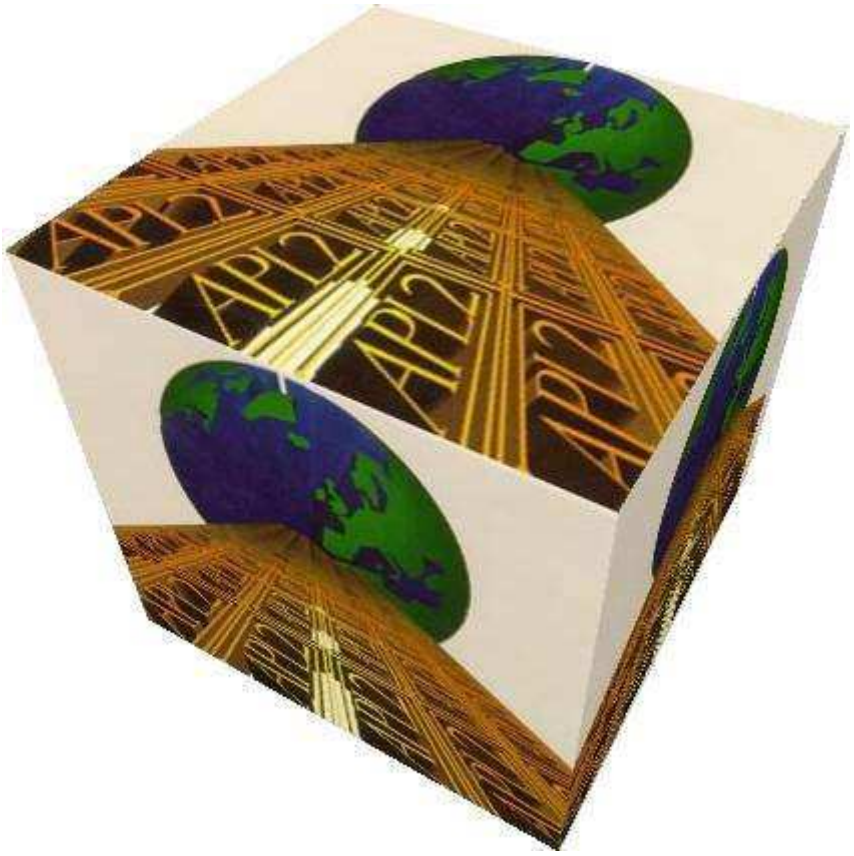
APL2 GRAPHPAK: User's Guide and Reference

SH21-1074-02

APL Products and Services
IBM Silicon Valley Laboratory
555 Bailey Avenue
San Jose, California 95141
APL2@vnet.ibm.com



Copyrights

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe on any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject material in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Programming Interface Information

This book is intended to help you write applications in APL2. It documents *General-Use Programming Interface and Associated Guidance Information* provided by APL2. General-use programming interfaces allow the customer to write programs that obtain the services of APL2.

Trademarks

IBM Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX/6000
APL2
GDDM
IBM
OS/2
WebSphere

Other Trademarks

The following terms are trademarks of other companies:

PostScript	Adobe Systems, Inc.
Sun	Sun Microsystems, Inc.
Solaris	Sun Microsystems, Inc.
TrueType	Apple Computer, Inc.
Windows	Microsoft Corporation
Windows NT	Microsoft Corporation

We Would Like to Hear from You

APL2 GRAPHPAK: User's Guide and Reference

Please let us know how you feel about this online documentation by placing a check mark in one of the columns following each question below:

To return this form, print it, write your comments, and mail it to:

> International Business Machines Corporation
> APL Products and Services - PGUA/E1
> 555 Bailey Avenue
> San Jose, California 95141
> USA

For postage-paid mailing, please give the form to your IBM representative.

You can also send us your comments by email. To send us this form, copy it to a file, write your comments using a file editor, and then send it to:

> apl2@vnet.ibm.com

Overall, how satisfied are you with the online documentation?

	Very Satisfied 1	2	3	Very Dissatisfied 4
Overall Satisfaction	___	___	___	___

Are you satisfied that the online documentation is:

Accurate	___	___	___	___
Complete	___	___	___	___
Easy to find	___	___	___	___
Easy to understand	___	___	___	___
Well organized	___	___	___	___
Applicable to your tasks	___	___	___	___

Please tell us how we can improve the online documentation:

```
_____________________________________________________________
_____________________________________________________________
_____________________________________________________________
```

Thank you! May we contact you to discuss your responses?

```
___Yes  ___No
Name:
```

```
_____________________________________________
Title:
_____________________________________________
Company or Organization:
_____________________________________________
Address:
_____________________________________________
_____________________________________________
_____________________________________________
Phone:
(___)_____________________________________
E-mail:
_____________________________________________
```

Please do not use this form to request IBM publications. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office servicing your locality.

Preface

GRAPHPAK is an APL2 workspace containing predefined functions that control graphics devices. This manual tells how to use those functions to produce computer graphics.

How Much APL2 Background You Need

This manual is for people who know only a little about APL2, as well as for experienced programmers. To use this manual, you will need to know how to:

- Assign a value to a variable. (As an example, `ABC←10` assigns the value 10 to variable `ABC`.)
- Form vectors and matrices. (Probably you will use the APL2 functions *catenate*, *laminate*, and *reshape*.)
- Execute a defined function. (`DRAW X` executes the function `DRAW` with the argument `X`.)
- Write a simple APL2 function.
- `)LOAD` and `)SAVE` a workspace. And you will have to know what happens when you do those things.
- Tell when a statement has finished executing and the system is ready for more input.

You will need to know whether or not you are using the APL2 session manager. If you are using it, you will need to know how it controls the display on your screen.

You will also need to know what is meant by types of data structures in APL2: scalars (single values), vectors (1-dimensional arrays), matrices (2-dimensional arrays), and arrays of higher dimension.

You will come upon some other APL2 words also: *rank*, *global variable*, and the like. You can find definitions for most of these in the glossary at the end of this manual. If you find terms that you think ought to be in the glossary, but aren't, please let us know. The Reader's Comment Form in We Would Like to Hear from You is a good way to tell us.

How Much Mathematics Background You Need

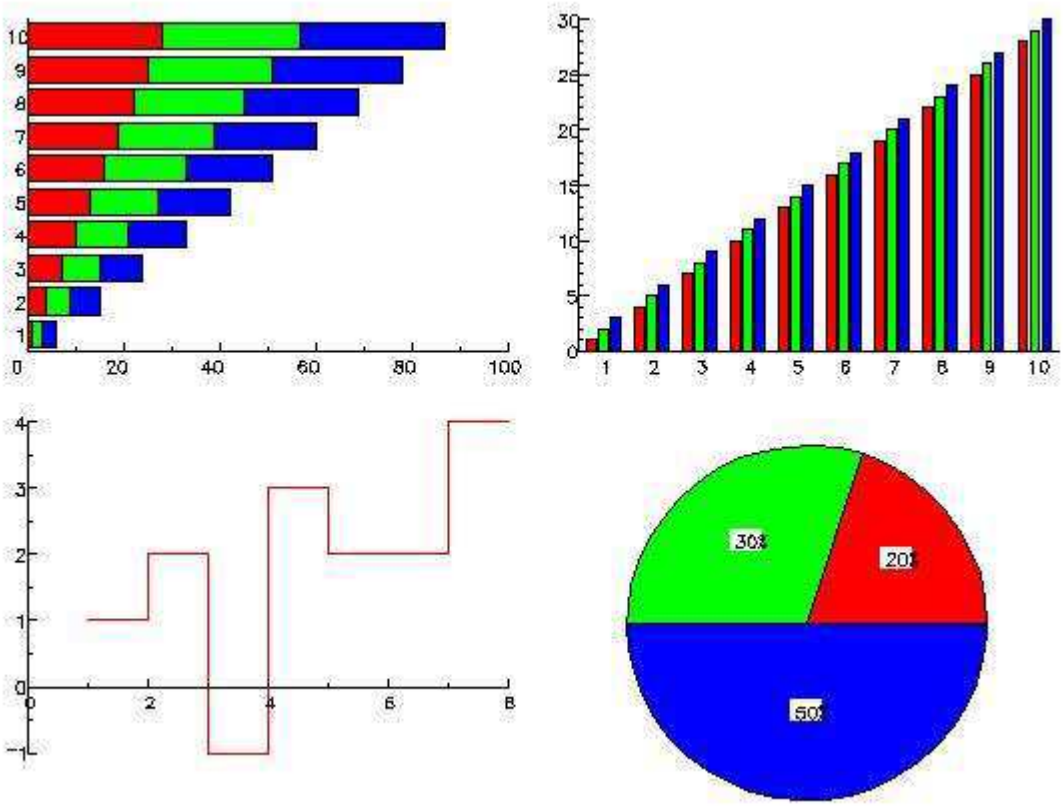
Some of the functions in GRAPHPAK use fairly sophisticated mathematical methods, like least-squares fitting. In order to use these methods it is not always necessary to understand them. This manual does not attempt to give the detail that would be needed for that kind of understanding.

Instead, the manual assumes that, if you want to fit a function to a set of points, you are already familiar with the method and know why you want to use it. If the manual mentions a method you don't know about, and doesn't explain it, then it probably doesn't do what you want. In that case, you can generally skip the entire section it appears in without missing anything you will need for other sections.

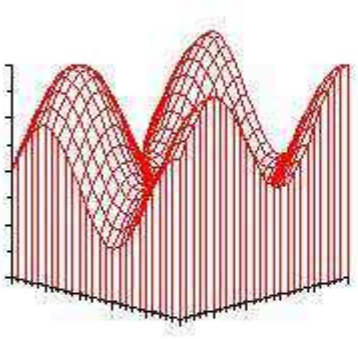
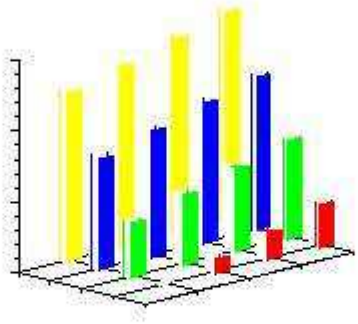
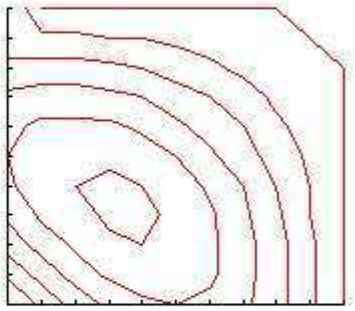
How This Manual is Organized

Getting Started and Drawing Line Graphs describes concepts and methods that are used in all the later sections.

Drawing Bar Graphs, Step Charts, and Pie Charts tells how to produce displays like those shown below.



Representing Surfaces shows how to draw representations of surfaces like those below.

Fitting Curves tells how to calculate and plot the best fit to a set of points. The functions fitted can have these forms:

- Average
- Straight line
- Polynomial
- Exponential
- Power
- Spline-like

Writing Text and Drawing Flat Pictures tells how to

- Write text anywhere on the screen
- Draw hierarchic diagrams (organization charts)
- Draw arbitrary lines and fill areas

Drawing Three-dimensional Objects tells how to draw and transform projections of 3-dimensional objects.

Function Reference gives an alphabetic list of the functions that you would likely use directly. It describes the syntax and results of each. (Functions that are typically used by other functions, and not directly by you, are not listed.)

Variable Reference gives an alphabetic list of the variables in GRAPHPAK that you may need to change explicitly. It summarizes their content and meaning. (Variables that you will not likely use directly are not listed.)

Installation and Customization tells how to:

- Do some things you have to do once, to install GRAPHPAK
- Remove comments and symbols from a private copy of GRAPHPAK, to save space
- Change the area of your screen that is given over to the APL2 session manager

APL2 Group Variables lists the group variables and shows which functions call which others. You may have to refer to this section if you copy or delete a portion of GRAPHPAK.

Device Dependencies lists the functions and variables that contain information about your output device. GRAPHPAK is supplied with all the different versions of APL2 for OS/2, AIX, Sun Solaris , Linux, Windows, DOS, and the mainframes. If you intend to use GRAPHPAK in applications that will be used on multiple platforms or want to take advantage of features specific to a particular platform, you may have to change some of these items.

Messages lists and explains the messages you might receive from GRAPHPAK, in order by message number.

There is also a glossary.

