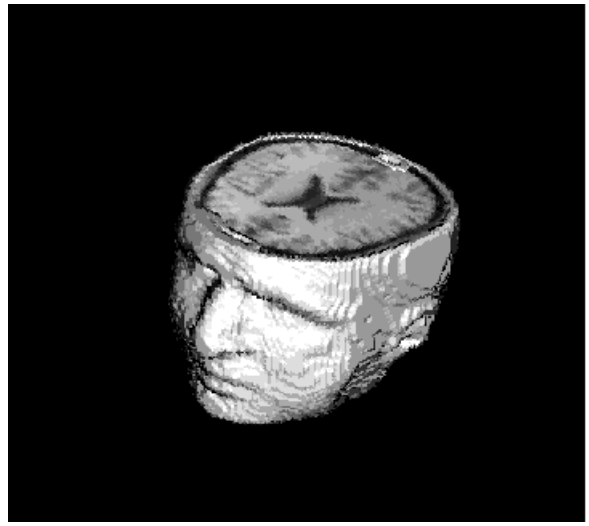


Figure 7-6 This example renders a diffuse iso-surface using actual voxel values. The surface of the head is shaded using diffusion, and the intensity values on top correspond directly to the voxel density values.

Program Listing

```
PRO gen_flat_head
   width = 125 & height = 85 & depth = 115
   load_seg_head, head, skull
   ; 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.
   overlap = skull * head
   overlap(where(overlap GT 0)) = 1
   head = head * (BYTE(1) - overlap)
   ; Remove portion of head that overlaps with skull.
   vox = BYTARR(width, height, depth)
   FOR i=0,76 DO vox(*,*,i) = $
        head(*, *, i) + (skull(*, *, i)*BYTE(255))
   ; Generate the slices of smoothed data we wish to view.
   diff = FLTARR(256) & diff(255) = 0.6
   amb = FINDGEN(256)/255.0 & amb(255) = 0.0
   ; Voxel value 255 is special, representing the skull surface.

   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)
   TVSCL, 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
   TVSCL, 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 7-7*, are saved to a file and displayed using `show_tran_head`.

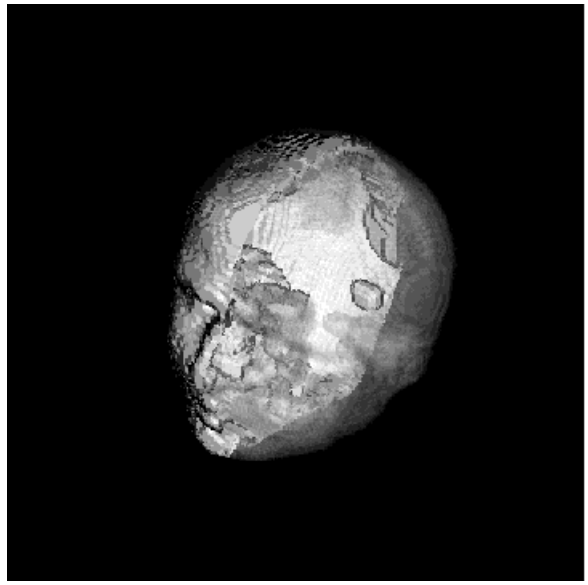


Figure 7-7 The voxel values within the iso-surfaces are completely transparent in this example, which renders a diffuse iso-surface and a partially transparent iso-surface.

Program Listing

```
PRO gen_tran_head
   width = 125 & height = 85 & depth = 115
   load_seg_head, head, skull
   ; 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 load_seg_head.pro (in the wave/demo/render directory).
   mask = BYTARR(width, height, depth)
   mask(*, height/2:*, *) = 1
   ; Generate a mask plane that will split head in half,
   ; allowing half to be diffuse and rest to be transparent.


   shell = skull * mask * BYTE(255) + $
        skull * (BYTE(1) - mask) * BYTE(254)
   ; Half surface = 255, other half = 254
   overlap = skull * head
```

```
    overlap(WHERE(overlap GT 0)) = 1
    head = head * (BYTE(1) - overlap)
    ; Remove portion of head that overlaps with skull.
    vox = shell + head
    diff = FLTARR(256) & diff(255) = 1.0 & diff(254) = 0.05
    ; 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).
    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)
    TVSCL, 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
    TVSCL, 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 7-8*, are saved to a file and displayed using `show_core_head`.

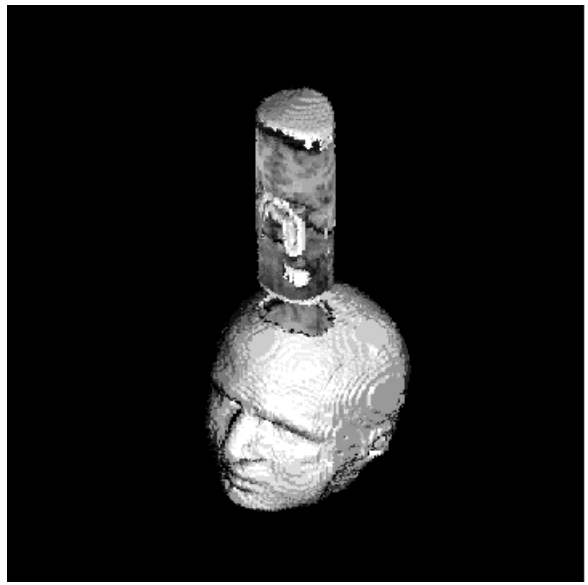


Figure 7-8 Two separate volumes are rendered simultaneously in this example, each using a different transformation matrix.

Program Listing

```
PRO gen_core_head
    width = 125 & height = 85 & depth = 115
    load_seg_head, head, skull
    ; 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.
    overlap = skull * head
    overlap(WHERE(overlap GT 0)) = 1
    head = head * (BYTE(1) - overlap)
    ; Remove portion of head that overlaps with skull.
    vox = head + (skull * BYTE(255))
    circle = BYTARR(width, height)
    radius2 = 16 * 16
    ; Create a circle (used for CYLINDER) mask plane.
    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

   core = BYTARR(width, height, depth)
   ; Mask out the core sample and "subtract" out from slices.
   FOR z=0, depth-1 DO BEGIN
core(*, *, z) = vox(*, *, z) * circle
vox(*, *, z) = vox(*, *, z) - core(*, *, z)
   ENDFOR

   diff = FLTARR(256) & diff(255) = 0.6
   amb = FINDGEN(256)/255.0 & amb(255) = 0.0
   ; Voxel value 255 is special, representing the skull surface.
   v0 = VOLUME(vox, Kdiff = diff, Kamb = amb)
   ; Surface and interior of skull.
   T3D, /Reset, Translate=[0.0, 0.0, 1.0]
   v1 = VOLUME(core, Transform = !P.T, Kdiff = diff, Kamb = amb)
   ; Core sample.
   T3D, /Reset, Rotate = [60.0, 0.0, -45.0]
   im = RENDER(v0, v1, x = 512, y = 512, Transform = !P.T, /Scale)
   TVSCL, 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
   TVSCL, 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 188 for examples of using TV and TVSCL with the RENDER function.

8

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, the 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 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 8-1* illustrates a plot with two levels of date/time labeling:

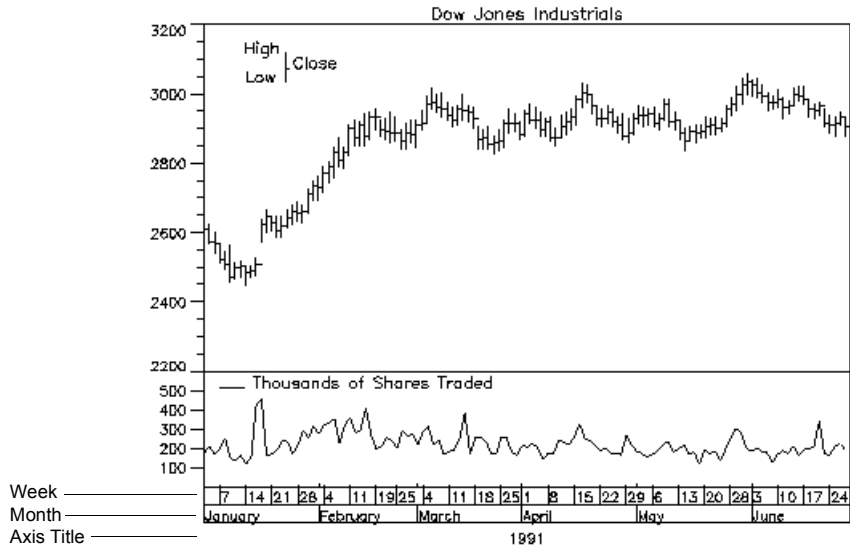


Figure 8-1 Date/time Plot

The 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:

q Read data into PV-WAVE.

q Convert data representing dates and/or times to date/time data.

q Manipulate the date/time data (optional).

q 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 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 data types, which must then be converted into 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 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 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 209 for details.

TIP In some instances, your input file may not contain explicit date/time information. You can generate a scalar date/time variable with one of the conversion functions and then use the DTGEN function to create a date/time variable containing an array of date/time structures. See *Generating Date/Time Data* on page 213 for details. Also see Examples 1, 2, and 3 in *Creating Plots with Date/Time Data* on page 219.

Manipulating the Date/Time Data

After you have created 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 date/time variable. You may also want to eliminate holidays and weekends from your data with the CREATE_HOLIDAYS and CREATE_WEEKENDS procedures. For details, see *Manipulating Date/Time Data* on page 214.

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 date/time axis, PV-WAVE provides several keywords. For details, see *Creating Plots with Date/Time Data* on page 219.

The Date/Time Structure

Date/Time data is stored in a structure (!DT) containing the fields shown in the following table.

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 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 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 date/time variable is used with any of the date/time functions.

NOTE Rather than modifying a 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 Date/Time Variables

Normally, you create date/time variables using the conversion functions or the DC_READ functions. However, you can also create an "empty" date/time variable by assigning !DT to a variable name. Here are a couple of examples:

```
date = {!DT}
    ; Creates a 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 Date/Time Data* on page 219 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 date/time data. There are three methods for generating variables containing date/time data:

- You can read data directly into date/time variables using the DC_READ functions in conjunction with the *DT_Template* keyword. See for an example.

  Refer to the section *Transferring Date/Time Data* in Chapter 8 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 date/time variables. These conversion routines are discussed in the next section.

  Refer to the section *Transferring Date/Time Data* in Chapter 8 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 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 Date/Time Data* on page 213.

Converting Your Data into Date/Time Data

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

- **STR_TO_DT** — Converts string data or variables containing string data into date/time variables.
- **VAR_TO_DT** — Converts numeric variables containing date/time information into date/time variables.
- **SEC_TO_DT** — Converts seconds into date/time variables.
- **JUL_TO_DT** — Converts the Julian day into a 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 date/time variables. The function has the form:

result = STR_TO_DT(*date_string*s [, *time_string*s])

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 the following table.

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 (,).

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 variables, you can use the VAR_TO_DT function to convert these variables into 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 date/time data and verify that a 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 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 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 date/time variable. For more information on how to use this function with the table functions, refer to the examples in the section *Using Date/Time Data in Tables* on page 253.

This function has the form:

result = JUL_TO_DT(*julian_date*)

Example

```
dt = JUL_TO_DT(87507)
    ; Converts the Julian day 87507 to a date/time variable.
PRINT, dt
{ 1992 4 15 0 0 0.00000 87507.000 0}
```

Generating Date/Time Data

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

- Create an initial 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 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 date/time data for all of the hours of the month:

```
date1 = VAR_TO_DT(92,4,1,1)
    ; Use VAR_TO_DT to create the initial date/time variable for
    ; April 1, 1992, 1:00 a.m.

PRINT, date1
{ 1992 4 1 1 0 0.00000 87493.042 0}

dtarray = DTGEN(date1, 720, /hour)
    ; Generate a date/time array variable that contains a 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 date/time variable that contains all of the weekdays for the month of January.

```
date1 = VAR_TO_DT(1992,1,1)
    ; Creates an initial date/time variable to use with the DTGEN function.

CREATE_WEEKENDS, ['sat', 'sun']
    ; Defines the weekend days.

dates = DTGEN(date1, 23, /Compress)
    ; Generates a 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 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 date/time variables. This section briefly describes each of these functions. For more information about these functions, see the *PV-WAVE Reference*.

Adding to a Date/Time Variable

You may wish to add any number of date/time units to one or more existing 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 date/time variable to produce a new variable.

```
dtvar = VAR_TO_DT(1992, 12, 31, 15)
    ; Create a date/time variable.
dtvar1= DT_ADD(dtvar, Hour = 30)
    ; Create a new date/time variable by adding thirty hours to dtvar,
    ; an existing 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 date/time variable that contains all the days of the month of May excluding weekends.