APL2 Documentation

Along with this manual, *APL2 GRAPHPAK: User's Guide and Reference*, the following additional on-line manuals are included with Workstation APL2:

- *APL2 User's Guide*
- *APL2 Language Summary*
- *APL2 Programming: Language Reference*
- *APL2 Programming: Developing GUI Applications* (for Windows only)
- *APL2 Programming: Using SQL*
- *APL2 Programming: Using APL2 with WebSphere*

The following table shows all the books in the APL2 library, organized by the tasks for which they are used. The APL2 library can be found on the web at *http://www.ibm.com/software/awdtools/apl/library.html*.

Information	**Book**	**Publication Number**
Introductory language material	An Introduction to APL2 APL2 Language Summary	SH21-1073 SX26-3851
Common reference material	APL2 Programming: Language Reference APL2 Reference Summary APL2 GRAPHPAK: User's Guide and Reference APL2 Programming: Using SQL APL2 Migration Guide	SH21-1061 SX26-3999 SH21-1074 SH21-1057 SH21-1069
Mainframe programming	APL2 Programming: System Services Reference APL2 Programming: Using the Supplied Routines APL2 Programming: Processor Interface Reference APL2 Installation and Customization under CMS APL2 Installation and Customization under TSO APL2 Messages and Codes APL2 Diagnosis	SH21-1054 SH21-1056 SH21-1058 SH21-1062 SH21-1055 SH21-1059 LY27-9601
Workstation programming	APL2 User's Guide APL2 Programming: Developing GUI Applications APL2 Programming: Using APL2 with WebSphere	SC18-7021 SC18-7383 SC18-9442

Related Publications

On APL2 mainframe systems, the graphic services used by GRAPHPAK are provided by the GDDM (Graphical Data Display Manager) product.

You must have GDDM to communicate between GRAPHPAK and a display terminal.

GDDM Publications:

- GDDM General Information, GC33-0319
- GDDM: Programming Reference, SC33-0332
- GDDM Manager Presentation Graphics Feature: Programming Reference, SC33-0333
- GDDM Application Programming Guide, SC33-0337
- GDDM Messages, SC33-0325

Syntax Notation

This manual uses a syntax notation that distinguishes between information you must enter exactly as shown and information that you may change as appropriate.

If an entry is to be made exactly as shown, it appears in `APL FONT`.

If an entry represents information that you may change according to your circumstances, it appears in *lowercase italics*. For example, a model entry might be described as

`DRAW` *right*

To use this model, you would enter `DRAW` (the name of an APL2 function) exactly as shown, and supply some other appropriate information (actual data, the name of an APL2 variable) in place of *right*.

Getting Started and Drawing Line Graphs

The purpose of GRAPHPAK is to help you create graphs, charts, and drawings on a computer display screen. The purpose of this document is to tell you how to use GRAPHPAK.

Perhaps you have already looked through this document to look at the pictures. That's a good way to start. With a little luck, one of the pictures looks like something you want to create. If so, you know exactly where you want to go. This chapter will try to get you there quickly.

If you found a likely picture, you probably found near it a set of APL statements that you thought might create it. If you tried to modify those statements to produce exactly what you wanted, you were probably frustrated. Even if you tried to execute just what appears in the manual, you might not have been pleased with the result. A few explanations might be needed. The next sections should help:

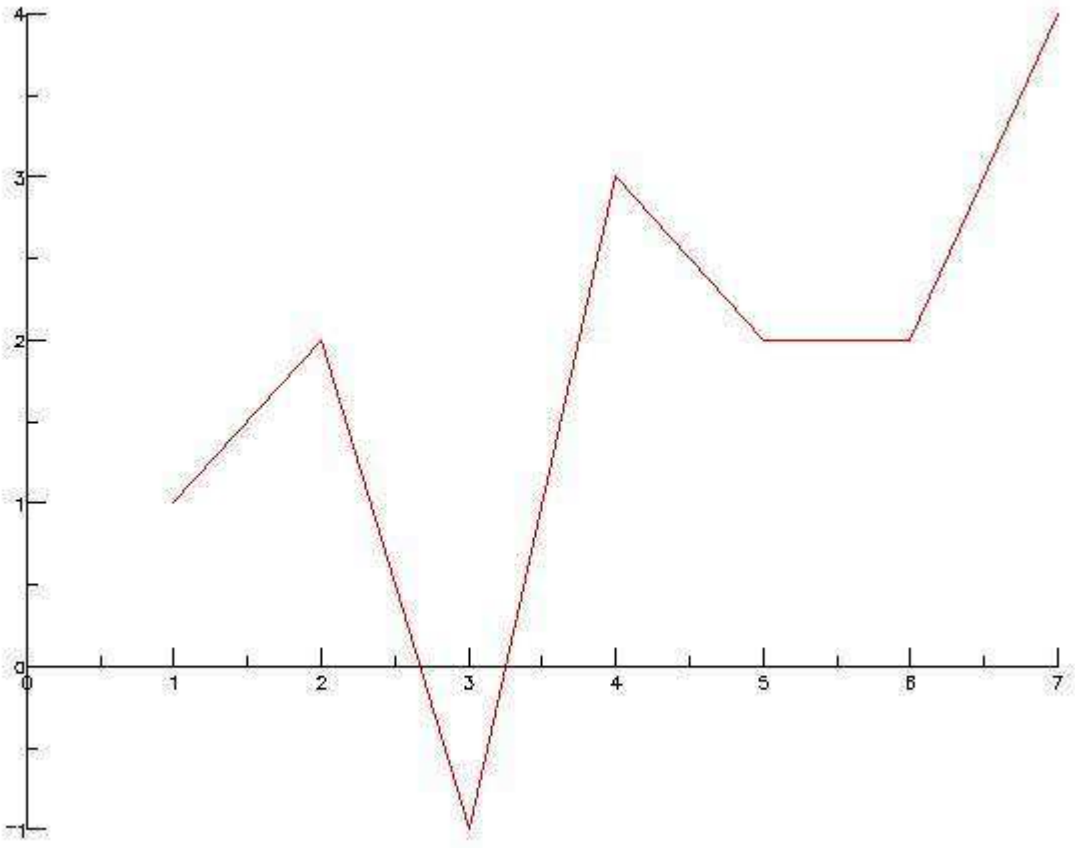
A Quick Example

But explanations can be tedious. Would you like to do something right away? If you have all the hardware and software installed, know how to sign on to APL, and know where to find the GRAPHPAK workspace, try this:

1. Sign on to APL and turn on the APL character set.
2. Load GRAPHPAK.
3. Enter these statements:

```
YDATA←1 2 ¯1 3 2 2 4
PLOT YDATA
VIEW
```

There is a good chance that your screen will display the following figure. If it did, and if these steps all made sense to you, you can skip the next section Getting Started and Drawing Line Graphs. Go on to A Model Function: PLOT Draws a Line Graph.



Getting Started and Drawing Line Graphs

This section explains the steps in the quick example.

When you sit down at a computer to use GRAPHPAK, you typically do something like this:

1. Sign on to APL2.
2. Load the GRAPHPAK workspace.
3. Copy data into the GRAPHPAK workspace.
4. Process the data with GRAPHPAK functions.

"Getting Started" includes the first three of these steps. Most of the rest of this manual tells how to do step 4 for different kinds of tasks.

Signing On

This step is necessary, but outside the scope of this manual. Instructions for signing on to APL2 are in the *APL2 User's Guide* for workstation systems or in *APL2 Programming: System Services Reference* for mainframe systems. Variations on the process may have been set up for your installation.

There are also options you can name when signing on. One of them allows you to use, or suppress, the APL2 session manager. The figures in this manual have been made using the session manager. On CMS and TSO, the session manager is not required.

Loading GRAPHPAK

When you are signed on to APL2 you get a series of messages, possibly ending with:

```
CLEAR WS
```

You can then load the GRAPHPAK workspace by giving the command:

```
)LOAD 2 GRAPHPAK
```

Here 2 is (probably) the number of the APL2 library that contains GRAPHPAK. When GRAPHPAK is loaded you'll get another message, containing an IBM copyright notice.

GRAPHPAK is an APL2 workspace like any other. You can add things to it, delete things from it, and store copies of it, even under other names. If you have a copy of the workspace in your private library, you don't need a library number in the)LOAD command. And if you have a copy stored under some other name, you can use that name instead of "GRAPHPAK".

Messages

Even the quick example can involve you in a series of messages issued by the programs you are using. You will receive the text of a message without a message number. If you wish to see the message number, issue the APL2 command MORE. Most of them, like these, are not disasters.

- If you are running on the mainframe, and you got the message,

  ```
  SYMBOL SETS DO NOT EXIST ON AUXILIARY STORAGE
  ```

  then some data sets, called "the symbol sets," were not loaded onto auxiliary storage when GRAPHPAK was installed. You can go ahead without them, but the system may not work as quickly. See What to Do with Symbol Sets to learn what the symbol sets are and what you can do about them.

- If you got the messages,

  ```
  GDDM RETURN CODE WARNING
  W CO-ORDINATE OUTSIDE PICTURE SPACE
  ```

  then the size of your session manager field, or the size of your graphic field, or both, are not what the examples in this manual used. See Changing the Session Manager and Graphic Fields to learn what these fields are and how you can change them.

- If you got any other GRAPHPAK message, look it up in Messages. There you will find a further explanation of what caused the message.
- If you got an APL2 error or trouble report, you probably spelled something wrong. This manual does not discuss APL2 errors. You can look them up in *APL2 Programming: Language Reference*.

Copying Data into GRAPHPAK

It is likely, but not necessary, that you will want to use data that exists in some other APL2 workspace. For example, to produce a simple graph like that shown in A Quick Example you need a set of Y-values, probably assigned to some variable - `YDATA` in the example. To plot `YDATA`, it must be in GRAPHPAK. If it exists in some other workspace, you can copy it into GRAPHPAK by giving the system command

```
)COPY wsid YDATA
```

where `wsid` is the name of the other workspace.

Other Methods

There are other ways to get data into GRAPHPAK. For example, you could

- Read the data from a file that is not an APL2 workspace. You can either associate a variable name with the file using associated processor 12 or read the file using a processor 10 function or an auxiliary processor. For information about using associated processors 10 and 12 and the auxiliary processors, consult the *APL2 User's Guide* for your workstation system or *APL2 Programming: System Services Reference* for the mainframe.
- Generate the data within GRAPHPAK by executing an APL2 function that you have defined within GRAPHPAK. This was done to produce several of the figures in this manual, for example Drawing Surface Charts.
- Enter the data into GRAPHPAK from the computer keyboard, as you did in the example.

If you are going to copy group variables under APL2, refer to APL2 Group Variables, and also to *APL2 Programming: Language Reference*.

A full discussion of any of these methods is outside the scope of this manual. In most of what follows, it is assumed that you have the data you need within GRAPHPAK.

Drawing Pictures

Drawing pictures is a special case. There are functions in GRAPHPAK that allow you to use your cursor or mouse to draw pictures. These functions assign data about the location of the cursor or mouse to variables in GRAPHPAK. For information, see the options of `READ` described in Writing Text and Drawing Flat Pictures.

A Model Function: PLOT Draws a Line Graph

Now you are through getting started. Pause here to think about how you'll be using GRAPHPAK.

The simplest model for using GRAPHPAK is this: You provide the data (in `YDATA` in the example), and find a function (`PLOT`) that produces the display you want. Combine the function and the data in a statement like

```
PLOT YDATA
```

What to Expect Next

The rest of this chapter looks at a sample task: plotting a line through a set of points. The sample serves two purposes:

- It introduces two functions, `PLOT` and `SPLOT`, that are both typical and simple.
- It illustrates concepts and methods that will be needed later, under the headings of
  - Basic Housekeeping Routines
  - Structuring Data for PLOT
  - Defining Spaces and Viewports
  - Putting on the Finishing Touches

A final example combines these concepts and methods into a new APL2 function like one you might want to create for yourself. This example is found in the section titled A Note on APL2 Functions.

A Bit of a Warning: The descriptions of functions given in the rest of this chapter are not necessarily complete. The same is true of descriptions in Chapters 2 through 6. But there are complete descriptions of the functions you need, in alphabetic order, in Function Reference, and complete descriptions of the global variables in Variable Reference.

The reason for omitting some of the details is to make some other things stand out more clearly. The first sections of this manual give you the big picture; if they don't tell you how to do something you want to do, try the later reference sections.