```
dates = REPLICATE({!DT}, 21)
    ; Creates a 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 date/time variable to use with DT_ADD.
FOR I = 1,20 DO dates(I)=DT_ADD(dates(I-1), $
```

```
/day, /Compress)
; Generates Date/TIme structures for the remaining days of the
; month. The Compress keyword excludes the weekend days.
```

Subtracting from a Date/Time Variable

The function DT_SUBTRACT subtracts a value from a 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 date/time variable.
dtvar1= DT_SUBTRACT(dtvar, Hour = 30)
    ; Create a new 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 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 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 Date/Time Variables

You can exclude holidays and weekend days from 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 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 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 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 conversion functions. You need to run this procedure after restoring any 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 a 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 date/time values. The function returns an array of floating point values containing the compressed Julian days—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 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 date/time axes.

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 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 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.
```

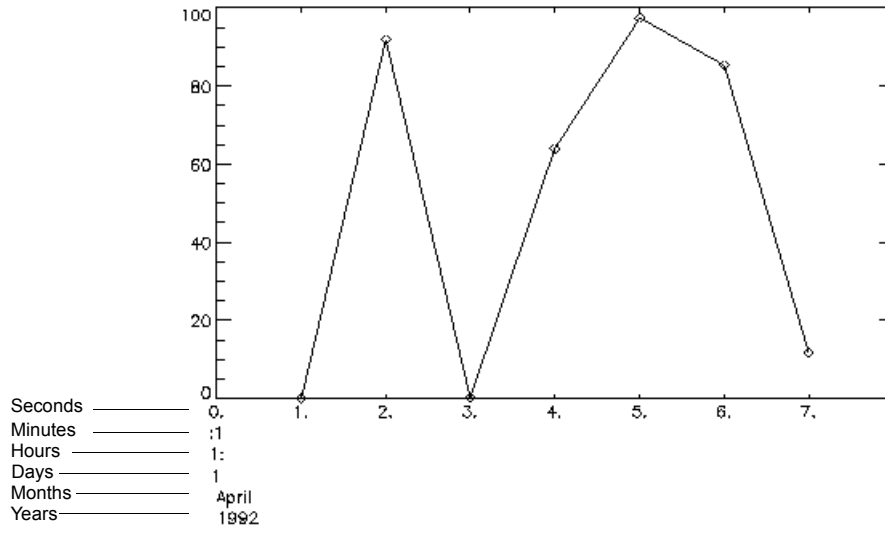


Figure 8-2 A date/time plot showing the first seven seconds of data for April 1, 1992. The keyword *Psym* value of -4 connects the data points with solid lines.

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 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 date/time structures.

x = DTGEN(fday, num, /Minute)
    ; Generates a 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.
```

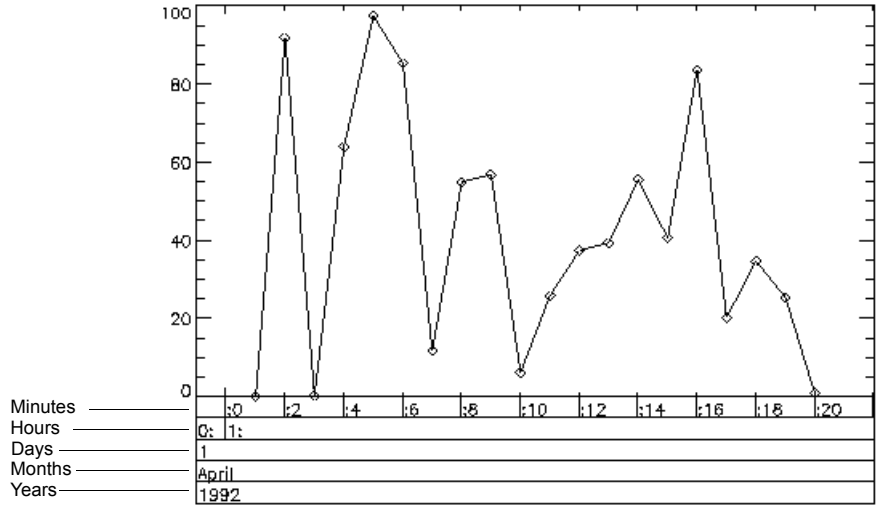


Figure 8-3 A date/time plot for the first twenty minutes of April 1. The *Box* keyword draws the boxes around the date/time labels.

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 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
```

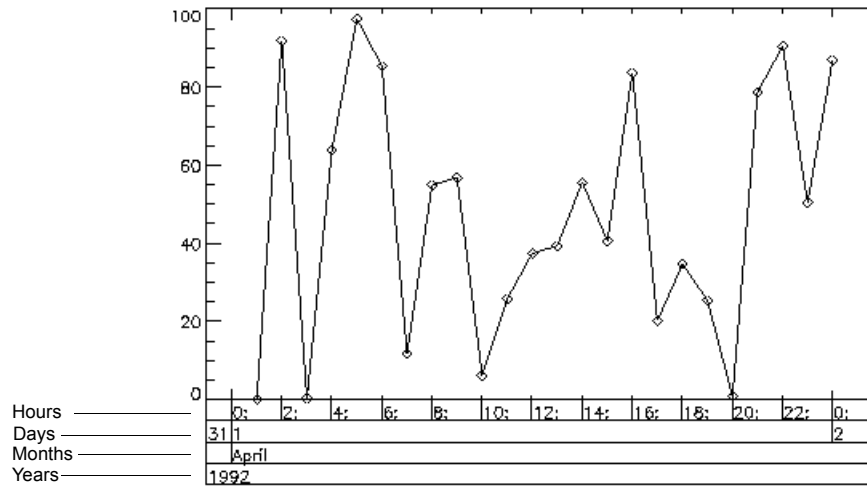


Figure 8-4 A date/time example with the data at each hour for April 1 plotted.

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 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 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 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 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.
```

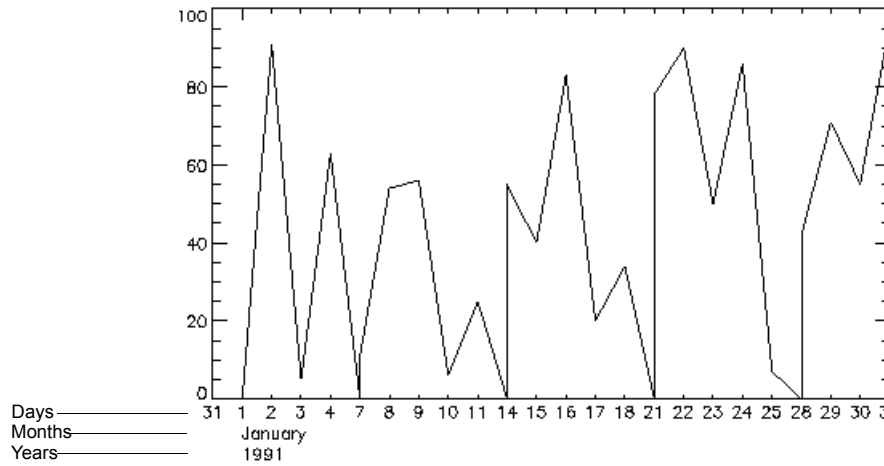


Figure 8-5 A date/time plot illustrating daily product sales for January. The *DT_Range* keyword restricts the days to January only. The *Compress* keyword eliminates weekend days (January 5, 6, 12, 13, 19, 20, 26, and 27).

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 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.
```

NOTE By default, week boundaries are plotted as Mondays. To label a different day of the week, use the Week_Boundary keyword. To specify an exact axis range, use the Dt_Range keyword. Looking at the above example, to have the tick labeling occur on the day of the week of the first data point, and to plot only the first 12 weeks of data use the following:

```
PLOT,dates,amount,Start_level=4,$
      Psym=-4,Dt_Range=[dates(0).Julian,dates(12).Julian],$
      Week_boundary=DAY_OF_WEEK(dates(0))
```

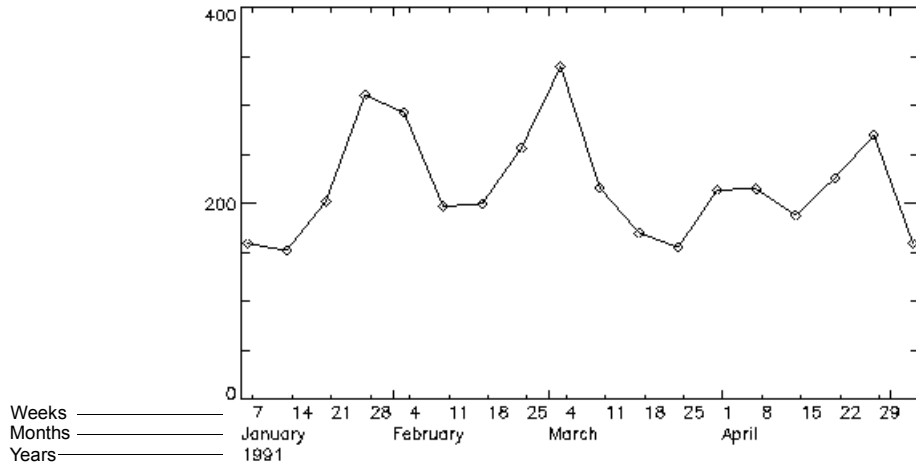


Figure 8-6 Date/time plot of product sales for each week from January 7 to May 6. The *Start_Level* keyword value of 4 ensures that the weekly amounts are plotted on the first level.

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 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 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.
```

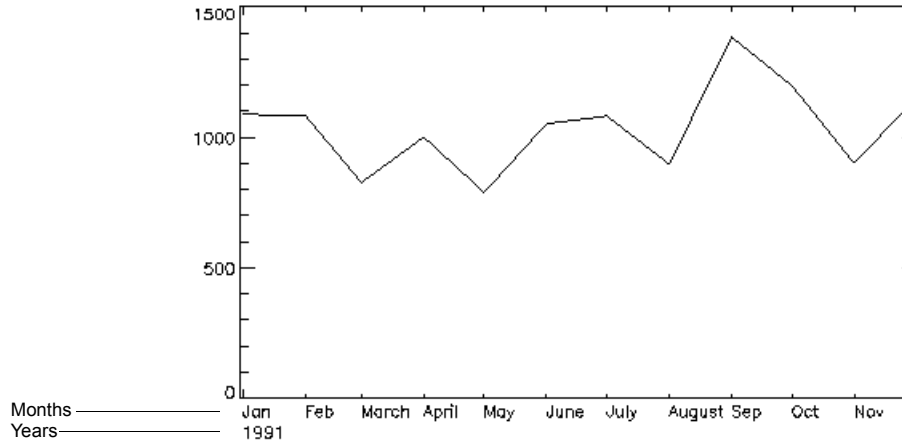


Figure 8-7 The monthly sales for 1991. The *Month_Abbr* keyword allows all month names to be written on the date/time axis. Without this keyword, no labels would be shown for February or September.

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 8-8*). 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 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.
```

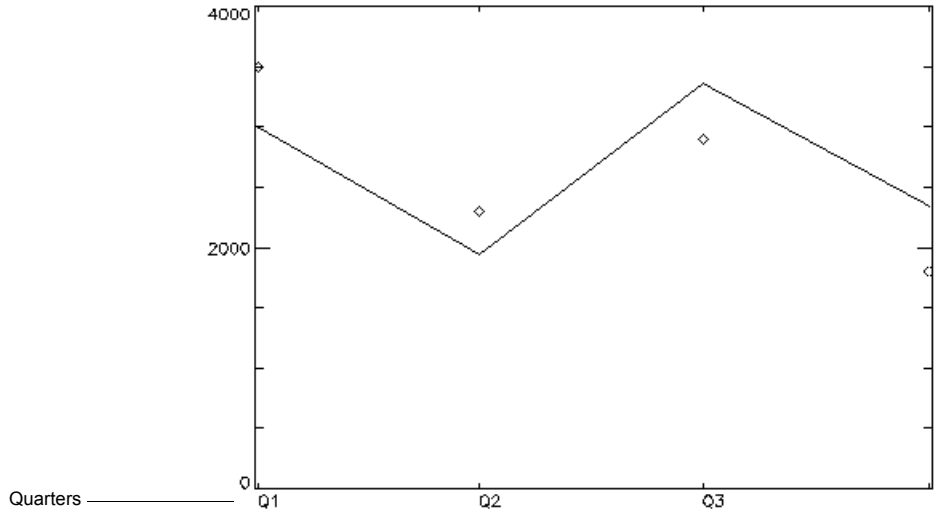


Figure 8-8 The quarterly sales for 1991. The plot also shows the projected sales (individual diamonds) for each quarter plotted with the OPLOT procedure.

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 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.
```

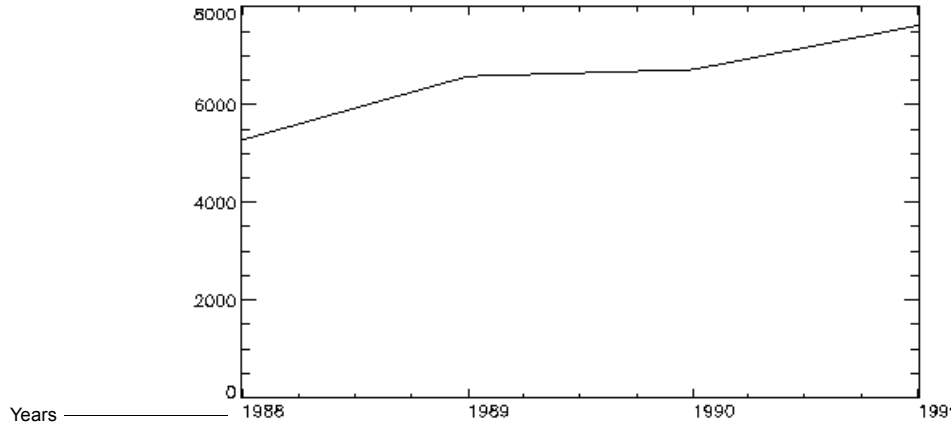


Figure 8-9 The yearly sales for the last four years. This plot can be generated using the JUL_TO_DT function (Example 8) or the *XType* keyword (Example 9).

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 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 8-9.
```

See for more information about the *XType* keyword. Also see the description of the DT_COMPRESS function in the *PV-WAVE Reference*.

Writing Date/Time Data to a File

There are two methods for writing date/time data to a file. You can use DC_WRITE functions to both convert and write data or you can first convert the 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 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*.