Basic Housekeeping Routines

These are the things about which people ask, "Why didn't the computer know I wanted to do that?" But no, every little thing needs a function to do it. This section describes

- VIEW
- COPY and COPYN
- FSSAVE and FSSHOW
- GSSAVE GSLOAD and PRINTFILE
- ERASE
- RESTORE

VIEW

Depending on how you use the APL2 session manager, you may have to use VIEW (It was used in the statements that produced A Quick Example, just in case.)

On Workstation Systems:

GRAPHPAK and the session manager display their output in separate windows. Normally, you can use window manager techniques to switch between windows. However, if the GRAPHPAK window is lost behind other windows or you want to bring it to the foreground under program control, you can use the VIEW function. VIEW will bring the GRAPHPAK window to the foreground and wait for user input.

On CMS and TSO:

With the APL2 Session Manager:

You have a session manager field and a graphic field available.

- On the session manager field, you enter APL2 statements and display character and numeric output.
- On the graphic field, you display graphic output.

If the two fields do not overlap, they appear together on your screen. If they do overlap, only one will be in the foreground at any given time.

- If the session manager field is in the foreground, use VIEW to switch to the graphic field to see graphic output. Press the Enter key to return to the session manager field.
- If they appear together, you don't need VIEW. Graphic output will appear in the graphic field as soon as it is created.

The figures in this manual were made using the session manager. If you are trying to execute the examples exactly, and think you have gotten out of step somewhere, see Changing the Session Manager and Graphic Fields to learn how to reset the session manager control.

Without the APL2 Session Manager:

You have an APL screen and a graphic field. The graphic field is under control of the graphics processor. Use VIEW to switch to the graphic field so you can see graphic output. Press the Enter key to return to the APL screen.

Remember VIEW: It's left out of the examples from now on.

COPY and COPYN

To produce a printed copy of graphic output, do one of these procedures:

```
copyname←destination
COPY
```

or

```
COPYN destination
```

The function COPY finds the print destination in the global variable copyname. The function COPYN simply assigns its right argument to copyname and executes COPY.

If copyname begins with a blank or asterisk, COPY will prompt you for a destination.

On OS/2 and Windows:

destination is either a file specification of the form:

```
[path]filename
```

or a null character vector. If a file specification is supplied, a metafile is written. If a null character vector is supplied, the default printer is used.

On CMS and TSO:

COPY and COPYN produce output on the alternate GDDM device.

COPY and COPYN use the global variable copyctl which controls the number of copies, page depth, and margin size.

On TSO, destination is the address of the printer.

On CMS, destination is a character vector of up to 8 characters that names a file which will be created and stored on your CMS disk. destination is used as the file name and ADMPRINT is the file type. After you have finished your APL2 session and signed off APL2, you can print the file using the print utility ADMOPUV:

```
ADMOPUV filename
```

The printer must be connected to virtual address 061.

GDDM allows ADMQPOST EXEC to process print files and nicknames for automatic processing. See *Graphical Data Display Manager Base: Application Programming Guide* for further information on the use of GDDM on CMS and TSO.

On Unix Systems:

`COPY` and `COPYN` are not supported on these operating systems.

FSSAVE and FSSHOW

Note: `FSSAVE` and `FSSHOW` are only supported on CMS and TSO.

To save a copy of graphic output in a file, execute

```
FSSAVE filename
```

where *filename* is a file name of your choice.

To read it back and display it again, execute

```
FSSHOW filename
```

where *filename* is the same name you used before.

There's little need for this operation with simple examples: To see the plot of `YDATA` again in a later session, it's easiest to save `YDATA` in your workspace and execute `PLOT YDATA` all over again. You might well want to use `FSSAVE` and `FSSHOW` with a complex example that takes many statements to produce.

GSSAVE, GSLOAD, and PRINTFILE

Note: `GSSAVE`, `GSLOAD`, and `PRINTFILE` are only supported on CMS and TSO.

To save a copy of graphic output in an ADMGDF file, execute

```
GSSAVE filename
```

where *filename* is a file name of your choice.

To load the output back onto the graphics screen, execute

```
GSLOAD filename
```

where *filename* is the same name you used before. The `VIEW` function may be needed to cause the graphics screen to be displayed.

ADMGDF files are more flexible than the files produced by FSSAVE. FSSAVE produces ADMSAVE files which can be processed very quickly. However, they are device dependent. This means that they can only be displayed on the same type of terminal on which they were created.

ADMGDF files, however, are device independent. They can be displayed on many kinds of terminals and can also be converted for use on other types of devices, for example printers and plotters.

To save a copy of graphic output in a GDDM print file, execute

```
PRINTFILE filename
```

where *filename* is a file name of your choice. GDDM print files cannot be read using GRAPHPAK

GDDM print files are device dependent and are used when printing graphics on 3800 family printers.

ERASE

Suppose you have just executed

```
PLOT YDATA
```

and you now execute

```
ZDATA←2 3 3 2 4 4 4
PLOT ZDATA
```

When you look at the result, you will see the two graphs superimposed. In some cases, this is what you want to happen. But to make the second example completely separate from the first, execute

```
ERASE
PLOT ZDATA
```

RESTORE

Some of the things you do in GRAPHPAK change the values of global variables that are used as controls. Often you want to change those values back again before doing something else. RESTORE restores many of them; execute

```
RESTORE
```

RESTORE will be mentioned again often, in connection with each variable it modifies.

Structuring Data for PLOT

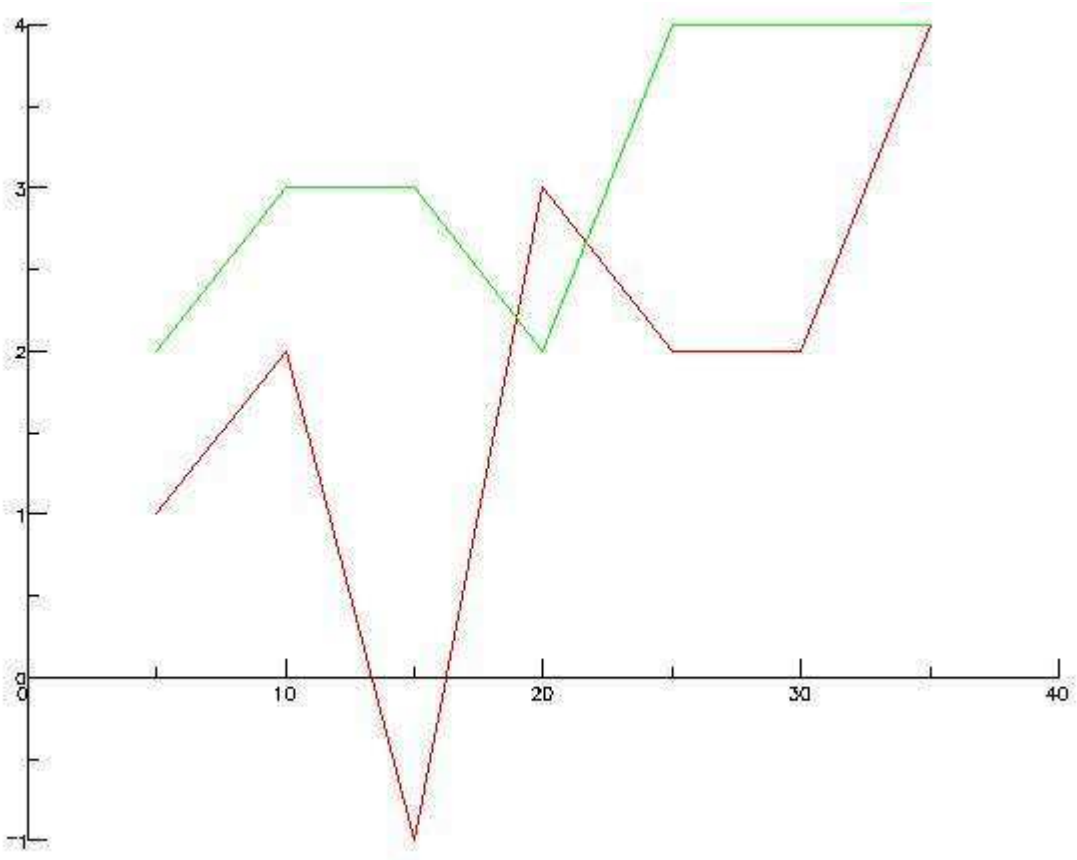
A Quick Example shows that PLOT will take an argument that is a vector of n Y-values, and plot them versus integral values of X from 1 to n. It will also take a 2-column matrix, containing X-values in its first column and corresponding Y-values in its second column. But there are other possibilities also.

The following figure shows a graph of YDATA and ZDATA versus 7 values of X from 5 to 35. It was created by the following statements:

```
X←5×⍳7
PLOT YDATA AND ZDATA VS X
```

The purpose of AND and VS (versus) is to structure data into the form required by PLOT.

Note: This example, and others throughout the manual, assumes that the APL2 index origin system variable (⎕IO) is set to 1, as it is when GRAPHPAK is distributed.



Plotting Several Graphs at Once with AND and VS

The names AND, VS, and PLOT have been chosen so that you can write a command that looks like English and have it mean to APL2 just about what it means to you. (But the use of parentheses follows algebraic notation,

not English.) The actual data structure that is produced is probably not important to you, though it is described in the reference section in the appendix.

How you write a command depends on the form of your data, as these examples show:

Example - Data in Vector Form

You have several pairs of vectors, say XA and YA, XB and YB, XC and YC, containing the X- and Y-values of several sets of points. To plot the graphs of all the sets on the same pair of axes, execute

```
PLOT (YA VS XA) AND (YB VS XB) AND (YC VS XC)
```

Other combinations of the data are possible. For example,

```
PLOT (YA VS XA) AND (YB AND YC VS XB)
```

Example - Data in Matrix Form

You have the Y-values of several sets of points in successive columns of the matrix YMAT, and the corresponding X-values in the corresponding columns of matrix XMAT. To plot the graphs of all the sets on the same pair of axes, execute

```
PLOT YMAT VS XMAT
```

Defining Spaces and Viewports

If you do the examples shown so far in this chapter, without changing anything else in GRAPHPAK, the graphs will take up most of your graphics window. You may want to make them a little larger, make them smaller, or put several graphs in the window side by side. To make these changes, you need to know how information about your window is kept in GRAPHPAK. This section describes the basic concepts you will need.

Virtual Space

Regardless of the physical size of any particular output device, `PLOT` prepares output for a 2-dimensional space of 100 units by 100 units. Each point in this space is known by its coordinates: the point (0,0) is at the lower left of the screen; the point (100,100) is usually off the screen at the upper right. The entire set of points is called *virtual space*.

When you use GRAPHPAK functions that draw graphs or charts, think of them plotting their output on virtual space. There are other functions (`WRITE`, `DRAW`, and `FILL`) that allow you to address virtual space directly.

Real Space

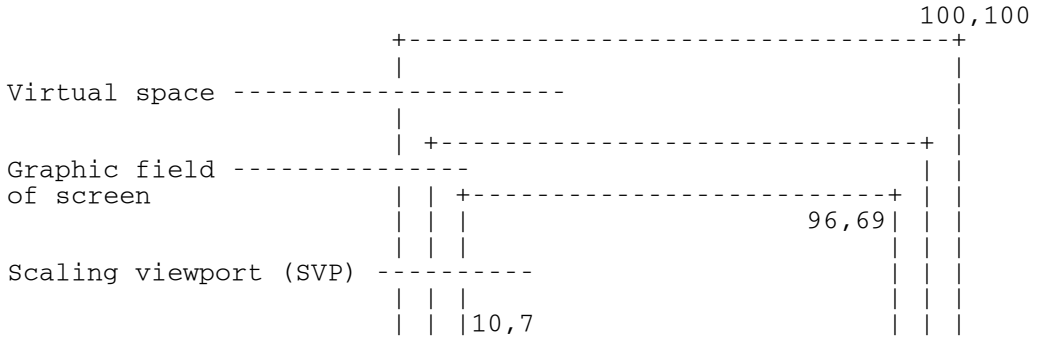
The set of real points on a cathode-ray tube at which an electron beam can be physically aimed, or the set of points on a piece of paper on which a distinct dot can be printed or at which a plotting arm can be positioned, is a *real space*. The number of distinct points in real space may differ for each type of device. Hence GRAPHPAK must keep information about the size of real space, and include functions that transform output for virtual space into output for real space. The information is kept in the global variable `dd`.

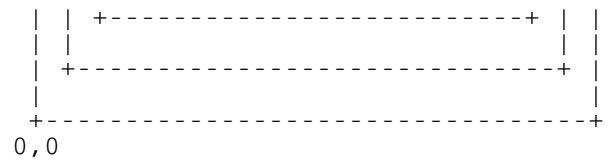
The Scaling Viewport

Overview

`PLOT` prepares output for virtual space, but usually not to fill all of virtual space. The typical display screen is already smaller than virtual space, and to allow room for titles and labels the typical graph should be smaller still. So `PLOT` (like most of the GRAPHPAK functions) scales down its output to fit a subset of virtual space; this subset is called the scaling viewport.

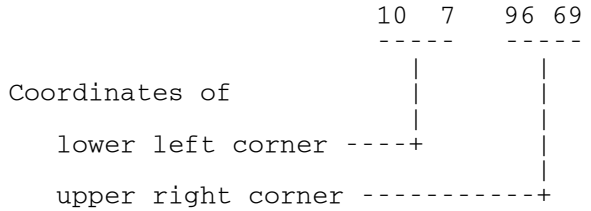
The following figure shows a possible relationship of the scaling viewport to virtual space and to your display screen.





Definition

The *scaling viewport* is a rectangle within virtual space that has been allotted for the output of plotting and charting functions. It is described by the global variable `svp`, a vector of length 4, as shown below. The numbers there are coordinates in virtual space.



Example

The figure uses the default value of `svp`, which defines a scaling viewport that fits comfortably into the display screen of the IBM 3279 model 3 terminal; it is 86 units wide (96 minus 10) and 62 units high (69 minus 7). The comfortable fit allows room for labels that fall outside the scaling viewport on the left and bottom of the screen. If your data is such that labels will fall on the right and/or top, you may need to adjust `svp` to avoid truncation of the labels.

Changing It

To change the size of a graph, or to relocate it in virtual space, change the scaling viewport. For example, to put the graph of `YDATA` above the graph of `ZDATA`, execute the following statements:

```
svp←10 8 90 32
PLOT ZDATA VS X
svp←10 36 90 60
PLOT YDATA VS X
```

In the example, the first assignment to `svp` makes the scaling viewport 80 units wide and 24 units high, with its lower left corner at (10,8) in virtual space. The graph of `ZDATA` is plotted in that rectangle. The second assignment defines a new scaling viewport, of the same size, with its lower left corner at (10,36); the graph of `YDATA` is plotted there.

Limits on It

Before defining a new scaling viewport, consider the size of your graphic field. You don't want to put part of the viewport outside the graphic field. The size is given in the global variable `dd`. The lower left corner of your screen is at (0,0) in virtual space; the coordinates of the upper right corner are in the first two elements of `dd`. (For the IBM 3279 model 3 terminal, and with the settings of the session manager that this manual uses, the upper right corner is at (100, 71.2).)

Restoring It

If you had just executed the previous example, your scaling viewport would now be set to 10 36 90 60. Before creating a new graph, you might want to set the scaling viewport back to its initial value. `RESTORE` resets it to a convenient size.

The Clipping Viewport

If you're tired of spaces and viewports, skip to the next heading. You can do everything in the manual without knowing about this.

GRAPHPAK contains the description of another viewport, the *clipping viewport*, that also helps to define what your output will look like. Functions that produce graphic output normally discard anything that would lie beyond the boundaries of the clipping viewport. Hence you can think of the clipping viewport as a device that cuts away unwanted parts of your output.

Typically, the clipping viewport is set to coincide with the boundaries of your graphic field. In that setting it prevents any attempt to produce a line beyond the edge of the field. However, you can also use the clipping viewport to define a new viewing area, of almost any shape you like, within the boundaries of your actual screen. Any output you try to produce can be cut off at the boundaries you establish for the clipping viewport.

The description of the clipping viewport is in the global variable `cvp`. The function `FIXVP` puts a description of a shape you define into the form required by `cvp`. (Use the function `VIEWPORT` to display and `FIXVP` to set `cvp`.)

What happens to lines that might extend outside the clipping viewport can be modified. The result depends on the value of the global variable `sci`.

`RESTORE` restores `cvp` to a description of the screen boundaries.

The Window in Problem Space

Don't skip this section: it might be just what you need.

Definitions

A set of points to be plotted defines a *problem space*, which is the set of all possible points with the same type of coordinates. For example, if you are plotting revenues by year, your problem space consists of all possible points with an X-coordinate that represents a year and a Y-coordinate that represents an amount of revenue.

In practice, you use a subset of your problem space - say only years from 1985 to 1995 and dollar amounts from 0 to $1,000,000. This subset is the *window* in your problem space.

Example

A Quick Example shows that `PLOT` will calculate a window in your problem space, based on the values you give it to plot. (The window in that figure extends from 0 to 7 along the X-axis and from -1 to 4 along the Y-axis.) But you can specify a different window. For example, if you are plotting percentages, you may want to

specify that the window extends upward to 100%. Or you may want the area of your graph to extend outward to 2010 and upward to $2,000,000 even though none of your points reaches either limit.

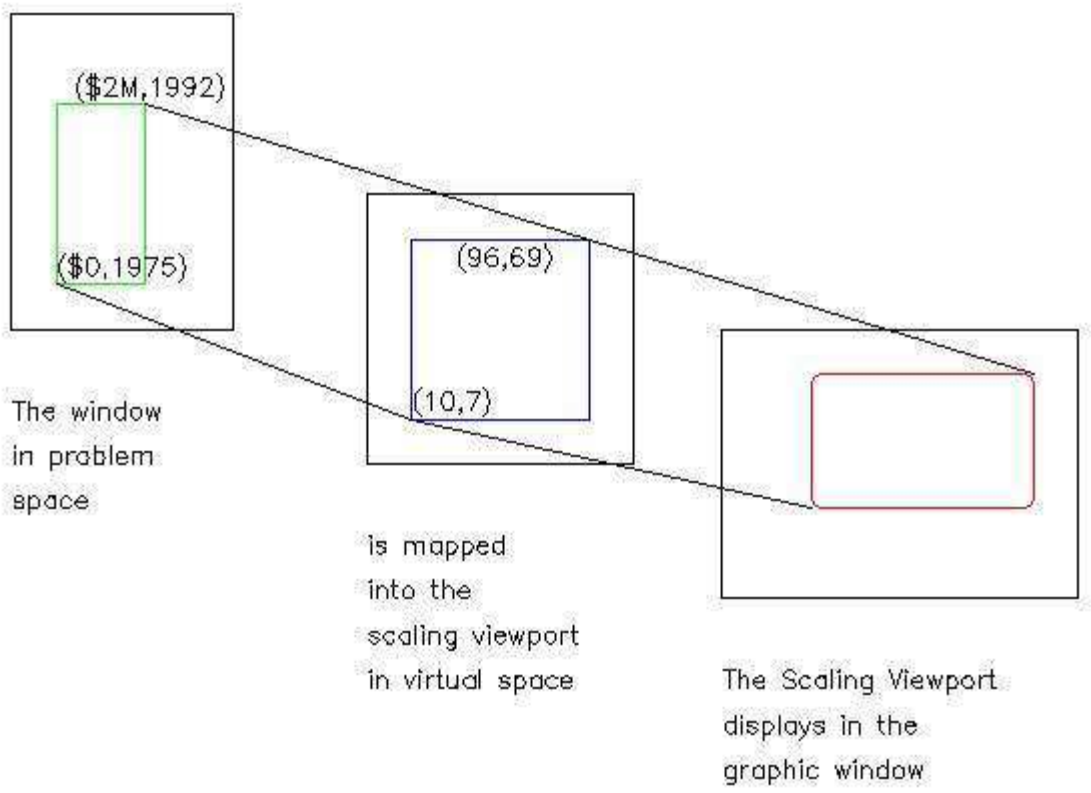
Changing It

To change the window in your problem space, assign its coordinates to the global variable `w`. The format of `w` is the same as the format of `svp`: the coordinates of the lower left corner followed by the coordinates of the upper right corner.

```
                              0   0  50 50
                              ---  ---
Coordinates of                  |    |
                                |    |
     lower left corner ---------+    |
                                     |
     upper right corner --------------+
```

Relationship to the Scaling Viewport and Virtual Space

The relationship between the window in problem space and the window in virtual space (the scaling viewport) is likely to become quite important to you. The figure below suggests a way to think of it: The plotting and charting functions in GRAPHPAK map a window in problem space into the scaling viewport. If you don't define a window explicitly, they will derive one from the data you supply.



Using Your Window

To make use of a window you have set yourself, use the function SPLOT (specified plot). For example, to plot the points of A Quick Example in a window extending from 0 to 10 on the X-axis, and as far below the X-axis as it does above, execute

```
w←0 ¯4 10 4
'S' SPLOT YDATA
```

The function SPLOT uses a left argument to specify variations in the display. This argument is a character vector whose elements can appear in any order; each controls a different aspect of the result.

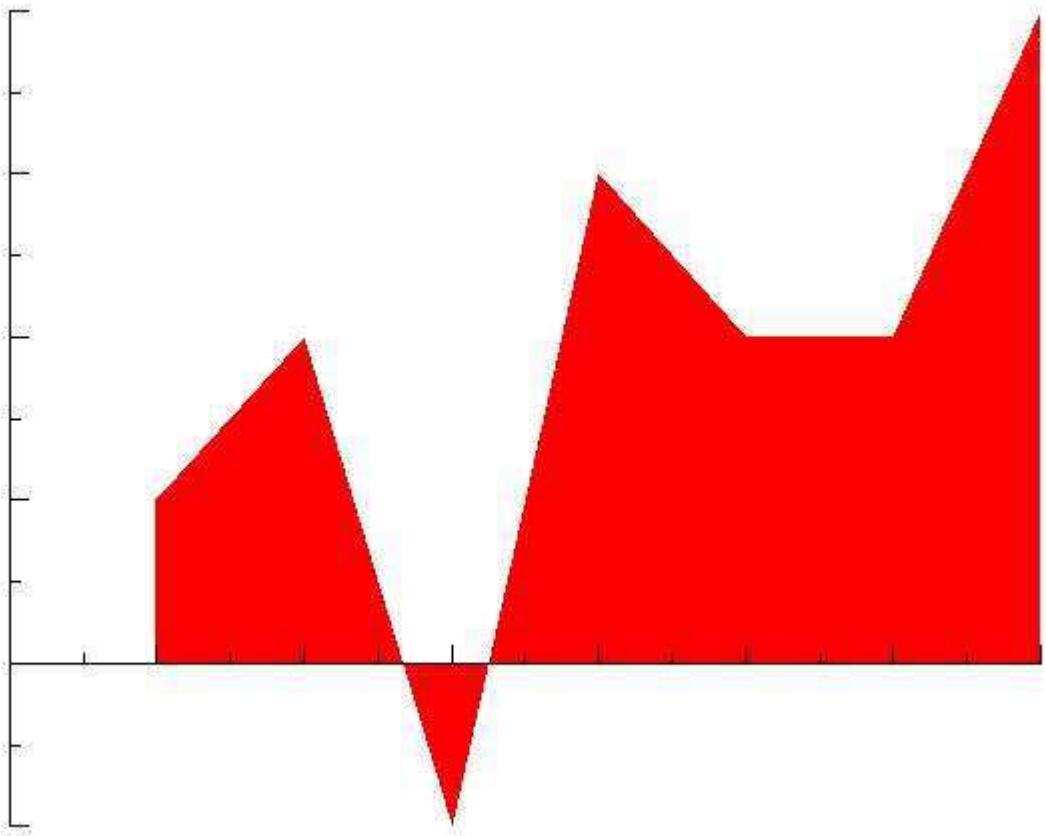
The character S, used here, tells SPLOT to suppress its automatic computation of a new window w from the problem data and instead to use the w you have specified. (Using 'S', you need not specify all four elements of w yourself. You may, for example, set only the lower limits and leave SPLOT to complete w by computing the remaining values. However, when specifying the lower X element as a single element, it must be preceded by a comma.) Other control characters you can use are described in the next section.

Putting on the Finishing Touches

The figures that have appeared so far in this chapter are pretty bare-bones affairs, not quite what you would want to present to a board of directors or an international conference. This section tells how you can

- Create axes, labels, annotations, and titles.
- Specify color, style, width, and mode.
- Fill and edge areas.
- Specify other aspects of line graphs.

So they will look like this...