NOTE 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 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 date/time variables to strings. The procedure has the form:

DT_TO_STR, *dt_var*

Example

Assume you have a 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 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 209 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 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 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 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 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 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 Date/Time Utility Functions

PV-WAVE contains several utilities for generating and obtaining information about 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 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 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 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 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 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 string or array of strings containing the month name of *dt_var*, where *dt_var* is a 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 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 an 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 date/time variable.
daynumber = DAY_OF_YEAR(dttoday)
PRINT, daynumber
   106
```

The DT_PRINT Procedure

This procedure takes the values in a 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
```

9

CHAPTER

Creating and Querying Tables

A table is a natural and easily understood way of organizing data into columns and rows. Many computer applications 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 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 table functions include:

- **BUILD_TABLE** — Creates a new table from 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).
- **GROUP_BY** — Performs summary (aggregate) functions to groups of rows in a PV-WAVE table variable.
- **ORDER_BY** — Sorts the rows in a PV-WAVE table variable to create a new table.

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

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
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 The second parameter in a QUERY_TABLE call is one string. The plus sign (+) used above is the 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 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 253.

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, PV-WAVE Programmer's Guide, before you proceed to learn about the table functions. Also see the section *Tables and Structures* on page 258.

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 as follows.

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
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 256.

Using INFO to View the Table Structure

You can use the INFO command to view the table structure. Tables are represented as arrays of structures (for more information on this, see *Tables and Structures* on page 258). 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
```

DATE	EXT	DUR	COST
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, !Data_Dir+'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 enclosed 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 256. For information on plotting table data, see *Plotting Table Data* on page 257.

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, data-structure notation must be used to specify the column (for more information, see *Tables and Structures* on page 258). 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')
```

NOTE The GROUP_BY function performs the same basic operation as the Group By clause of QUERY_TABLE, but with a more compact syntax. For detailed information on GROUP_BY, see the *PV-WAVE Reference*.

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, shown in the following table, operate on the lowest-level grouping to produce the desired result.

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)

Calculation Functions (Continued)

Function	Description	Phone_Data Applications
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

```
EXT             SUM_COST         SUM_DUR
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')
```

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.

NOTE The ORDER_BY function performs the same basic operation as the Order By clause of QUERY_TABLE, but with a more compact syntax. For detailed information on ORDER_BY, see the *PV-WAVE Reference*.

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.

You can use 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')
```

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

NOTE 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 251, 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 in the previous section 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.

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 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 208. 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 257.

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 258.

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 the *PV-WAVE Reference*. You can also refer to the description of the PRINT function 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
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 258.

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'
```

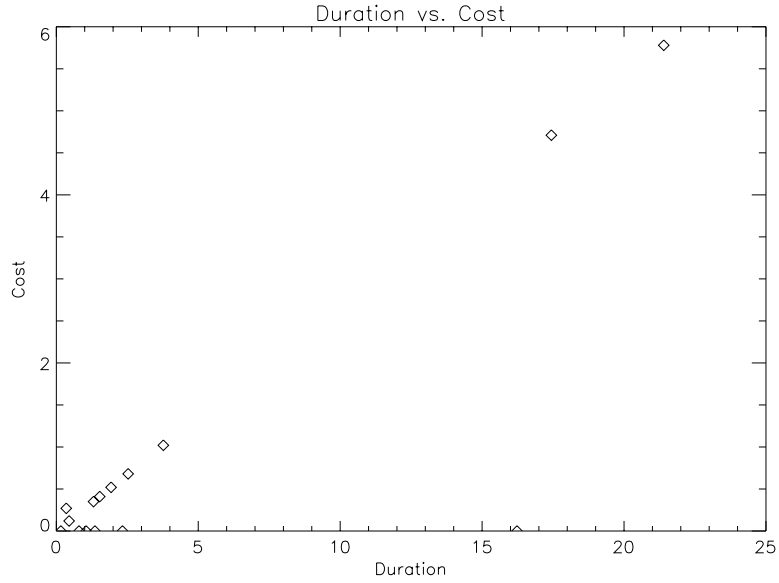


Figure 9-1 Plot of data from a table.

Tables and Structures

As noted previously, a table is represented 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 in the *PV-WAVE Programmer's Guide*.

The basic syntax of 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 241, 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
```

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*.

10

CHAPTER

Using 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. PV-WAVE simply sends the characters to the graphics device, which displays them using these built-in fonts.

This chapter discusses how to work with both software and hardware fonts.

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 some window systems) to publication quality (for example, PostScript and Windows True Type fonts).

3D Transformations

Software and some hardware characters go through the same 3D transformations as the rest of the plot, yielding a better looking plot. Hardware font drivers that

support 3D transformations include X Windows, WIN32 (on Windows NT platforms only), PostScript, and WMF.

Text Rotation

Both hardware and software fonts can be rotated. For example, you can use the *Orientation* keyword with the XYOUTS procedure to rotate a text string.

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.

UNIX and OpenVMS USERS You may notice that under X Windows the size of the software fonts varies from device to device. When you start PV-WAVE, the 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 (software fonts), 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.

Localized Fonts

"Local" fonts refer to fonts that contain characters that are required for a specific language. For example, a font designed for French text contains characters that are not found in an English text font.

The PV-WAVE software fonts contain a limited set of localized characters. If you require characters outside this set (French characters, for example), then you must use a suitable hardware font.

For information on adding a local font to the set of available hardware fonts, see *String Resource File for Font Mappings* on page 268.

Using Software Fonts

To use software fonts, you must set the value of the !P.Font system variable to –1. This is the default setting. For example:

```
WAVE> !P.Font = -1
```

This section explains how to format software text and select different software fonts.

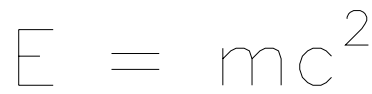
You can embed formatting and font commands in the string arguments of plotting keywords such as *Title*, *Subtitle*, *XTitle*, and *YTitle* and in the *string* parameter of the XYOUTS procedure.

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.

The following table 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.

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 above.)

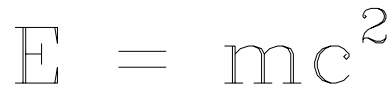
`!D`	Shift down to the first level subscript and decrease the character size by a factor of 0.62.
`!E`	Shift up to the exponent level and decrease the character size by a factor of 0.44.
`!I`	Shift down to the index level and decrease the character size by a factor of 0.44.
`!L`	Shift down to the second level subscript. Decrease the character size by a factor of 0.62.
`!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. 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 , in the *PV-WAVE Reference*.

More examples showing font selection appear later in this chapter. The following table lists the font selection commands.

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

Font Selection Commands (Continued)

Font Command	Description
!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
!16	Cyrillic
!17	Triplex Roman
!18	Triplex Italic
!20	Miscellaneous

Using Hardware Fonts

To use hardware fonts, you must set the value of the !P.Font system variable to 0 (zero). For example:

```
WAVE> !P.Font=0
```

NOTE By default, software fonts are enabled (!P.Font is set to –1).

TIP If you want to use hardware fonts by default, add the statement !P.Font=0 to the PV-WAVE startup file.

Hardware Font Formatting Commands

In general, you can use hardware fonts in the same way you use software fonts. The text formatting commands (for example, !U) and font commands (for example, !3) described and listed in the previous section can all be used with hardware fonts.

NOTE A string resource file is used to map software font commands to specific hardware fonts. A resource file with default settings is provided; however, you can easily change these defaults. This resource file is described in detail in *String Resource File for Font Mappings* on page 268.

Using PostScript Formatting Commands

NOTE The default PostScript fonts changed with PV-WAVE 6.21. The previous default PostScript font was 12 point Helvetica. The new default PostScript font is 14 point Times Roman. You can change the default font by editing the file `fontmap_ps`, which is discussed in *String Resource File for Font Mappings* on page 268.

PV-WAVE provides a set of DEVICE procedure keywords that can be used to set the default font for the PostScript driver. See the appendix called *Output Devices and Window Systems* in the PV-WAVE Reference for a list of these DEVICE keywords.

For example, to set the default PostScript font to boldface Helvetica, use the commands:

```
SET_PLOT, 'ps'
DEVICE, /helvetica, /bold
```

Additional Text Formatting Commands

The following formatting commands for hardware fonts can be used with the WIN32, WMF, X, and PostScript drivers:

Formatting Command	Description
!FB	Switch to the bold face of the current font.
!FI	Switch to the italic face of the current font.
!FU	Underline the current font.
!FN	Switch to the normal form of the current font.
!P*xx*	Switch to point size *xx* of the current font, where *xx* is a two digit integer (01-99).

String Resource File for Font Mappings

PV-WAVE provides a string resource file that lets you map PV-WAVE software font commands to device-specific hardware fonts. This section explains the basic format and location of this file.

Format of the Fontmap String Resource File

The string resource file used for font mapping consists of a tab or space separated list of font numbers and target fonts. For example, the first five lines of the default fontmap file for the WIN32 driver are as follows:

3	Times New Roman, 14
4	Symbol, 14
5	Times New Roman, 14, Bold
6	Times New Roman, 14, Bold, Italic
7	Symbol, 14, Bold, Italic

For example, if the output device is set to WIN32 and hardware fonts are selected (!P.Font = – 1), then you can use the !5 command in a text string to produce Times New Roman, 14 point, boldface text in your plot:

```
XYOUTS, x, y, '!5Carbon Dioxide Data'
```

Location of the Fontmap String Resource File

By default, the fontmap string resource files for each device are in:

(UNIX) `<wavedir>/xres/!Lang/kernel`

(OpenVMS) `<wavedir>:[XRES.!Lang.KERNEL]`

(Windows) `<wavedir>\xres\!Lang\kernel`

Where `<wavedir>` is the main PV-WAVE directory.

Files are named according to the convention: `fontmap_`*device*. For example, the fontmap file for the WIN32 device is:

```
fontmap_win32
```

TIP If you change the `fontmap_x` file to add a localized font, we recommend that you make a similar change to the `fontmap_ps` file. The reason for this is that VDA Tools assume that the X Window and PostScript font number load equivalent fonts.

Using the WAVE_FONTMAP_PATH Environment Variable

You can also use the environment variable `WAVE_FONTMAP_PATH` to specify a path of directories to search for the font map file.

Font map files are named by the following convention:

```
GETENV('WAVE_FONTMAP_FILEBASE') + '_device'
```

where *device* is one of the following supported devices:

Device	Description
WIN32	WIN32 Driver
WMF	Windows Metafile Driver
X	X Windows Driver
PS	PostScript Driver

The default value of the variable `WAVE_FONTMAP_FILEBASE` is `fontmap`. Thus, the default font mapping file for the WIN32 driver is:

```
<wavedir>\xres\!Lang\kernel\fontmap_win32
```

TIP The value of `WAVE_FONTMAP_FILEBASE` can be the name of a hidden file.

Text Formatting Examples

The following sections demonstrate how to format text strings.

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 10-1*. 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 10-1 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 10-2*:

```
A = '!6!L!!L!S!E!!Exponent!R!I!!Index' + $
     '!N!!N!S!I!!I!R!E!!E!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
```

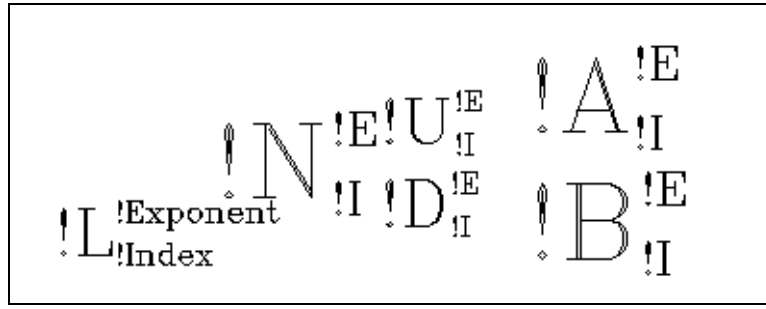


Figure 10-2 Formatted text

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 10-3* was produced.

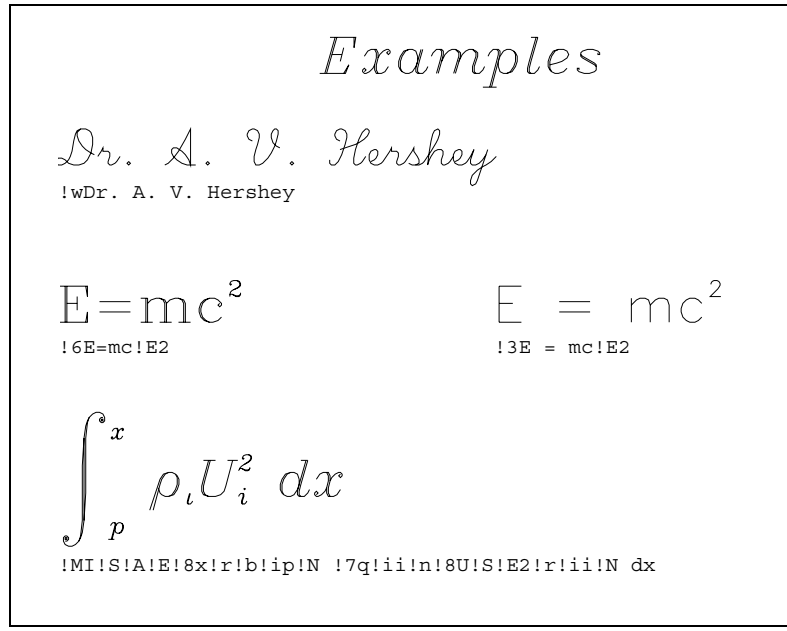


Figure 10-3 Changing text font and formatting mathematical expressions

Detailed Discussion

The bottom integral term shown in *Figure 10-3* was formed by the 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
```

The formatting commands used to produce *Figure 10-3* are summarized in the following table.