Creating Axes, Labels, Annotations, and Titles

The plotting and charting functions in GRAPHPAK will generally draw axes for you by default, if you want. But you can suppress these, and create new or additional axes if you like.

You can also specify three different types of identifiers for parts of a graph.

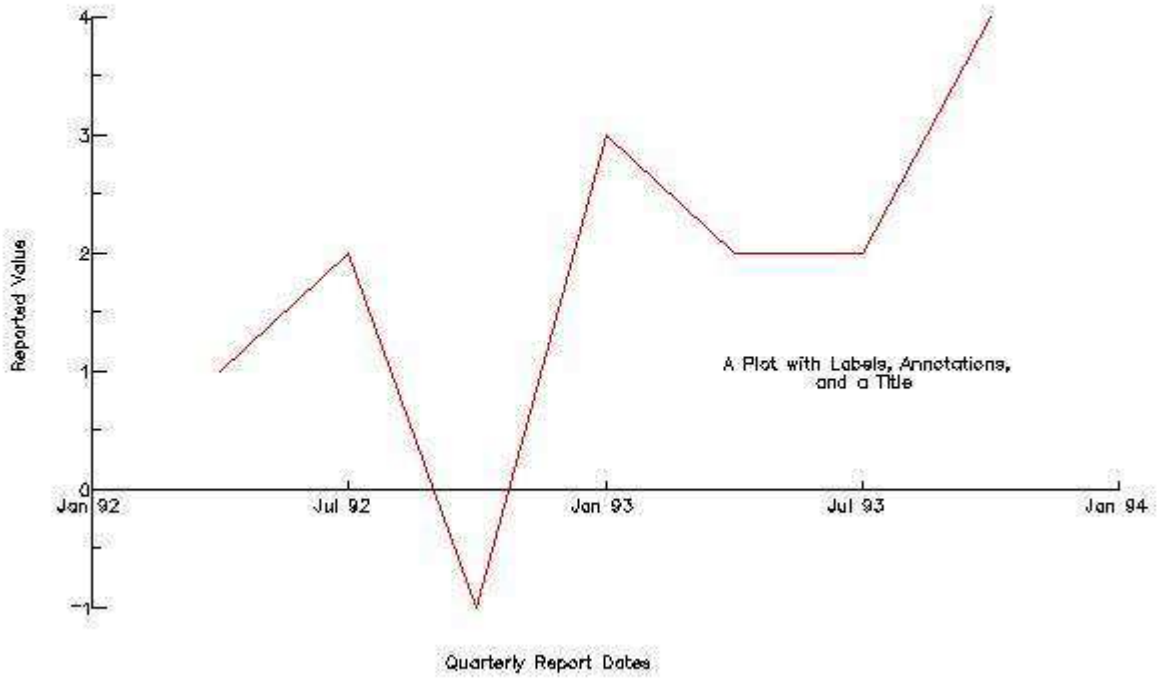
Labels
: identify the tick marks on axes. (The numbers 10, 20, 30 in Structuring Data for PLOT are labels.)

Annotations

identify an entire axis. (For example, you could identify the Y-axis with the annotation REPORTED VALUE, as in the following figure.)

Titles

identify the entire figure. (The graph below bears the title, "A Plot with Labels, Annotations, and a Title.")



The following statements were used to produce the figure:

```
YDATA←1 2 ¯1 3 2 2 4
svp←10 10 85 60
'A' SPLOT YDATA
0 2 4 6 8 AXIS 0
0 2 4 6 8 LBLX 5 6⍴'Jan 92Jul 92Jan 93Jul 93Jan 94'
¯1 0 1 2 3 4 LBLY 0
¯0.1 ANNX 'Quarterly Report Dates'
¯0.25 ANNY 'Reported Value'
6 1 TITLE 'A Plot with Labels, Annotations,¨          and a Title'
```

What's Coming

This sequence of statements includes the following items:

1. The function `SPLOT`, introduced earlier.
2. A character that tells `SPLOT` not to use the default method of locating tick marks on the axes and labeling them.

3. New functions to specify the axes, labels, annotations, and titles you prefer.

This section describes the new items.

Using Control Characters in Left Arguments

Quite a few of the functions in GRAPHPAK use characters and digits in a left arguments as controls. You have already seen the use of `S` in the left argument of `SPLOT`. Some other control characters are coming up soon. Many more appear in later chapters.

General Rules for Control Characters

- No two functions allow exactly the same list of control characters. When in doubt as to what a function allows, check its description in Function Reference.
- Even if you want none of the options allowed by the control characters, you can't ignore the left argument if the function uses one. At the very least, you have to use an empty vector (`''`) as the argument.
- When two different functions allow some of the same control characters, those characters mean pretty much the same thing to both functions. (Several functions use `A`, `L`, and `S` as `SPLOT` does.)

Note: Lower Case and Underbarred Control Characters

In previous versions of GRAPHPAK, some functions supported underbarred control characters. All these functions now support lower case control characters rather than underbarred characters.

Because these characters are usually coded in literal character strings in functions, they are unaffected by the CASE invocation option setting. User functions which use underbarred control characters should be modified to use lower case control characters.

Control Characters for SPLOT

This is not a complete list, but only an introduction. By using control characters for `SPLOT` you can:

- Omit drawing default axes. (Use `A`.)
- Put default labels on the axes. (Use `L`.)

The figure above was plotted using `A`. The axes were drawn by a separate statement.

AXIS and AXES

Use these functions if you have used `A` in the left argument of `SPLOT`.

`AXIS` uses its left argument to control the X-axis, and its right argument to control the Y-axis. If an argument is a vector, `AXIS` will put tick marks on the corresponding axis at the numbers given in the vector. If the argument is 0, `AXIS` draws a default set of tick marks. The figure in Creating Axes, Labels, Annotations, and Titles was plotted using the statement

```
      0 2 4 6 8 AXIS 0
```

which put tick marks at 0, 2, 4, 6, and 8 on the X-axis and at default locations on the Y-axis.

AXES uses no arguments, and draws two default axes. It is equivalent to 0 AXIS 0.

With other values of the arguments for AXIS you can specify color, style, width, and mode for an axis and its tick marks independently; specify a length for the tick marks; or specify that there be no tick marks at all. You can also vary the location of an axis. (By default, the X-axis is placed to intersect the Y-axis as near as possible to Y=0. And similarly for the Y-axis.) And you could use AXIS more than once, to put several sets of axes on the same graph.

LBLX and LBLY

These functions put labels on the X- and Y-axes, respectively. Their left arguments give the locations at which labels are to be placed on an axis. (They could be the same vectors that were used as arguments for AXIS.)

The right argument for LBLX or LBLY is a numeric vector or a character vector or matrix. It contains the labels themselves, generally one to each row.

The label can include two APL characters for special purposes:

¨ signals the beginning of a new line of a label.

⌶ signals the beginning of a new label.

(You can change the characters used for these purposes; they are held in the global variable dlc.)

Creating Axes, Labels, Annotations, and Titles was plotted using the statements

```
      0 2 4 6 8 LBLX 5 6 ⍴ 'Jan 92Jul 92Jan 93Jul 93Jan 94'
      ¯1 0 1 2 3 4 LBLY 0
```

Here the 0 in the right argument of LBLY specifies that the labels are the numbers in the left argument. In this case, the left argument could have been 0 to indicate that data values will be used to label the axis.

A Warning About Space

When you first put labels, annotations, and titles on your graphs, it is possible that they will, in whole or in part, fall outside the viewport defined by svp. On the mainframe, you may see the message

```
GDDM RETURN CODE WARNING
W CO-ORDINATE OUTSIDE PICTURE SPACE
```

This generally means that you created a graph that filled the space available on your screen, and then tried to put labels or titles outside that. You can view the result and see how much is missing. Then

1. Erase the screen.
2. Reduce the size of your scaling viewport, by assigning new coordinates to svp.
3. Create your graph, with its labels and titles, again.

This was done in creating Creating Axes, Labels, Annotations, and Titles, for example.

You may also find that the characters used in labels and the like are larger than you expect. The size of the characters is determined by the last use of the function WRITE. To set them back to their default size, execute

```
0 0 1 WRITE ''
```

(More interesting uses of WRITE are described in Writing Text and Drawing Flat Pictures.)

ANNX, ANNY, HOR, and VER Write Annotations

ANNX and ANNY specify, respectively, annotations to be placed near the X- and Y-axes. The annotations in our graph were created by the statements

```
¯0.5 ANNX 'Quarterly Report Dates'
0 ANNY 'Reported Value'
```

ANNX and ANNY use both right and left arguments. The left argument is a distance of the annotation from its default position, given in problem-space units. (The displacement is vertical for ANNX and horizontal for ANNY.) The right argument contains the annotation.

The default positions for annotations are held in the global variables ofx and ofy. These variables are set by the functions LBLX and LBLY, respectively. So if you are using both LBLX and ANNX, use LBLX first. RESTORE restores the default values.

HOR and VER specify annotations for the horizontal and vertical axes, respectively, and produce default labels. They don't allow displacements, and take only right arguments. (Thus HOR *annotation* is equivalent to 0 ANNX *annotation* followed by 0 AXIS ''.)

TITLE Writes Titles

TITLE gives a title and its location on the graph. Its syntax is

```
P TITLE C
```

Here P is a numeric vector that controls the position and other characteristics of the title. Its first two elements give the coordinates (in problem space, not virtual space) of the center of the title. Using other elements, you may cause your coordinates to refer to an edge or corner point of the title, or you may position the title in relation to the scaling viewport, or put a box around the title and fill it with a pattern, or specify legends that describe the lines in the graph (as in A Note on APL2 Functions).

C is a character vector or matrix that contains the title, as in the functions LBLX or ANNX. It can also contain coding to specify legends that show the symbols, colors, or patterns used in plotting the graph.

The title on Creating Axes, Labels, Annotations, and Titles was added by the statement

```
6 1 Title 'A Plot with Labels, Annotations,¨          and a Title'
```

You can use TITLE more than once to put more than one title on a single plot. Also, you can use the function WRITE to write text in the graphic field along with a plot; see its description in Writing Text and Drawing Flat Pictures.

Specifying Color, Style, Width, and Mode

The next of the finishing touches on your output is perhaps an attractive color. But changing the colors of your output is not merely a matter of what you do in GRAPHPAK, but also of what your output device can do. The IBM 3279 display terminal can show eight colors. What happens when you ask for color, then, is device-dependent.

Similar remarks apply to style, width, and mode. The IBM 3279 can produce lines in seven styles (solid, dotted, dashed, and so on). It can produce lines in two widths. It can produce only one mode. Other devices have other capabilities; an additional mode, for example, might be a blinking line on some displays.

The discussion that follows describes ways to change colors, styles, widths, and modes. The operations described can be done on any device that supports GRAPHPAK. But bear in mind that the visible results may differ considerably from one device to another.

Attributes, Values, and Attribute Codes Defined

Color, style, width, and mode are *attributes*. Particular colors, styles, widths, and modes are represented by numbers, and these numbers are *values* of the corresponding attributes. A line is drawn, or a character is written, with a *set of attribute values* that gives one value for each attribute of the line.

A set of attribute values can be represented by a single number, called an *attribute code*. If this code is

negative
: it represents a set of attribute values coded in base 100 and combined into a single integer. There is a set of functions described below that construct negative attribute codes. You do not need to remember a formula for negative codes; just note that an unexpected negative number in a series of control codes is probably an attribute code. (You can find a description of the code in av.)

positive
: it is an index into the attribute vector (av), which contains negative codes. av is described below.

Specifying Attribute Codes

The functions COLOR, STYLE, WIDTH, and MODE build negative attribute codes. To assign to the variable CODE the attribute code for color 3, style 4, and width 2, execute

```
CODE ← COLOR 3, STYLE 4, WIDTH 2
```

In this example, it wasn't necessary to specify MODE: if one of the attributes is omitted, a default value is used.