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”.

Formatting Commands Used in Example 3 (Continued)

Format Command	Description
`!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 10-4*.

```
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.
```

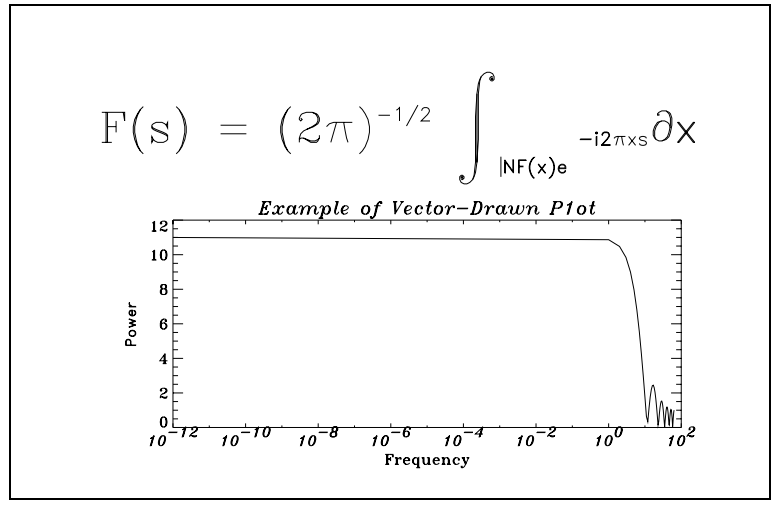


Figure 10-4 Example of a plot drawn with software text

11

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 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 include:

- **RGB** — Red, Green, and Blue (the default)
- **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 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 11-1*. 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 11-1*, 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.

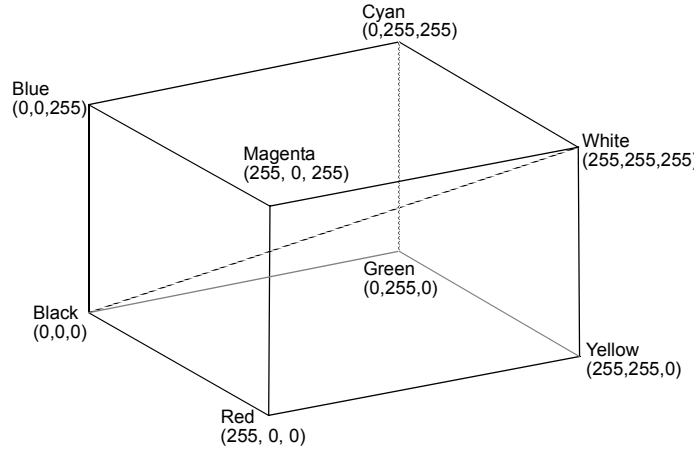


Figure 11-1 RGB color cube. Primary and secondary colors are located at the corners of the cube; grays are along the main diagonal. (After Foley and Van Dam).

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.Eight bits per pixel permits the simultaneous display of $2^8 = 256$ colors. selected from the much larger set of 2^{24} colors.

Windows USERS You can run PV-WAVE on a 24-bit display; however, PV-WAVE only uses 256 colors.

For a more thorough comparison of 8-bit and 24-bit displays, refer to the *PV-WAVE Reference*.

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:

4 loading predefined and custom color tables

4 modifying color tables to create special effects

4 plotting colors used for the elements of "simple" plots

UNIX and OpenVMS USERS To use color, 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`. This file is in:

  (UNIX) `<wavedir>/bin`

  (OpenVMS)`<wavedir>:[BIN]`

  (Windows) `<wavedir>\bin`

  Where `<wavedir>` is the main PV-WAVE directory.

- **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.

The 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 plot colors, see *Controlling Plot Colors* on page 288.

For color and gray-scale devices, the default is to display 8-bit graphics using the color table B–W Linear (standard color table number 0).

UNIX and OpenVMS USERS On a monochrome display, by default, color graphics are dithered. For more information about dithering, see *Displaying Images on 24-bit Devices (UNIX/OpenVMS)* on page 132.

Number of Colors in the Color Table Under UNIX/OpenVMS

Under UNIX and OpenVMS, 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 the *PV-WAVE Reference*.

Loading a Predefined Color Table: LOADCT

Use LOADCT to load one of the predefined color tables. There are 16 color tables, ranging from 0 to 15, in the file `colors.tbl`. This file is in:

(UNIX) `<wavedir>/bin`

(OpenVMS) `<wavedir>:[BIN]`

(Windows) `<wavedir>\bin`

Where `<wavedir>` is the main PV-WAVE directory.

The standard color tables are listed in the following table.

Standard Color Tables

No.	Color Table Name	No.	Color Table Name
0	Black and White 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

Standard Color Tables (Continued)

No.	Color Table Name	No.	Color Table Name
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. For example, the following command loads the Red Temperature color table:

```
LOADCT, 3
```

NOTE 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.

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* in the .

Modifying the Predefined Color Tables

The MODIFYCT procedure is used to update the file `colors.tbl` 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 file is in:

(UNIX) `<wavedir>/bin`

(OpenVMS) `<wavedir>:[BIN]`

(Windows) `<wavedir>\bin`

Where `<wavedir>` is the main PV-WAVE directory.

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 11-2*.
- **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 11-3* on page 286.
- **WgCtTool** — A simple widget that can be used interactively to modify a 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 11-4* on page 288.

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.

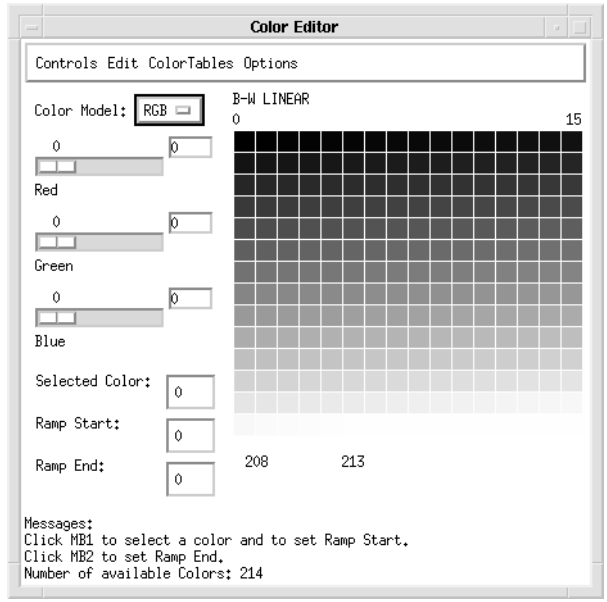


Figure 11-2 PV-WAVE provides several widget-based tools that you can use to interactively modify the color tables. Here, the WgCeditTool window displays color table number 0, B–W Linear. The WgCeditTool window lets you use the mouse to create a new color table based on either the HLS, HSV, or RGB color systems.

Shifting the Color Table to the Left or Right

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.

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.

TIP Following the same basic procedure, you can experiment with other 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 288. 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 236 (Windows) or 256 (UNIX/OpenVMS) (if you are using all available 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 color table. The horizontal form of the color bar is shown in *Figure 11-3*.

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.

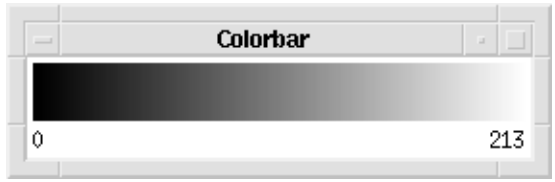


Figure 11-3 WgCbarTool creates an array of colors that match the colors in the current color table; the color array can be shifted to the right or left using the mouse. This color bar widget has been created using the /Horizontal option; the default is for the color bar to be displayed in a vertical 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 288. 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 (UNIX/OpenVMS) or 236 (Windows) 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

For example, the color table's red, green, and blue components could be stretched to emphasize a certain range of values in the image. To stretch all three components, 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 288.

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*.

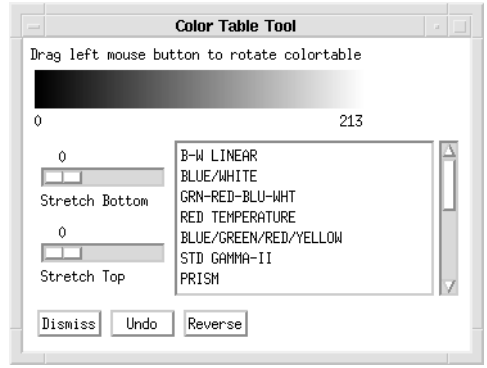


Figure 11-4 The WgCtTool window lets you interactively modify system color tables by stretching, rotating, and reversing them. The color bar in WgCtTool is the same one shown separately in *Figure 11-3*.

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 Plot Colors

The currently loaded color table determines 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 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 282.

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 291.

For more information about the standard color tables, refer to *Loading a Predefined Color Table: LOADCT* on page 281.

Default Plot Colors

By default, when drawing vector graphics on the screen using the default color table, Black and White 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 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 289.

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 the system variable !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.

NOTE For monochrome systems, !D.N_Colors is 2, and for color systems it is normally 256. Under Windows, !D.N_Colors is always 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.

UNIX and OpenVMS USERS For more information about why colors are reserved for the window manager in an X environment, refer to the *PV-WAVE Reference*.

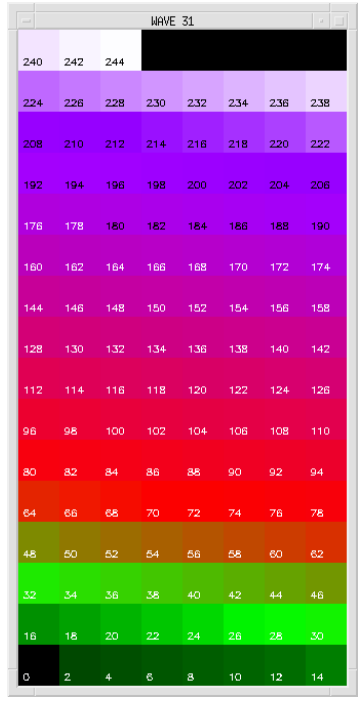


Figure 11-5 PV-WAVE provides sixteen standard color tables, and users can easily modify these color tables or define their own. Here, the COLOR_PALETTE procedure is displaying every other color in the current color table, along with its color table index. The black cells in the upper-right corner of the window represent colors that are not available to PV-WAVE because they have been reserved by some other application, such as the window manager. The Motif version of the color palette is shown here.

TIP The COLOR_PALETTE procedure displays an array of color cells and the color table index associated with each one, as shown in *Figure 11-2*. The largest number you see displayed in the array of color cells reflects the current value of !D.N_Colors. For more information on this procedure, see the *PV-WAVE Reference*.

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
    ; 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 (UNIX/OpenVMS)

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 the *PV-WAVE Reference* 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 the *PV-WAVE Reference* 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

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 the *PV-WAVE Reference*.

The write mask is used to control how one graphics pattern interacts with another graphics pattern when plotting to a graphics window. The write mask allows you to create special effects when overlaying or superimposing images and patterns.

UNIX and OpenVMS USERS 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 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. 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:

- **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 color tables.

Interactive (Wave Widgets) Color Table Procedures

The procedures listed in this section are WAVE Widgets applications, and thus are available using the Motif look-and-feel. For more information on WAVE Widgets, refer to the *PV-WAVE Application Developer's Guide*.

- **WgCbarTool** — This procedure creates a simple color bar that can be used to view and interactively shift a 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 color tables in many different ways.
- **WgCtTool** — This procedure creates a simple widget that can be used interactively to modify a 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. 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 below create windows that have a "generic" look-and-feel:

- **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 showing 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.

Windows USERS Some of these routines, such as C_EDIT and COLOR_EDIT, are written to be used with a three-button mouse. If you are using a two-button mouse, you can use the <Alt> key in combination with the left mouse button to simulate a middle mouse button.

12

CHAPTER

Mapping with PV-WAVE

The PV-WAVE mapping procedures let you create a variety of mapping applications. Scientific data, economic data, aerial photography data, and other kinds of data can be plotted with maps generated by PV-WAVE.

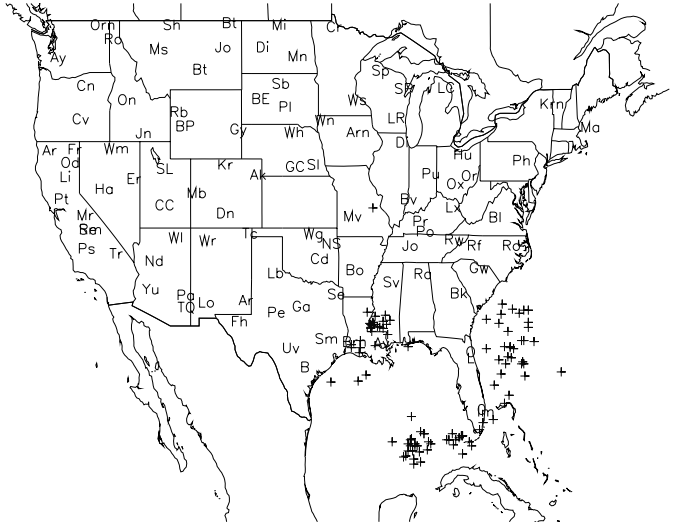


Figure 12-1 The data points plotted on this map of the United States represent lightning strikes recorded by remote sensors.

This chapter discusses how to use the PV-WAVE mapping procedures, and includes the following topics:

- Introduction
- Using Map Projections and Datasets

- Creating and Customizing Maps
- How to Optimize Your Mapping Application
- Accessing Other Map Datasets
- Defining Your Own Projections
- Creating Interactive Map Applications

Introduction

Typical uses for the PV-WAVE mapping procedures include:

- Scientific applications where wide-area data is plotted with coastline and political boundaries.
- Business applications where geographic data is displayed and highlighted to reflect a measured quantity.
- Military, environmental, and remote sensing applications where satellite imagery and digitized aerial photography are integrated with maps.

The mapping procedures, map datasets, and demonstration files are located in the PV-WAVE mapping directory:

(UNIX) `$VNI_DIR/mapping-1_1`

(OpenVMS) `VNI_DIR:[MAPPING-1_1]`

(Windows) `%VNI_DIR%\mapping-1_1`

Where `VNI_DIR` is the main Visual Numerics installation directory.

The PV-WAVE mapping procedures can be adapted easily to work with your own projections and map datasets. The mapping procedures include:

- **MAP** — Plots map data with a specified projection.
- **MAP_CONTOUR** — Plots contours on a map.
- **MAP_PLOTS** — Plots vectors or points on the current map projection.
- **MAP_POLYFILL** — Fills specified regions of a map.
- **MAP_REVERSE** — Converts X-Y coordinate data to longitude and latitude coordinates.
- **MAP_VELOVECT** — Plots a two-dimensional vector field on a map.
- **MAP_XYOUTS** — Adds text to a map.
- **USGS_NAMES** — Queries a database of longitude/latitude coordinates for states, counties, cities, and towns in the United States.

Using Map Projections and Datasets

This section is not intended to teach mapping concepts, but rather to highlight some of the concepts that are central to using the PV-WAVE mapping routines.

For more information on mapping projections and in-depth discussions of algorithms and uses of the projections PV-WAVE generates, refer to the following publications:

> *Map Projections Used by the U.S. Geological Survey*, Geological Survey Bulletin 1532, John P. Snyder, Second Edition, 1983.
>
> *An Album of Map Projections*, U.S. Geological Survey Professional Papers 1453, John P. Snyder and Philip M. Voxland, 1989.

Both are available from:

> USGS ESIC: Open File Report Sales
> Box 25286, Building 810
> Denver Federal Center
> Denver, CO USA 80225
> Phone: (303) 236-7476
> FAX: (303) 236-4031

What Are Map Projections?

A central problem facing cartographers for centuries was how to represent the features of a spherical globe on a flat map. Many methods have been devised for "flattening" the globe onto a map, and these methods are called map projections. PV-WAVE can generate 17 different types of projections. It is also possible for you to design your own algorithms and use them in PV-WAVE.

A map projection transforms spherical coordinates into two-dimensional X-Y coordinates. The spherical coordinates of the globe are defined by lines of longitude and latitude.

Longitude is the angle in degrees east or west of the prime meridian passing through Greenwich, England, and latitude is the angle in degrees north or south of the Equator.

Types of Projections

Each different map projection preserves different characteristics of the globe it represents. The characteristics preserved by four important projection types are described below:

- **Equal Area Projection** — Preserves the relative area of features at the expense of distortions in shape, angles, and scale. In an equal area projection, a coin placed on any part of the map will cover the same area.
- **Conformal Projection** — Preserves the shape of small features correctly, but large feature are distorted. Most large scale maps use some type of conformal projection.
- **Equidistant Projection** — Preserves the scale or measured distance between certain points and all other points on the map. This allows true distances to be measured using a ruler.
- **Azimuthal Projection** — Preserves local direction, or the angle between one point and other points on the map.

Projections can exhibit one or more of the above properties, thus there are azimuthal equidistant and azimuthal equal-area projections. There are subclasses of these projections which preserve more specialized characteristics. On Mercator projections all rhumb lines (lines of constant direction) are shown as straight lines, and on Stereographic projections all small circles (e.g. lines of latitude) and great circles (intersection of a plane passing through the center of the sphere and the surface of the sphere) are shown as circles on the map.

In order to achieve some of the above properties, map projections are usually constructed in such a way that the surface of a sphere is “projected” or mapped to either a cylinder, cone, or plane (referred to as azimuthal). Thus the projection may refer to “Conic”, “Cylindrical”, or “Azimuthal” in its name to identify the construction method.

Map Projections Available in PV-WAVE

PV-WAVE can generate the following map projections:

- Equidistant Cylindrical
- Lambert Conformal Conic
- Cylindrical Mercator
- Sinusoidal
- Albers Equal-Area Conic
- Polyconic
- Polar Stereographic

- Oblique Stereographic
- Oblique Orthographical
- Polar Orthographical
- Oblique Azimuthal Equidistant Oblique
- Polar Azimuthal Equidistant Oblique
- Polar Azimuthal Equal-Area
- Oblique Azimuthal Equal-Area
- Transverse Mercator
- Mollweide (Ellipsoid)
- Satellite (3D mapping onto a sphere)
- User-defined projection

What Are Map Datasets?

In PV-WAVE a map dataset is a set of polylines (a series of connected points) or polygons (points which describe a filled area). These polylines or polygons can have a number of classification attributes associated with them which aid in selecting features to be plotted. These attributes allow the desired polylines or polygons for a map to be selected based on the area being mapped and the features you want to plot.

Two datasets are included with PV-WAVE for creating world and US maps: The World Databank II dataset for global maps, and a dataset based on the USGS Digital Line Graph data for U.S. maps. In addition, a USGS database of U.S. map information is included with PV-WAVE.

The World Databank II Dataset

World Databank II is the default dataset used by the PV-WAVE MAP procedure. The World Databank II dataset is a subset of a public domain dataset provided by the U.S. Department of Commerce, merged with updated country data from the National Imagery and Mapping Agency (NIMA). All of the attribute information from the original dataset is provided in PV-WAVE, but the resolution has been reduced by sampling the polylines in order to make the dataset manageable both in terms of disk space and memory requirements.

The subsetted dataset contains approximately 300,000 points, which provides good resolution for most applications. The dataset consists of a series of polylines, and each polyline has attributes associated with it. You can subset a map by specifying these attributes with the *Select* keyword (to the MAP procedure). The attributes include coastlines/islands/lakes, rivers, international boundaries, and U.S. state boundaries.

The USGS Digital Line Graph Dataset

The USGS Digital Line Graph Dataset is composed of polygons that draw U.S. states and counties. The polygons allow you to create either line maps or maps filled with color. The *Select* keyword (to the MAP procedure) lets you plot specific states and counties. This dataset can be selected by using the keyword `Data = 'usgs_db'` with the MAP procedure.

The USGS Name Database

The USGS_NAMES function queries a built-in database of populated places in the United States. This database lets you find the longitude and latitude and for most cities and towns in the U.S. In addition, you can use the database to determine the FIPS codes for states and counties. See the *PV-WAVE Reference* for details on this procedure.

Reading Other Map Datasets Into PV-WAVE

The World Databank II and USGS Digital Line Graph datasets are provided with PV-WAVE, as are procedures used to read them into PV-WAVE. To read another dataset other than World Databank II and USGS Digital Line Graph Dataset, you must write a procedure tailored to read that dataset. For more information, see *Accessing Other Map Datasets* on page 317.

Creating and Customizing Maps

This section explains how to create maps in PV-WAVE using the MAP procedure and its keywords. In addition, procedures for annotating maps, creating map overlays, and combining maps and images are discussed.

Plotting a World Map

The MAP procedure, by default, displays a world coastline map taken from the World Databank II dataset.

For example, the following MAP call produces the map shown in *Figure 12-2*.

```
TEK_COLOR

MAP, Range = [-180, -90, 180, 90], $
     Select = {,GROUP:['cil','bdy','pby',$
     'riv']}, Color = -1, /Gridlines, $
     Gridstyle = 1, Gridcolor = 15
```

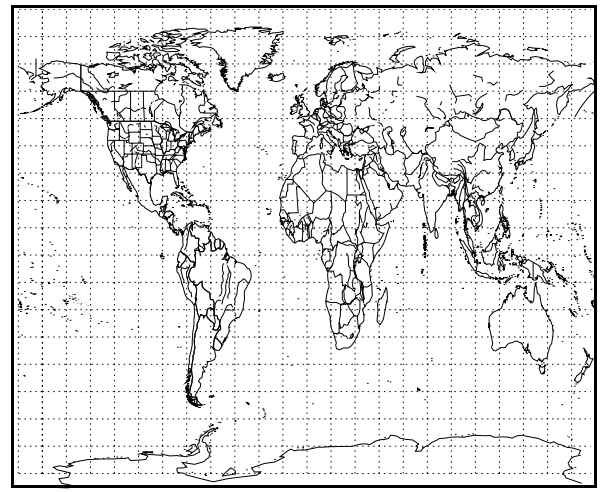


Figure 12-2 An equidistant cylindrical projection plotted from World Databank II data. Coastlines, islands, lakes, political boundaries, and rivers are plotted, as well as longitude/latitude gridlines.

In this example, keywords are used to specify the range of data, select the map features to plot, create longitude/latitude gridlines, and add color. MAP accepts some additional optional keywords. Some of the keywords are discussed in the following sections. For a complete list and description of the keywords, see the description of MAP in the *PV-WAVE Reference*.

The *Data* keyword is used to specify the map dataset to plot. The World Databank II dataset (the default) and the USGS Digital Line Graph Dataset are provided with PV-WAVE. You can also write custom procedures to read other map datasets. For example:

```
MAP, Data = 'usgs_db', Range = $
      [-125, 25,-70,55], /Gridlines, /Axes, Gridstyle = 2
        ; Plots a U.S. map using data from the USGS Digital Line
        ; Graph Dataset. The Range keyword is used to specify the
        ; map region in longitude/latitude coordinates.

MAP, Data = 'mydbase'
    ; Plots a map using data from a dataset that you have supplied and
    ; for which a custom read procedure has been written.
```

NOTE By default, MAP plots vector data. That is, it works like the PLOT command and uses some of the same keywords as PLOT. If you specify the *Filled* keyword to MAP, then MAP works like POLYFILL and uses some of the same keywords as POLYFILL. The MAP procedure can only produce a "filled" map if the dataset it reads provides polygon data. Note that the World Databank II dataset does *not* provide polygon data, but the USGS Digital Line Graph Dataset does.

Specifying a Map Projection

The *Projection* keyword for the MAP procedure lets you specify a projection for the map that is drawn. PV-WAVE provides several built-in projections, but you can also add your own projection algorithm.

For example:

```
MAP, Projection = 3
    ; Create a Cylindrical Mercator projection using the
    ; World Databank II data.

MAP, User = 'myprojection'
    ; Use a projection algorithm supplied by a user. The projection name,
    ; "myprojection", is the name of a PV-WAVE routine in which the
    ; projection is defined.
```

For information on adding your own projection algorithm to PV-WAVE, see *Defining Your Own Projections* on page 320.

To specify a projection, set the *Projection* keyword to the corresponding map projection index number. The map projections and their index numbers are:

Map Projections in PV-WAVE

Index	Projection	Index	Projection
1	Equidistant Cylindrical	11	Oblique Azimuthal Equidistant Oblique
2	Lambert Conformal Conic	12	Polar Azimuthal Equidistant Oblique
3	Cylindrical Mercator	13	Polar Azimuthal Equal-Area
4	Sinusoidal	14	Oblique Azimuthal Equal-Area
5	Albers Equal-Area Conic	15	Transverse Mercator
6	Polyconic	16	Mollweide (Ellipsoid)
7	Polar Stereographic	99	Satellite (3D mapping onto a sphere)
8	Oblique Stereographic	–1	User-defined projection (automatically set if the *User* keyword is supplied)
9	Oblique Orthographical	0	No projection
10	Polar Orthographical		

Subsetting the Map Dataset

This section discusses the *Select*, *Range*, *Zoom*, *Center*, and *Resolution* keywords. These MAP keywords are used to subset the map dataset in different ways. For more information on subsetting, see *How to Optimize Your Mapping Application* on page 313.

Selecting Map Attributes

Use the MAP *Select* keyword to specify the type(s) of map data (attributes) to plot.

To use the *Select* keyword, you need to know the special attributes (e.g., cities, political boundaries, rivers, continents) that are defined in the dataset. For example, the World Databank II includes coastlines, international boundaries, state boundaries, and rivers.

The following MAP command uses the *Select* keyword to specify that "CIL" (coastline, island, and lake) and "RIV" (river) data be plotted.

```
MAP, SELECT = {, GROUP: ['CIL', 'RIV']}
```

NOTE The *Select* keyword takes an unnamed structure as its input. For information on unnamed structures, refer to the *PV-WAVE Programmer's Guide*.

Specifying the Map Limits

There are two ways to specify the portion of a map dataset to display. You can use the *Range* keyword or the *Zoom* and *Center* keywords.

Using Range Keyword

To use the *Range* keyword, you need to know the extent of the map data (its range in longitude and latitude). The World Databank II dataset, for example, is global in extent.

The *Range* keyword specifies the range of longitude and latitude values to be displayed. *Range* requires a four-element array containing the minimum longitude, minimum latitude, maximum longitude, and maximum latitude values to be plotted.

For example, the following MAP command uses the World Databank II data and plots the world map from between –90 and 90 degrees longitude and between –45 and 45 degrees latitude.

```
MAP, RANGE = [-90, -45, 90, 45]
```

Using Zoom and Center

For some applications the *Zoom* and *Center* keywords might be more convenient to use than the *Range* keyword to specify map limits.

To "zoom" in on a point on a map, use the *Zoom* and *Center* keywords. *Center* specifies a two-element array containing the longitude and latitude of the point to zoom in on. The *Zoom* keyword specifies a "zoom factor". A zoom factor of 1 (the default) plots the entire globe. A zoom factor of 2 plots one-half of the globe, and so on.