The functions can be used in combination, in any order. If they are used in combination, the arguments of each must be of the same length (except that one may have an argument of length one). The following examples show some of the possibilities.

```
CODE ← COLOR 1 2 3 4, STYLE 3
```

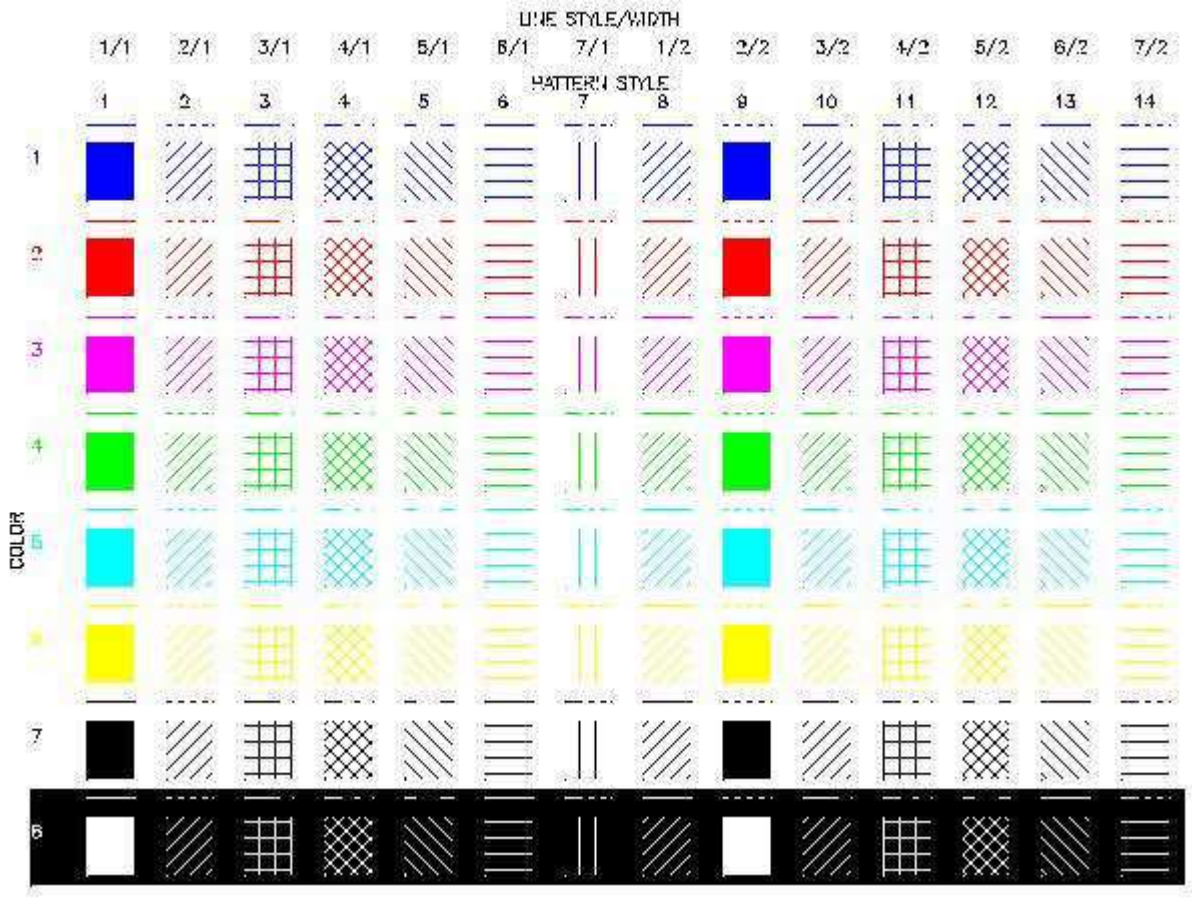
makes CODE a vector of four attribute codes. They specify colors 1 through 4, each combined with style 3.

```
CODE ← STYLE 1 2 3, COLOR 4 5 6
```

assigns three attribute codes. They specify style 1 combined with color 4, style 2 combined with color 5, and style 3 combined with color 6.

What Does It Look Like

There is a group of demonstration programs in GRAPHPAK. One of them displays the results of various combinations of attribute values; ATTRIBUTES produces the following figure:



If you try out all the color values in turn, watch out for the neutral color; it may not show up against the background.

Using Attribute Codes

Every function that produces a display looks somewhere for an attribute code, or set of codes, to control it. To use an attribute code, find out where to put it.

For example, the function TITLE can create a border around your title. (One appears in Specifying Other Aspects of Line Graphs.) The third position of the left argument controls its attributes. Try, for example,

```
(6 1, (COLOR 2, STYLE 3)) TITLE 'A TITLE WITH A BORDER'
```

The Attribute Vector (av)

There is also a way to use attribute codes without building them yourself. RESTORE sets the global variable av, called the attribute vector, to a systematic variation through the set of all attribute codes that your output device will recognize. Any function that looks for an attribute code recognizes a positive number as an index into the attribute vector; it uses the code the index points to.

For example, execute

```
RESTORE
6 1 18 TITLE 'A TITLE WITH A BORDER'
```

If your output device allows 8 color specifications, then this produced the same result as the example before. The first eight positions of av coded all colors with style 1, the second eight positions coded all colors with style 2, and color 2 combined with style 3 appeared in position 18.

USE

Many functions - HOR, VER, AXIS, ANNX, LABX and so on - look to av for default values of attributes. It is probably better not to change the contents of av at this point, but later you may want to. There is a function that does it, called USE, described in Writing Text and Drawing Flat Pictures.

The Plot Attribute Vector (pa)

The place to put attribute codes for plotting is in the plot attribute vector (pa).

PLOT and SPLOT, and many other functions that turn up in Chapters 2 through 4, look in the plot attribute vector for attribute codes. What they find there can either be negative codes or indexes into the attribute vector. (RESTORE puts into pa a set of indexes into av.)

If PLOT or SPLOT is plotting more than one graph, the function looks repeatedly at elements of pa for further attribute codes. It will plot the first graph with the values specified by the first element, the second graph with the attributes specified by the second element, and so on. If there are more graphs than elements in pa, the search for values will wrap around to the first element again. (And if an index into av is larger than the number of elements in av, it also causes the search to wrap around to the first element of av again.)

Changing Colors for PLOT and SPLOT

Probably the simplest way to change colors, styles, and widths for plotting is to do something like this:

```
pa ← STYLE 1 2 1 2, COLOR 2 3 4 5
PLOT A AND B AND C AND D VS X
```

If the color and style combinations aren't to your taste, change the assignment to pa.

SPLOT allows another possibility: You can use digits in its left argument that are indexes into pa. With these, you can change colors without altering pa.

Or you can change the attribute vector av. But this changes default values for almost everything.

USING Changes Colors for Titles

There are quite a few parts to a graph, and you can vary the attribute values for all of them. To do some of the things you may have in mind, you may have to study the function descriptions in Function Reference very carefully. But changing the color of a title seems common, and there is a special function for it.

```
4 1 18 TITLE 'ANOTHER TITLE' USING COLOR 4
```

does about what you would expect: It sets the color of the text of the title to a particular value, without changing the border or the output of any other function.

USING can be used in this way with WRITE, DRAW, and FILL, described in Writing Text and Drawing Flat Pictures. It cannot be used with the functions that draw graphs and charts, axes, labels, or annotations.

Filling and Edging

For another finishing touch, you can fill the area between a graph and its X-axis. Use an F in the left argument of SPLOT.

The area to be filled is crossed by an (imaginary) point along a series of horizontal lines. Whenever the point crosses a boundary of the area, it turns on a visible pattern. Or, if the pattern was already on, it turns it off. If the graph and the axis do not form a closed figure, the area is first closed by extending vertical lines from the endpoints of the graph to the X-axis. The result is like coloring between the lines in a coloring book, but a bit more regular. The technique is called *filling*, and the pattern used is called the *fill pattern*.

The pattern used for filling is controlled by the plot attribute vector (pa). You can display most of the available patterns by executing the demonstration function ATTRIBUTES. When filling is called for, the style value in the attribute code selects one of the available fill patterns. The method of controlling pattern selection for multiple graphs is complex, and is left to the description of pa in Variable Reference.

The edges of a filled area can be made to contrast with the fill pattern by using F in the left argument. This technique is called *edging*. The attribute values for edges are also given by pa. The default value is in the first element of av.

Specifying Other Aspects of Line Graphs

Here is a mixed bag of other things you can do to a simple line graph with the functions in GRAPHPAK.

Using Logarithmic Axes

For this, there are two more characters for the left argument of SPLOT:

`X` Make the X-axis logarithmic to the base 10.
`Y` Make the Y-axis logarithmic to the base 10.

Marking Points on a Graph

Two more characters for the left argument of `SPLOT`:

`P` Mark the points of the graph with symbols; do not connect them with a line.
`p` Mark the points of the graph with symbols and connect them with a line.

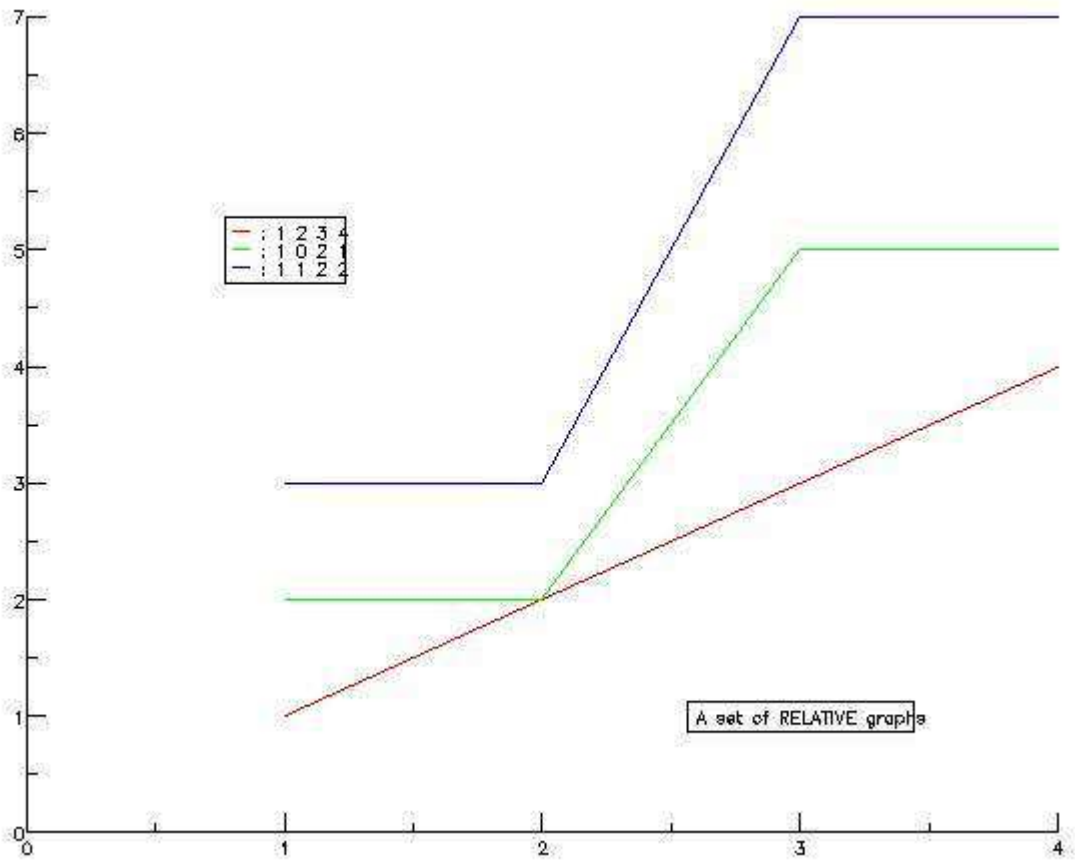
The symbols used to mark the points of successive graphs are contained in the global variable `pc`. As distributed, the value of this vector is `○□⋆∘×∇∆`.

Plotting Relative Graphs

Suppose you have three sets of data - expenses for labor, for overhead, and for materials, say - and you want to plot a graph of each so that the figures add up. That is, the height of the third graph must show the total expense. (The graph below shows an example.) The successive graphs are said to be *relative* to each other.

Plotting them uses another control character.

`R` Add each set of data, element by element, to the previous set, and plot the resulting graphs.



Plotting on a Reference Grid

You can extend the major tick marks on the axes vertically or horizontally across the area of the plot to form a reference grid. The global variable `tm` controls the extension; it can have the following values:

`0` or `0 0`	Do not extend tick marks.
`1 0`	Extend X-axis tick marks vertically.
`0 1`	Extend Y-axis tick marks horizontally.
`1` or `1 1`	Extend both sets of tick marks.

A Note on APL2 Functions

The previous section described the statements needed to produce Creating Axes, Labels, Annotations, and Titles. There were quite a few of them. If you are doing much of this work, remember that you are dealing with APL2 functions: you can combine them into larger functions that will allow you to execute many repetitive steps with one statement.

The function ZIP below may serve as a model. ZIP takes as an argument a matrix M. Each row of M contains the following information:

- A product number
- A quantity of the product on hand at the beginning of the week
- Quantities of the product produced on each of five days of the week (in position 3 through 7)
- Quantities of the product shipped on each of five days of the week (in positions 8 through 12)

ZIP M produces, for each product, a graph that shows, for each day of the week,

- Quantity produced
- Quantity shipped
- Quantity on hand at the end of the day

  (This amount is calculated as the quantity on hand at the end of the preceding day, plus the amount produced, less the amount shipped.)

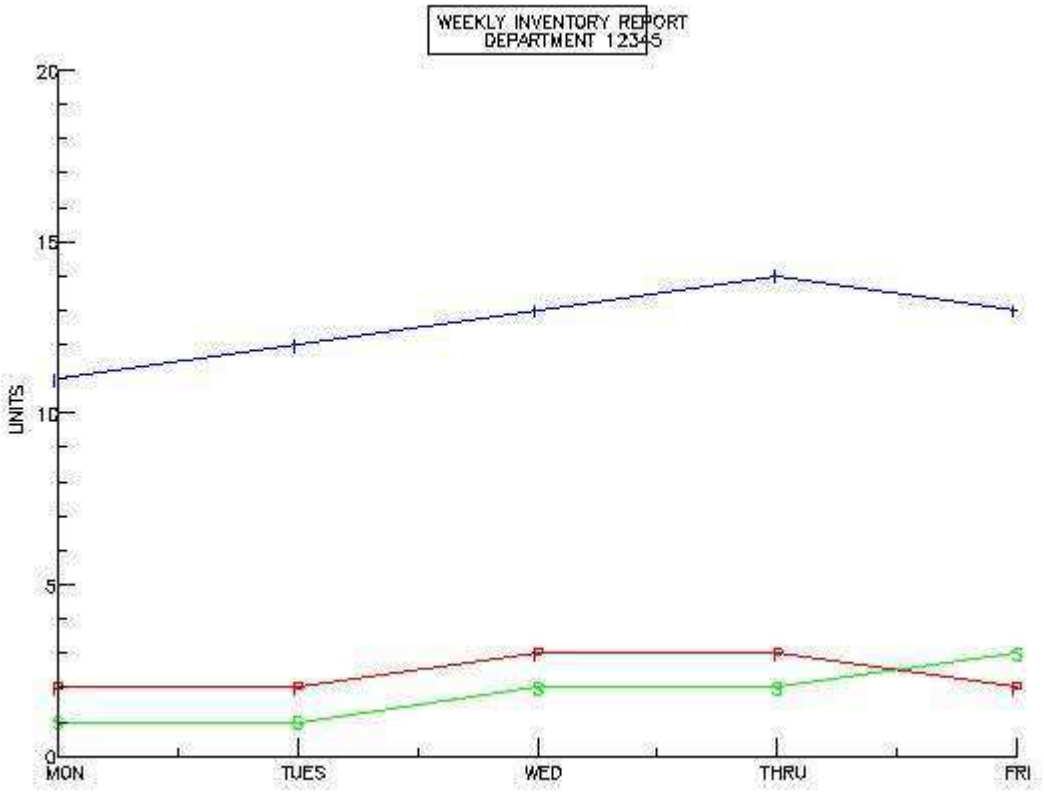
ZIP also exemplifies a number of methods and concepts introduced in this chapter, plus a few new ones. Note the following:

- The necessary data, M, must come from somewhere. Its source is not mentioned.
- ZIP doesn't use VIEW; all its output is printed by the COPYN function. Still, it needs to use ERASE.
- ZIP uses FSSAVE also, to create a copy of each graph that could later be displayed on the screen. (The graph for each department will be in a file called PICnnnnn, where nnnnn is the department number.)
- The output is titled with legends that show the color and symbol used for plotting each type of quantity. This is done by using values in the arguments of the TITLE function that have not been explained in this chapter. They are described with the syntax of TITLE in Function Reference.

Somewhat more elaborate models of plotting line graphs appear in the demonstration functions REVENUES and CAYUGAPLOT.

```
      ∇ZIP;END;INDEX;DEPT;ONHAND;PRODUCED;SHIPPED;INVENTORY;pc
[1]   ⍝ Clear the screen
[2]    RESTORE
[3]   ⍝ The number of graphs is the number of lines in M
[4]    END←1↑⍴M
[5]   ⍝ Set the window, the scaling viewport, and plot characters
[6]    w←1 0 5 20
[7]    svp←10 7 60 50
[8]    pc←'PSI'
[9]   ⍝ Go through the loop for each value of INDEX from 1 to END
[10]   INDEX←1
[11] LOOP:DEPT←M[INDEX;1]
```

```
[12]   ONHAND←M[INDEX;2]
[13]  ⍝ Calculate the amounts
[14]   PRODUCED←M[INDEX; 3 4 5 6 7]
[15]   SHIPPED←M[INDEX; 8 9 10 11 12]
[16]   INVENTORY←1↓+\ONHAND,(PRODUCED-SHIPPED))
[17]  ⍝ Draw the graph
[18]   ERASE
[19]   'pS' SPLOT PRODUCED AND SHIPPED AND INVENTORY
[20]  ⍝ Put on the finishing touches
[21]   1 2 3 4 5 LBLX 5 4⍴'MON TUESWED THURFRI '
[22]   0 LBLY 0
[23]   0 ANNY 'UNITS'
[24]   0 0 1 0 1 TITLE 'WEEKLY INVENTORY REPORT¨    DEPARTMENT ',⍕DEPT
[25]   6 0 1 0 3 TITLE '⊂⍴⊃  PRODUCED¨⊂⍴⊃  SHIPPED¨⊂⍴⊃  INVENTORY'
[26]  ⍝ Save a copy for print and a copy for display
[27]   COPYN 'PRT',⍕DEPT
[28]   FSSAVE 'PIC',⍕DEPT
[29]  ⍝ Repeat the loop
[30]   →(END≥INDEX←INDEX+1)/LOOP
[31] ∇
```



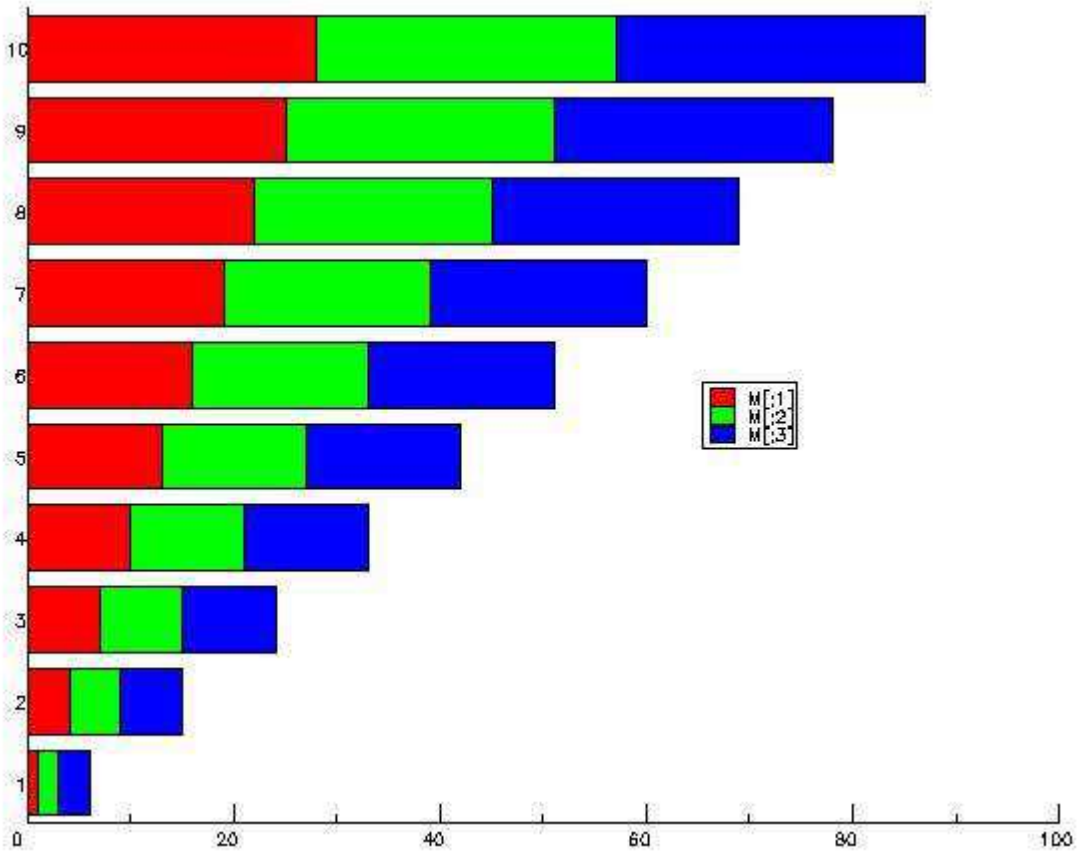
Summary of Basic Concepts

The examples in this chapter may not show the particular kind of output you hope to produce. If some other chapter has what you're looking for, you can go directly from here to there without reading the intervening material.

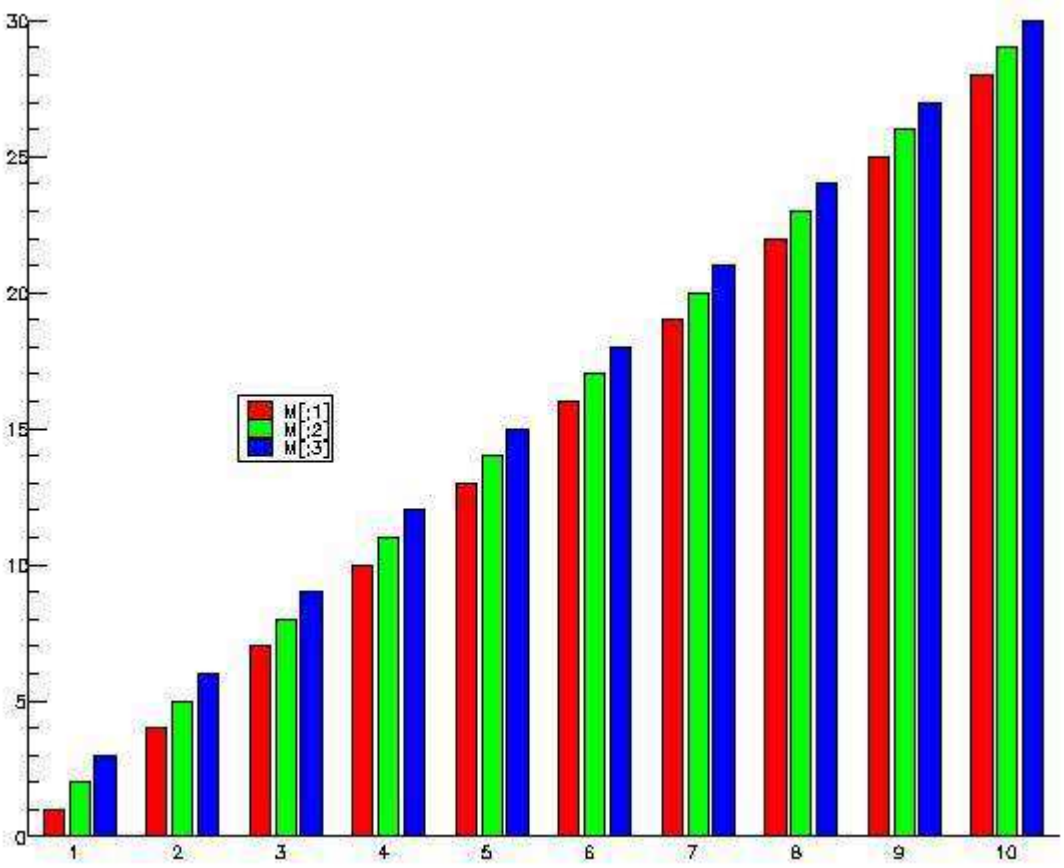
But to produce displays like the examples in any other chapter, you will have to use some basic concepts that were introduced here. Later on they are used without explicit mention. Keep in mind that:

1. You have to start APL2, load GRAPHPAK, get data from somewhere.
2. You have to find a GRAPHPAK function that does what you want. (`PLOT` and `SPLOT` were introduced in this chapter; others appear later.)
3. There are some basic housekeeping routines (`VIEW`, `COPY` and `COPYN`, `FSSAVE` and `FSSHOW`, `ERASE`, `RESTORE`).
4. You may want to specify a new scaling viewport (`svp`) or a window in your problem space (`w`).
5. There are ways to write labels, annotations, and titles; to specify color, style, width, and mode; to specify attributes for edging and filling; to change other aspects of the output. The ways may differ according to the function you use.
6. You are using an APL2 workspace. You can do all these things within other APL2 functions.