For example, the following MAP call produces the plot shown in *Figure 12-3*:

```
MAP, Center = [-90, 30], Zoom = 2, $
     /Gridlines, Gridstyle = 1
```

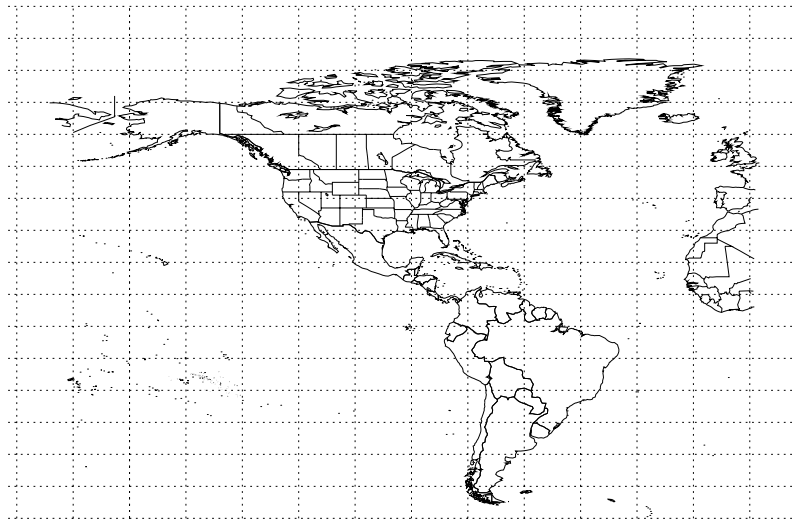


Figure 12-3 A partial world map plotted using the Center and Zoom keywords.

Plotting Great Circles, Straight Lines, and Text

Use the MAP_PLOTS procedure to draw either great circles or arbitrary straight lines on a map. MAP_PLOTS can also be used to compute geographical distances. To annotate a map, use the MAP_XYOUTS procedure.

Drawing Great Circles

A great circle is the intersection between a plane passing through the center of a sphere and the surface of a sphere. On a map, great circle lines represent the

shortest distance between two geographical points. On most projections, great circles appear as curved lines.

By default, MAP_PLOTS computes the great circle between two points on a map projection and draws the great circle line.

Drawing Arbitrary Straight Lines

Use MAP_PLOTS with the *NoCircle* keyword to draw straight lines between two points on a map. Straight lines can be used to highlight or draw attention to a particular feature. The lines drawn are not great circle lines.

Calculating Distances

The *Distance* keyword to MAP_PLOTS returns in a named variable the actual distance in miles or kilometers between points. If two points are plotted, the distance is returned as a scalar; if multiple points are plotted, then an array of distance values is returned.

Adding Text to Maps

The MAP_XYOUTS procedure lets you position text on a map at specified longitude and latitude coordinates. This routine takes as parameters a point, specified as a longitude and latitude coordinate, and a text string. Keywords let you modify the text size, thickness, color, and alignment.

Example

The following example uses MAP, MAP_PLOTS and MAP_XYOUTS to plot a great circle between two cities (Boulder, Colorado and London, England) and label the cities. The distance between the cities is also calculated and placed into a text label. (The map produced by this code is shown in *Figure 12-4* on page 308.)

```
MAP, RANGE = [-150, 30, 30, 70], /Axes, $
     /GridLines, GridColor = 10, GridStyle = 1
       ; Plot the map.

MAP_PLOTS, [-105.3, -0.1], [40.0, 51.5],$
     Distance = d, /Miles, Color = 5, $
     Psym = -2, Thick = 2
       ; Plot a great circle between two points. Return the
       ; distance between the points with the Distance keyword.

MAP_XYOUTS, -103.0, 38.0, 'Boulder', $
     Color = 5, Charsize = 1.5, Charthick = 2
       ; Label one city.
```

```
MAP_XYOUTS,    2.0, 49.0, 'London',  $
      Color = 5, Charsize = 1.5, Charthick = 2
        ; Label the other city.
MAP_XYOUTS,  -65.0, 42.0, 'Distance =' + $
      STRCOMPRESS(STRING(d(0), $
      Format = '(I5)'))+ ' miles', $
      Color = 5, Charsize = 1.5
        ; Add a text string containing the distance.
```

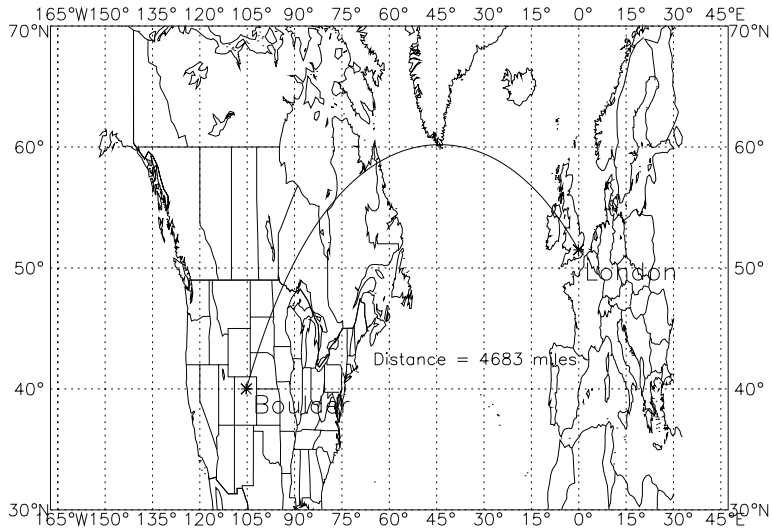


Figure 12-4 A map projection that shows the great circle arc and labels the distance between Boulder, Colorado and London, England.

Adding an Image Under the Map

Use the *Image* keyword to specify the name of an image (2D array) to be drawn under the map. The image is warped to fill the entire area specified by the *Range* keyword.

The following example warps a 2D array of global elevation data onto a sinusoidal map projection of the earth. The map produced by this example code is shown in *Figure 12-5*.

```
file = '$VNI_DIR/mapping-1_1/data/'+'earth_elev.dat'
        ; (UNIX only) Get the path/filename of file containing
        ; a 2D array of global elevation data.
file = 'VNI_DIR:[MAPPING-1_1.DATA]' + 'earth_elev.dat'
```

```
        ; (OpenVMS only) Get the path/filename of the global elevation data.
elev = FLTARR(720, 360)
    ; Create array to hold image.
OPENR, 1, file, /Xdr
    ; Open and read the image data file into the array.
READU, 1, elev
CLOSE, 1
WINDOW, Xsize=720, Ysize=360, Colors=128
TVLCT, 150, 150, 150, 63
red = BYTARR(63)
grn = BYTSCL(FINDGEN(63)^2.0)
blu = BYTSCL((FINDGEN(63)))
TVLCT, red, grn, blu, 0
TVLCT, 255, 255, 255, 127
    ; Define and load a colortable.
MAP, Projection = 4, Range = $
     [-180,-90,180,90], Image = elev
       ; Reference the image array with the Image keyword to warp
       ; the image around the map projection.
```

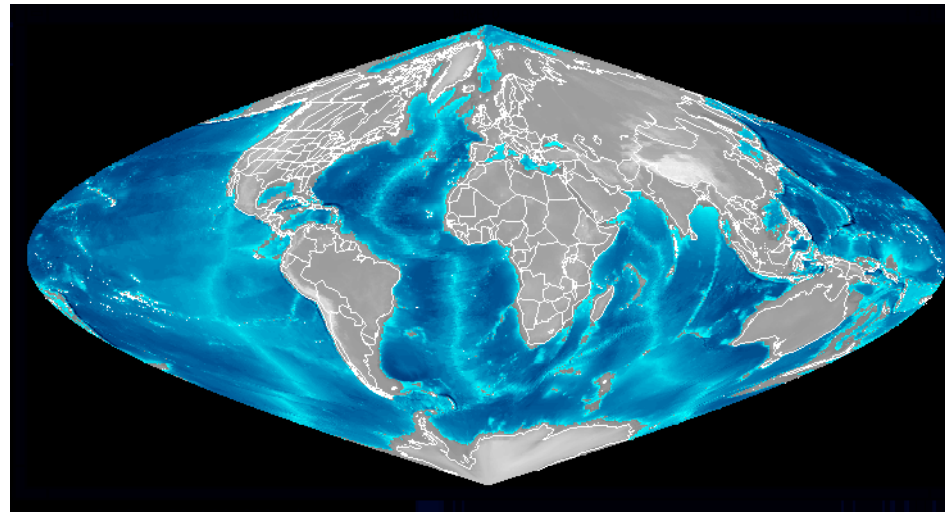


Figure 12-5 A 2D array of global elevation data is warped around a sinusoidal map projection of the globe. This map data is displayed by PV-WAVE from the World Databank II dataset.

Adding Contour Lines

MAP_CONTOUR lets you overlay contours on a map. This routine works like the regular CONTOUR procedure in PV-WAVE, except that MAP_CONTOUR assumes that the X and Y vectors specified or created by default are specified in terms of longitude and latitude coordinates.

When plotted, the contour data is projected so that the contour lines accurately describe features on the surface of the globe.

The following example plots contour data on a map. The result is shown in *Figure 12-6*.

```
file = '$VNI_DIR/mapping-1_1/data/'+$
     'earth_elev.dat'
       ; (UNIX only) Get the path of file containing 2D array of
       ; global elevations.
file = 'VNI_DIR:[MAPPING-1_1.DATA]' +$
     'earth_elev.dat'
       ; (OpenVMS only) Get the path of the global elevation data.
file = '%VNI_DIR%\mapping-1_1\data\'+$
     'earth_elev.dat'
       ; (Windows only) Get the path of file containing 2D array of
       ; global elevations.
elev = FLTARR(720, 360)
    ; Create an array to hold the image.
OPENR, 1, file, /Xdr
    ; Open and read the image data file into the array.
READU, 1, elev
CLOSE, 1
water = BYTSCL(elev, Max = 0.0, Top = 63)
land = BYTSCL(elev, Min = 0.0, Top = 63)
elev = water + land
elev = REBIN(elev, 360, 180)
DATA = FLOAT(elev(-150+180:30+180, 30+90:70+90))
       ; Subset the array of elevation data. The elevation dataset
       ; contains an elevation for each degree of longitude and
       ; latitude, from –180 to 180 degrees longitude and
       ; from –90 to 90 degrees latitude. The array expressions in
       ; this command subset the data corresponding to the range
       ; of data used in the MAP procedure call.
MAP, Projection = 4, Range = $
   [-150, 30, 30, 70], Thick = 2
TEK_COLOR
```

```
MAP_CONTOUR, DATA, C_Colors = $
    [2,8,16,18,20,4], Levels = [20,30,35,40,50]
```

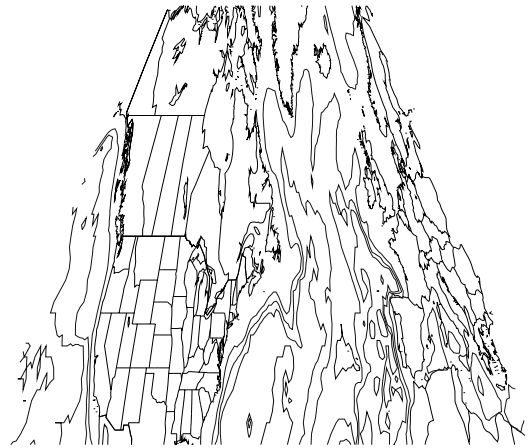


Figure 12-6 Contour lines are plotted over a map projection.

In addition to line plot overlays, the *Fill* and *Pattern* keywords allow contours to be filled in the same way that POLYCONTOUR is used to fill contour plots. The same cautions associated with POLYCONTOUR apply, in that all contour lines must be closed. This is usually accomplished by padding the data with zeros or some other value outside the range of the data. For an example of this technique, see the POLYCONTOUR procedure in the *PV-WAVE Reference*.

NOTE MAP_CONTOUR with the *Fill* keyword is not supported for projection 99 (Satellite), but good results can be obtained by creating a two-dimensional image with CONTOUR and POLYCONTOUR and then using the *Image* keyword with MAP to wrap this image onto the globe.

Adding Vector Lines

MAP_VELOVECT creates two-dimensional velocity vector plots. It works just like the PV-WAVE VELOVECT procedure, except that MAP_VELOVECT takes longitude and latitude coordinates as input. When the vector lines are plotted, the current map projection is taken into account so that the vector lines are depicted accurately on the map, as shown in *Figure 12-7*.

```
u = fltarr(20, 20)
   v = fltarr(20, 20)
```

```
   FOR j = 0, 19 DO BEGIN
      FOR i = 0, 19 DO BEGIN
         x = 0.05 * float(i)
         z = 0.05 * float(j)
         u(i, j) = -sin(!Pi*x) * cos(!Pi*z)
         v(i, j) = cos(!Pi*x) * sin(!Pi*z)
      ENDFOR
   ENDFOR
MAP, Projection = 4, Range = [-150, 30, 30, 70]
MAP_VELOVECT, u, v, Color = 5
```

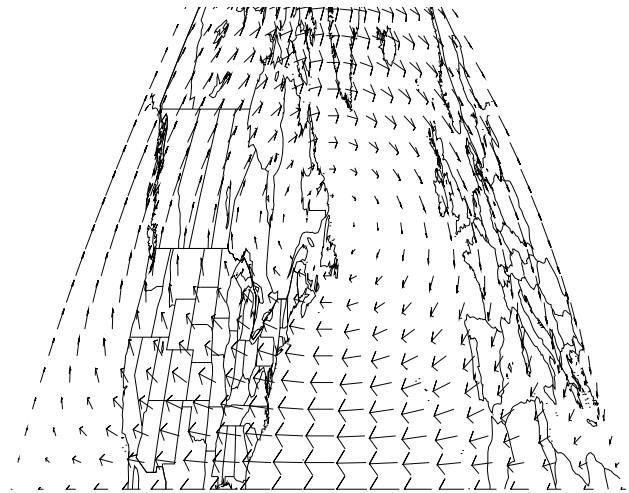


Figure 12-7 Vector lines are added to a map projection.

Creating Filled Maps

The MAP_POLYFILL routine provides essentially the same functionality as the POLYFILL routine for plotting filled polygons, except that the data provided is assumed to be longitude/latitude data which will be projected before being plotted. Standard POLYFILL keywords such as *Color*, *Pattern*, *Fill_Pattern*, *Line_Fill*, *Linestyle*, *Thick*, *Psym*, *Spacing* and *Symsize* can be used to specify the characteristics of the polygon to be plotted.

NOTE MAP_POLYFILL cannot be used with the Satellite (3D Mapping onto a Sphere) projection.

How to Optimize Your Mapping Application

This section describes several methods for improving the performance of your mapping application. These methods will help you to design a mapping application that performs well at an acceptable resolution.

In general, map data is processed and displayed in four stages in PV-WAVE. When considering how to improve the performance of a mapping application, it is helpful to keep in mind these four stages, illustrated in *Figure 12-8*.

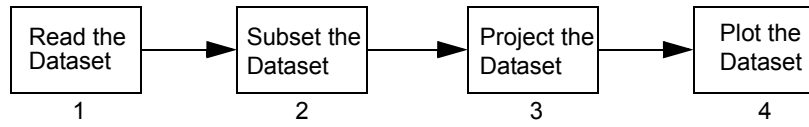


Figure 12-8 Basic stages required to generate a map in PV-WAVE.

This section focuses on performance improvements that can be gained in stages 1 and 2. Stage 3 cannot be improved substantially, because it depends on array operations, which are already greatly optimized by PV-WAVE.

The performance of stage 4, plotting the data, depends largely on the size of the dataset being plotted. Reducing the size of the dataset is the primary focus of this section.

A summary of methods for improving the speed at which PV-WAVE reads and displays map data is shown in the following table. More details are provided in this section.

Methods for Optimizing Mapping Applications

Optimization Method	Advantages	Disadvantages
Subsetting data with MAP keywords *Select*, *Range*, *Center*, and *Zoom*.	Keywords are easy to use and accessible to all PV-WAVE users. Allow you to display only those portions of the map that are of interest. Reduces the overall amount of data to be plotted.	Subsetting can be a slow process for very large datasets. May not be sufficient if a large portion of the map must be displayed at once.

Methods for Optimizing Mapping Applications (Continued)

Optimization Method	Advantages	Disadvantages
Reduce the number of map data points plotted with the *Resolution* keyword.	Useful for wide-area maps where fine detail may not be necessary. Reduces the overall amount of data to be plotted.	Cannot be used with the usgs_db dataset.
Use the *File_Path* keyword to save a subsetted dataset in a binary file that can be read and displayed quickly with the *Read_Path* keyword.	Subsequent calls to MAP can skip the steps of reading and subsetting the map data. Provides excellent performance for applications that make calls to the MAP procedure to plot and replot the same dataset.	Reduces the detail/accuracy of the original map.
Use DEVICE, /Copy or TVRD to create an image of a basemap that can be rapidly redisplayed.	Provides excellent performance.	Resolution is limited to the resolution of the window into which the data is copied.
Write a C or FORTRAN procedure to read and subset a large map dataset before placing the data in memory.	Useful for handling user-supplied datasets that are too large to read into memory in one chunk. PV-WAVE can access the C or FORTRAN procedure via LINKNLOAD. Reduces the overall amount of data to be plotted.	Only useful for C or FORTRAN programmers. Some knowledge of PV-WAVE connectivity features is also required. Reduces the detail/accuracy of the original map.

Subsetting Data with MAP Procedure Keywords

Perhaps the easiest way to improve the performance of a mapping application is to subset the map dataset using keywords provided with the MAP procedure. These keywords are passed directly to the procedure that reads the map dataset, so that the data is subsetted before it is read into memory.

For additional information on the keywords described below, see the description of the MAP procedure in the *PV-WAVE Reference*. See also *Subsetting the Map Dataset* on page 305.

Subsetting with the Select Keyword

The *Select* keyword reduces the amount of data returned to the MAP procedure. It lets you specify only the map features that you want to plot from the dataset. Selecting a subset of the available map features can improve performance significantly.

The *Select* keyword can be used to subset the `world_db` and `usgs_db` datasets.

Subsetting with Range, Zoom, and Center Keywords

These keywords let you subset a map dataset by specifying a range of data to plot. In other words, only the data that falls within a selected area is returned by the MAP procedure. With the *Range* keyword, you specify an area to plot within a range of longitude and latitude values. The *Zoom* and *Center* keywords allow you display an area surrounding a specified point.

Subsetting with the Resolution Keyword

The *Resolution* keyword reduces (samples) the number of data points that are plotted, thereby reducing the map resolution. This keyword can be useful when plotting a wide area map where the full resolution of the database might not be discernible given the resolution of the output device.

This keyword can only be used with the `world_db` dataset.

Use File_Path and Read_Path Keywords to Avoid Re-reading Data

As noted previously, the procedure that reads the dataset (e.g., `world_db` or `usgs_db`) is responsible for performing most of the subsetting. However, each time the MAP procedure is called, the map-reading procedure is called, and the process of subsetting the data is repeated.

This ensures that the full resolution of these datasets can be accessed when needed, but can slow down performance when the same data subset must be plotted repeatedly.

The *File_Path* and *Read_Path* keywords to the MAP procedure store a subsetted dataset in a binary file and then read it for subsequent calls to MAP. The *Read_Path* keyword restores the data quickly without calling the dataset-reading procedure.

Thus the mapping process is reduced to reading a small dataset, projecting it, and plotting it. This method provides the optimal performance and the best resolution for large datasets. The demonstration routines for mapping use this technique to

optimize their speed of execution, and the same technique can be used in your own applications. The demonstration routines are in:

(UNIX) `$VNI_DIR/mapping-1_1/demo`

(OpenVMS) `VNI_DIR:[MAPPING-1_1.DEMO]`

(Windows) `%VNI_DIR\mapping-1_1\demo`

Where `VNI_DIR` is the main Visual Numerics installation directory.

Creating a Basemap Image

There are other techniques, not strictly related to mapping, which can be used in some circumstances to further increase the performance of drawing a basemap. For instance, if a basemap needs to be redrawn quickly many times in an X Window System environment, the command:

```
DEVICE, Copy
```

can be used to rapidly update a window from a copy held in another window. The following example demonstrates this, and can be used to update a map in a fraction of a second, but with the limitation that the resolution is limited to that of the window in which the map is created.

```
WINDOW, 1, Xsize = 600, Ysize = 400, /Pixmap
    ; Create an invisible pixmap window.
MAP, Center = [10, 50], Zoom = 7
    ; Draw a map of Europe.
WINDOW, 0, Xsize = 600, Ysize = 400
    ; Create a visible window of the same size.
DEVICE, Copy = [0, 0, 600, 400, 0, 0, 1]
    ; Copy the contents of the pixmap window to the visible window.
    ; (This can be repeated indefinitely.)
```

It is also possible to use the TVRD command to copy the contents of a window containing a basemap into a byte array variable. Saving this variable in a file allows the basemap to be restored by simply reading the variable and redisplaying the image, as in this example:

```
WINDOW, Xsize = 600, Ysize = 400
    ; Create a window.
MAP, Center = [10, 50], Zoom = 7
    ; Draw a map of Europe.
basemap = TVRD(0,0, 600, 400)
    ; Copy the contents of the window to the basemap variable.
```

```
SAVE, basemap, File = 'mybasemap.dat'
    ; Save the basemap variable.
```

Then to restore the basemap later:

```
RESTORE, File = 'mybasemap.dat'
    ; Restore the basemap image.
WINDOW, 1, Xsize = 600, Ysize = 400
    ; Create a window.
TV, basemap
    ; Redisplay the basemap image.
```

Optimized Data Reading

Two procedures designed to read map datasets are provided with PV-WAVE. These procedures, `world_db` and `usgs_db`, read map data directly into memory whenever they are called.

If you supply your own dataset (that is, a dataset other than the World Databank II or USGS datasets) and write a procedure to read it, it is possible that it will be too large to read the entire dataset into memory at once.

In this case, a C or FORTRAN program can be written to read and subset the data prior to placing it in memory. Using connectivity features, such as LINKNLOAD, a PV-WAVE procedure can be written to call the C or FORTRAN program in a mapping application. This method can provide the best access speed for large datasets, and can be useful when you need to perform a lot of testing to determine how to subset the data correctly.

See *Accessing Other Map Datasets* on page 317 for information on writing a procedure to read a dataset.

Accessing Other Map Datasets

Two map datasets are included with PV-WAVE: the World Databank II dataset and the USGS Digital Line Graph Dataset. These built-in map datasets are subsets of the actual datasets, and are included to provide relatively fast and efficient access to map data while still maintaining adequate resolution when a small area is plotted.

Two procedures are provided to read these datasets: `usgs_db.pro` and `world_db.pro`. These procedures are called by the MAP procedure.

A procedure called `ascii_db.pro` is also provided in the mapping library to help you read your own map datasets that are in ASCII format. You can use the `ascii_db` procedure as a template for reading a user-defined dataset.

If you use the MAP procedure with the *Data* keyword set to `ascii_db`, use the *Select* keyword to specify the name of the file containing the ASCII data. The ASCII file containing the data must conform to the following format:

- Two columns of numbers, either comma or space separated.
- The beginning of each polyline or polygon indicated by a record with "999" in the first column and the color in the second column.
- This record is followed by any number of records containing pairs of coordinates for the X and Y or longitude and latitude.

Writing a Procedure to Read a Map Dataset

To use any map dataset, PV-WAVE must be given specific information about the dataset. This information is placed in a procedure file that is executed when the MAP procedure is called with the *Data* keyword.

A procedure for reading a map dataset reads and subsets the map dataset using attribute selections of the data. A map dataset procedure must contain the following four positional parameters:

- ***data*** — A 2-by-*n* floating point array of longitude/latitude points (in degrees) returned from the dataset.
- ***index*** — A 2-by-*m* long array containing the starting and ending indices of each polyline or polygon in the map dataset.
- ***select*** — A variable passed from the MAP procedure via its *Select* keyword. This passed variable can be an unnamed structure containing as tag fields the names of section criteria recognized by the map dataset (such as cities, countries, rivers). The use of this variable is entirely defined within the map dataset procedure.
- ***resolution*** — A variable containing the number of points to skip in a large dataset. A higher resolution value improves performance at the expense of map detail. This variable must be present in the user defined procedure, but its use is optional and can be ignored if it is not needed.

Another way to control the size of the dataset returned by the dataset-reading procedure is to use the !Map.X.Range and !Map.Y.Range system variables. These system variables contain minimum and maximum longitude and latitude values for the current plot, and can be used to subset the dataset and reduce the size of the data array returned.

NOTE When the MAP procedure is called, a new system variable !Map is created to contain parameters used by the mapping routines. These parameters are also used in user defined projections and user-defined map dataset procedures. For information on the fields of this system variable, refer to the file `map.pro` in:

(UNIX) `$VNI_DIR/mapping-1_1/lib/map.pro`

(OpenVMS) `VNI_DIR:[MAPPING-1_1.LIB]MAP.PRO`

(Windows) `%VNI_DIR%\mapping-1_1\lib\map.pro`

The dataset-reading procedure returns the data to be projected and plotted based on the selection criteria, area being plotted, and optionally the resolution desired. The procedure can read the entire map into a variable the first time it is called and then subset the data directly from memory, as is done for the World Databank II and USGS Digital Line Graph Dataset procedures. Or, the procedure can read the dataset from disk each time the procedure is called, which would be appropriate for very large datasets.

Example Programs Are Provided

For more information on how to create a procedure to read a map dataset, look at the following procedures that are provided with PV-WAVE:

(UNIX) `$VNI_DIR/mapping-1_1/lib/world_db.pro`

(OpenVMS) `VNI_DIR:[MAPPING-1_1.LIB]world_db.pro`

(Windows) `%VNI_DIR%\mapping-1_1\lib\world_db.pro`

(UNIX) `$VNI_DIR/mapping-1_1/lib/usgs_db.pro`

(OpenVMS) `VNI_DIR:[MAPPING-1_1.LIB]usgs_db.pro`

(Windows) `%VNI_DIR%\mapping-1_1\lib\usgs_db.pro`

(UNIX) `$VNI_DIR/mapping-1_1/lib/ascii_db.pro`

(OpenVMS) `VNI_DIR:[MAPPING-1_1.LIB]ascii_db.pro`

(Windows) `%VNI_DIR%\mapping-1_1\lib\ascii_db.pro`

Use these programs to guide you in writing your own procedure to read a dataset.

TIP To convert data stored as degrees, minutes, and seconds to a single floating point value for use in the mapping routines, the formula is:
value = degrees + minutes / 60.0 + seconds / 3600.0

Defining Your Own Projections

You can specify a new projection by creating a PV-WAVE procedure defining the projection algorithm. Then, call MAP and specify the name of your projection routine with the *User* keyword. The projection routine is passed a single parameter, *values*, a 2-by-*n* floating point array of longitude/latitude values to be transformed and returned in the same array.

A second parameter, *index*, must also be included, but its use is optional. On return *index* can contain a vector of integers specifying the indices in the *values* array to plot. If *index* is not used, it is assumed that all projected points in *values* are valid.

In addition a keyword parameter *Reverse* must be defined so that if the procedure is called with *Reverse*, the data can be passed through a reverse transformation from two-dimensional data coordinates to longitude/latitude values. The system variables !Map.Center and !Map.Parameters also may be used in a user defined procedure to supply other necessary information for the projection algorithm.

NOTE When the MAP procedure is called, a new system variable !Map is created to contain parameters used by the mapping routines. These parameters are also used in user defined projections and user-defined map dataset procedures. For information on the fields of this system variable, refer to the file `map.pro` in:

(UNIX) `$VNI_DIR/mapping-1_1/lib`

(OpenVMS) `VNI_DIR:[MAPPING-1_1.LIB]`

(Windows) `%VNI_DIR%\mapping-1_1\lib`

Example

An example user-defined projection is provided in:

(UNIX) `$VNI_DIR/mapping-1_1/demo/userproj.pro`

(OpenVMS) `VNI_DIR:[MAPPING-1_1.DEMO]userproj.pro`

(Windows) `%VNI_DIR%\apping-1_1\demo\userproj.pro`

This example projection program produces a tilted perspective (modified azimuthal) projection. Use this program to guide you in writing your own projection procedures.

Here is an example map produced with the "user-defined" projection `userproj.pro`:

```
MAP, Center=[-74, 41], Zoom = 2, /Gridlines, $
     Gridstyle = 1, Gridcolor = 15, $
     User = 'userproj', Parameters = [10.0, 200.0, 20.0]
```

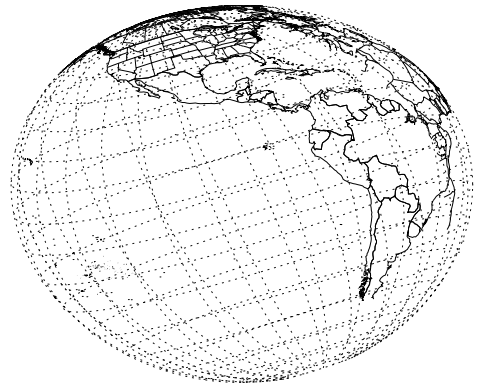


Figure 12-9 A user-defined projection produced this tilted view of the globe.

Creating Interactive Map Applications

The MAP_REVERSE procedure converts the X-Y coordinate output from routines like CURSOR and WtPointer into longitude and latitude coordinates. CURSOR and WtPointer return the coordinates of the cursor or pointer after a mouse click.

NOTE MAP_REVERSE cannot be used with the Satellite (3D Mapping onto a Sphere) projection.

TIP If you are developing an interactive widget-based mapping application, look at the demonstration program `map_test.pro` in the directory:

(UNIX) `$VNI_DIR/mapping-1_1/demo`

(OpenVMS) `VNI_DIR:[MAPPING-1_1.DEMO]`

(Windows) `%VNI_DIR%\mapping-1_1\demo`

Where `VNI_DIR` is the main Visual Numerics installation directory.

This program uses PV-WAVE Widgets to create an interactive mapping demonstration. You can copy this code and use it as a template for creating your own mapping applications. To run the demonstration, move to the `demo` directory using the PV-WAVE CD command, then type `map_test` at the PV-WAVE prompt.

13

CHAPTER

PV-WAVE on the World Wide Web

PV-WAVE provides a collection of features that allow you to process and present data across the Internet. You can also use the new functionality to work efficiently across private intranets.

PV-WAVE's Web-enabling technology allows you to:

- Develop HTML and VRML files.
- Open remote files for PV-WAVE processing.
- Use your local PV-WAVE installation as a helper application.
- Use a remote (server-side) PV-WAVE installation.

HTML, VRML, and file handling capabilities are part of the standard library and documented in the PV-WAVE Reference. Many of these routines and all of the other web-enabling features are also outlined and demonstrated in the following directory:

(UNIX) `<wavedir>/demo/web`

(OpenVMS) `<wavedir>:[DEMO.web]`

(Windows) `<wavedir>\demo\web`

where `<wavedir>` is the main PV-WAVE directory.

Standard Library Web-Enabling Routines

PV-WAVE's standard library contains Hypertext Markup Language (HTML) routines and Virtual Reality Modeling Language (VRML) routines. (For detailed

descriptions, see the PV-WAVE Reference.) Plus, the following routines have been added to facilitate file input and output:

- OPENURL
- READ_XBM
- WRITE_XBM

All of these features — previously available only in the user library or from the Visual Numerics Web site — are now fully supported and integrated into the product.

The PV-WAVE HTML routines allow you to create World Wide Web documents. Developing HTML directly with PV-WAVE allows you to put the results of your data visualization and analysis out on the Internet for others to see. Along with graphical representations and formatted reports of your data, HTML makes it convenient for you to describe and highlight important information.

The VRML routines allow you to describe three-dimensional representations of your data that can be viewed with a VRML-enabled browser.

Several other routines are also useful for processing or producing graphical material for usage across the Internet:

- IMAGE_CREATE
- IMAGE_READ
- IMAGE_WRITE

PV-WAVE as a Helper Application

From a Web browser, PV-WAVE can be set up as a helper application, allowing you to open virtually any PV-WAVE procedure anywhere on the Internet and execute it on your system.

For information on creating a helper application, see the README file in:

(UNIX) `<wavedir>/demo/web/helper-app`

(Windows) `<wavedir>\demo\web\helper-app`

where `<wavedir>` is the main PV-WAVE directory.

OpenVMS USERS This functionality is not available on OpenVMS.

Using PV-WAVE Remotely with CGI

You can also use PV-WAVE as a remote Web server in conjunction with the Common Gateway Interface (CGI). For an example of using PV-WAVE on a remote Web server, look in the directory:

(UNIX) `<wavedir>/demo/web/cgi`

(Windows) `<wavedir>\demo\web\cgi`

where `<wavedir>` is the main PV-WAVE directory.

OpenVMS USERS This functionality is not available on OpenVMS.

The CGI demonstration shows how you can use CGI to invoke PV-WAVE on a server. For more information, see the REAME in the above directory.

Another practical example of combining PV-WAVE with CGI is contained in the directory:

(UNIX) `<wavedir>/demo/web/weather`

(Windows) `<wavedir>\demo\web\weather`

where `<wavedir>` is the main PV-WAVE directory.

This example demonstrates a means by which a Java applet can retrieve data from a remote data source, analyze that data using PV-WAVE, and display it in a Web browser.

This approach has the advantage of allowing remote users access to PV-WAVE's visualization and analysis technology in conjunction with remotely located, frequently updated data sources — all while taking advantage of the computational power that resides on the server.

NOTE The CGI demonstrations require a working knowledge of CGI. Visual Numerics does not support CGI and takes no responsibility for individual CGI configurations.

14

CHAPTER

Using the XML Toolkit

This chapter introduces the capabilities of the PV-WAVE:XML Toolkit routines, with more details and examples in the PV-WAVE Reference manual.

Introduction

PV-WAVE:XML Toolkit functions let you import, interrogate, modify, create and export XML data. The data source and destination can either be variables in memory or physical XML files. You can use PV-WAVE to analyze, manipulate, and visualize imported data.

Below is a list of the PV-WAVE XML Toolkit routines. See the PV-WAVE Reference manual for a more detailed discussion of each routine.

- XmlAddNode - *Adds a node to an existing node*.
- XmlDocDump - *Creates an XML document either in memory or to a file*.
- XmlEvaluate - *Evaluates the XPATH expression*.
- XmlFreeDoc - *Frees memory for parsed documents*.
- XmlFreeNode - *Removes nodes and sub-nodes from documents*.
- XmlNewDoc - *Creates a new XML document*.
- XmlNewNode - *Creates a new XML node*.
- XmlParse - *Parses an XML file or an in-memory document and builds a node tree*.
- XmlSetAttr – *Adds or replaces an attribute for a node*.

- XmlSetContent - *Sets content for a node*.
- XmlSetContext - *Sets the context or current sub-node for queries*.
- SAXInit - *Initiates XML SAX event handler.*

What is XML

There are many good texts on XML, but in a nutshell it represents a markup language which allows information to be structured and shared between different systems. It comes from the HTML and SGML markup languages and is really a subset of those that do not have the specific goals of page layout. Here is a simple example of an XML Document:

```
<?xml version="1.0?">
<Chart>
   <ChartTitle>
      <Attribute name="Title">Line Chart</Attribute>
      <Attribute name="FontSize" value="14"/>
      <Attribute name="FontStyle" value="3"/>
   </ChartTitle>
</Chart>
```

The XML document contains a prolog which defines some basic version information, and a number of tags or “nodes” which themselves can specify content (for example: “Line Chart”) or attribute information (for example: name=“title”). The nodes can be arranged in a hierarchical tree to indicate associations and groupings.

XML is finding uses in many different applications, but a primary use is the exchange of information between different applications or processes. Because it is text based it allows information to be exchanged between different platforms and easily moved over the web. Because there are XML parsers available in many languages it is easy to write application that can create and access XML data.

The XML parser used in the PV-WAVE:XML Toolkit is based on the libxml library (http://xmlsoft.org). You can receive XML data in PV-WAVE from files, sockets, RPC’s, via JWAVE, or other sources. The XML documents can be parsed, searched, modified, and written back out to files or directly to other applications using sockets, etc. You can even create XML Documents from scratch using the toolkit. PV-WAVE can be easily integrated with other applications that use XML as an information interchange mechanism.

Starting the PV-WAVE XML Toolkit

At the WAVE> prompt, enter the following commands to load and initialize the PV-WAVE:XML Toolkit:

```
@Xml_Startup
```

or

```
Xml_Init
```

PV-WAVE:XML Toolkit Capabilities

Data Interchange Is Convenient

XML is quickly becoming a standard for interchanging structured information among different hardware, software, and operating systems. PV-WAVE can be integrated into any application where XML is used as a common communication protocol. If you can access your XML data from the computer on which PV-WAVE is running, you can extract, analyze, sort, and customize this data within PV-WAVE.

Use the Standard XML XPATH Syntax

You can query or update your XML data from PV-WAVE using the XML path language (XPATH) commands. You do not need to learn new syntax or complicated function and procedure calls.

For XML queries, PV-WAVE:XML Toolkit retrieves all of the nodes specified by the XPATH statement. The results of the query are placed in a PV-WAVE table variable.

The Simple API for XML (SAX) is Supported

If your XML data is too large to process with simple XPATH statement, you can use the Simple API for XML (SAX) to perform the same task. This is an "event-driven" API. For example, the start of an element in an XML document generates an event and notifies the user-defined function with the information associated with this event. One application of this feature is that you can retrieve the data incrementally without having to maintain a copy of the entire XML document in memory. For more information, refer to the Chapter 2, *SaxInit Procedure*, in the *PV-WAVE Reference* manual.

Handling Data Inside PV-WAVE

PV-WAVE:XML Toolkit imports query data into a PV-WAVE table variable. This format is very convenient for using other PV-WAVE procedures and functions to analyze and display the data from the query. Almost all PV-WAVE functions can access the data directly from the table. In addition, PV-WAVE has some procedures and functions that are specially designed to work with table variables. For more information, refer to the PV-WAVE functions BUILD_TABLE, QUERY_TABLE, UNIQUE, GROUP_BY, and ORDER_BY.

In addition to reading from an XML source, you can also create new or modify previously parsed XML DOM (Document Object Model) trees in memory and from this generate XML documents to be saved in files or passed to other applications.

Using the PV-WAVE XML Toolkit

There are two methods used to parse XML documents, DOM parsers and SAX parsers. DOM (Document Object Model) parsers read an entire XML document and create a memory model of this document that can be queried and manipulated. All but one routine, (name of routine here?), in the PV-WAVE:XML Toolkit are used for DOM parsing and manipulation. SAX parsers offer an event driven approach to parsing a document. Both of these methods and the PV-WAVE:XML Toolkit routines used are described below.

DOM Parser

The DOM parser builds a memory model of an entire XML document that can be queried and manipulated in powerful ways. The XmlParse routine performs this function and creates a “handle” to the DOM object which is used by other routines. Typically after parsing the document you would use the XmlEvaluate routine to perform a query on the document and return “nodes” of interest. Each node is essentially a tag in XML which can contain content and attributes.

In many applications that read XML this is probably all you would need to do. If you wanted to modify the contents of the document and pass it on to another application you could use routines like XmlSetContent and XmlSetAttr. You can also add new nodes with XmlNewNode and XmlAddNode. Finally, you can write out the internal DOM representation to an XML document with XmlDocDump.

In some cases you may need to generate an XML document entirely from scratch. You can use XmlNewDoc, XmlNewNode, XmlSetAttr and XmlAddNode, along with XmlDocDump to accomplish this. See the

SAX Parser

The second method of parsing XML documents is the SAX (Simple API for XML) parser. The single routine SAXInit is used for this. The SAX parser simply reads the XML file and calls a user specified callback function for each element it finds in the XML document. There are a number of events defined which cause the callback routine to be executed and you can filter out which of these events you are interested in.

With the SAX parser you can not do sophisticated queries on the XML document, but it is an easy way to filter a large document and pull out just particular items of interest. You can not modify or manipulate the XML document as you can with a DOM model, but simply get the contents and attributes of interest in the order that they appear in the document.

Because the PV-WAVE:XML Toolkit offers low level access to most of the functionality of the underlying parser, there are few limitations in the kinds of applications you can find for XML and PV-WAVE.

User's Guide Index

A

B

C

D

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z