A Bar Graph, with Stacked Data



A Column Chart, with Grouped Data



Drawing Bar Graphs, Step Charts, and Pie Charts

This section tells how to draw the following kinds of graphs:

Bar Graphs
: This includes column charts, which are bar graphs with the bars running vertically instead of horizontally.

Step Charts
: These are often used for frequency functions, and in those cases are also known as histograms.

Pie Charts

There are demonstration functions in GRAPHPAK that produce examples of these charts. Execute `REVB` to get a bar graph and `REVC` to get a column chart. `PIES` produces pie charts and `NHIST` a step chart. `MILERUN` gives a more elaborate bar chart, and `HEALTH` a more elaborate column chart.

Drawing Bar Graphs

A *bar graph* or *bar chart* represents data by rectangles stretched horizontally, as in A Bar Graph, with Stacked Data. That graph was produced by the statement

```
'BLf' CHART M
```

A *column chart* does the same kind of thing with rectangles stretched vertically, as in A Column Chart, with Grouped Data. That graph was produced by the statement

```
'GLf' CHART M
```

In both graphs, `M` is this matrix:

```
 1  2  3
 4  5  6
 7  8  9
10 11 12
13 14 15
16 17 18
19 20 21
22 23 24
25 26 27
28 29 30
```

These Things are Different

The data that produced the line graphs in Getting Started and Drawing Line Graphs could as well be plotted with bar or column charts. To make a bar or column chart from your data, instead of a line graph, you need to do only two, maybe three, things differently.

- Use the function `CHART` instead of `PLOT` or `SPLOT`.
- Structure the data for multiple plots differently.
- Pick the control characters for the left argument of `CHART` from a new list.

The following sections describe each of these things in turn.

These Things are the Same

Just for reminders, when using `CHART` you can still

- Specify attribute codes with the functions `COLOR, STYLE, WIDTH`, and `MODE`.
- Write labels, annotations, and titles with the functions `AXIS, LBLX, LBLY, ANNX, ANNY, HOR, VER`, and `TITLE`.

These techniques were all described in Getting Started.

CHART Draws Bar and Column Charts

Use CHART as you would SPLOT. For example, replace the first line of Creating Axes, Labels, Annotations, and Titles with

```
'A' CHART YDATA
```

and compare the result.

Structuring Data for CHART

The right argument of CHART can be a vector, and in that case it is used exactly as is the right argument of PLOT or SPLOT.

For multiple sets of data, there is a difference. CHART does *not* use an array built by AND and VS. It does use a matrix, but the first column is *not* a set of X-values. (That is, it does not specify the positions of bars or columns along the axis of the graph.)

For multiple sets of data, each set of data must be one column of the matrix. For example, to draw a bar chart of ADATA, BDATA, and CDATA, execute

```
'BFL' CHART ADATA,BDATA,[1.1]CDATA
```

(In the example, ',[1.1]' laminates CDATA with BDATA, and ',' catenates the resulting matrix with ADATA. ADATA, BDATA, and CDATA must all have the same length.)

Control Characters for CHART

CHART uses a left argument of control characters, much as SPLOT does. With the control characters you can

- Specify a bar chart rather than a column chart. (Use B).
- Group multiple sets of data, as in A Column Chart, with Grouped Data, rather than stacking them, as in A Bar Graph, with Stacked Data. (Use G.)
- Suppress the axes. (Use A.)
- Label the axes. (Use L.)
- Fill (use F) or fill and edge (use f) the bars or columns.
- Use the scale factors implied by the window in problem space. (Use S.)

If you want to do none of these things, use an empty vector ('').

Results of CHART

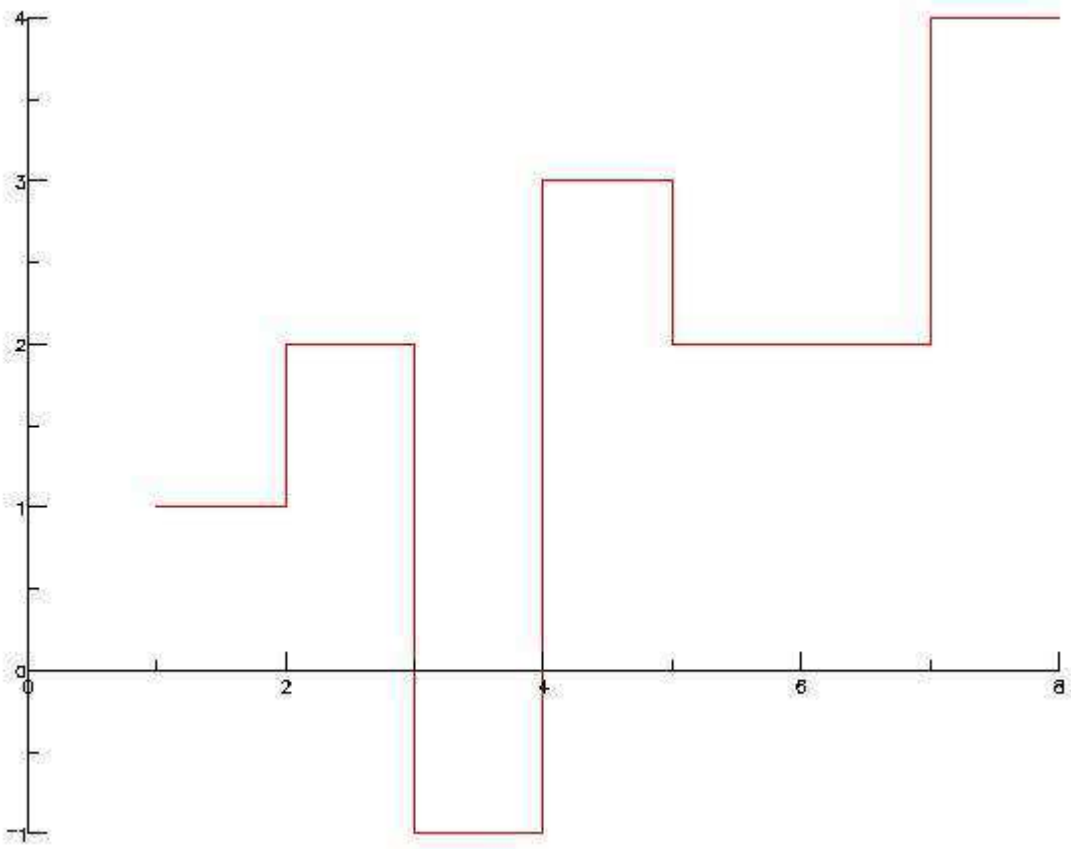
The chart in A Bar Graph, with Stacked Data has the following characteristics:

- The data is *stacked.* That is, the lengths of the bars in each stack are 6, 15, 24, ..., the sums of each row of M. Also the bars are divided to show the relative contribution of each set of data. (If you used g in the left argument, the bars would be grouped, like a horizontal version of A Column Chart, with Grouped Data picture.)

- 1, 4, 7, ... was the first column of the matrix, and so produced the bars at the left end of each stack. (If the data were grouped instead of stacked, the first column of the matrix would produce the bottom set of bars or the left-hand set of columns.)
- All the data in the first column of the matrix is shown with the same set of attribute values. The second column is shown with another set, and so on.
- The bars are centered on the axis at points 1 through 10, because each column of the matrix has 10 elements. But you could label the axes in any way you want. For example, to label the locations of the bars with the letters `A B C D E F G`, do this:
    1. Omit the character `L` from the left argument of `CHART`.
    2. After using `CHART`, execute
    ```
    10 LBLX 'A⍪B⍪C⍪D⍪E⍪F⍪G'
    0 LBLY 0
    ```
- The width of each bar is 0.8 (80%) of the distance between the centers of two adjacent bars. You can change this value by assigning a new value to the variable `bw`. `RESTORE` will set it back to 0.8.

Drawing Step Charts

The picture below is a step chart. It looks a lot like a column chart with columns wide enough to touch each other, and with the dividing lines between them erased.



What's New

The figure uses the same data that was used to produce A Quick Example. To get a step chart instead of a line graph, again you have to do two, maybe three, things a little differently:

1. Use the function `STEP` instead of `PLOT` or `SPLOT`.
2. Structure the data differently for some situations.
3. Pick the characters for the left argument of `STEP` from a new list.

The following sections describe each of these things in turn. You can specify attribute values, and write labels, annotations, and titles, with the same functions you used for `CHART` and `SPLOT`.

STEP Draws Step Charts

Use `STEP` as you would `SPLOT`. For example, out step chart was produced by the statement

```
'L' STEP YDATA
```

Structuring Data for STEP

Structure the data for STEP as for PLOT or SPLOT. If the right argument is a

vector
: the elements of the vector are Y-values, its indexes are the corresponding X-values.

matrix
: then the first column is a set of X-values and the remaining columns are successive sets of Y-values. (But there is an exception to this rule, described below.)

If you have only one set of X- and Y-values, you can use the function VS instead of constructing a matrix for the right argument. For example,

```
'' STEP YDATA VS X
```

Exception for Disconnected Steps

If you use B in the left argument, STEP needs a 3-column matrix for its right argument. In this case, the first two columns contain X-values. Values in the first column are the beginning points of horizontal steps; values in the second column are the corresponding endpoints. Values in the third column are the Y levels of the steps.

Control Characters for STEP

Like CHART and SPLOT, STEP uses a left argument of control characters. The statement that produced our step chart used L, to label the axes. With other control characters you can

- Suppress the axes. (Use A.)
- Draw disconnected steps. (Use B.)
- Fill (use F) or fill and edge (use f) the area under the steps.
- Plot each set of data relative to the previous one. (Use R.)
- Use the scale factors implied by the window in problem space. (Use S.)

Results of STEP

The step chart picture tells nearly all about the results of STEP. But note how the last point on the graph is plotted. In going from one pair of (X,Y) coordinates to the next, STEP draws first a horizontal line and then a vertical one. Hence, if the last two pairs of coordinates in a sequence are not the same, the plot should end with a vertical line up (or down) to the last point. But without a horizontal line at the last point, its Y-value might be less noticeable than the values at other points. So in this case STEP continues the graph to the right by one more index. (If the steps were different widths, STEP would draw a small arrow.)

Plotting Frequency Functions

Step charts are often used to display frequency functions. The X-coordinates of the beginning and end of each step are known as *class boundaries*; the distance between two successive class boundaries is the *class interval*. The height of each step represents the number of measurements that occur between the corresponding class boundaries.

In order to plot the frequency function of a set of raw data, you need a function that constructs the function in the form required by STEP. Here it is.

FREQ Constructs Frequency Functions

To plot a frequency function, execute

```
'L' STEP (classes FREQ data)
```

In this statement,

data
is a numeric vector containing your raw data.

classes
is a numeric vector that specifies the class boundaries and whether the graph is to be normalized or not.

- A graph is *normalized* when the Y-axis is adjusted so that the total area under the curve is equal to 1. If you want this done, make the first element of *classes* equal to 1. Otherwise, make it 0.
- If you want all class intervals to be the same, make the second element of *classes* equal to the interval size, and stop. You need no more elements.
- If you want class intervals of different sizes, put the class boundaries in elements 2, 3, 4, ... of *classes*.

Example

Suppose you would like to plot the frequency function of a set of data contained in the vector FDATA. Suppose further that the elements of FDATA are integers, and that you would like to use class boundaries of ¯0.5, 2.5, 5.5, ..., 32.5. Also, you do not want the graph normalized. Execute

```
CLBNDS←¯3.5+3×⍳12
'FL' STEP (0, CLBNDS) FREQ FDATA
```

Plotting Cumulative Distributions

A step chart of a cumulative distribution shows, for each value, the number of data elements less than or equal to that value. If you are plotting frequency functions, you are very likely plotting cumulative distributions also. The following example illustrates no new feature of GRAPHPAK, but reminds you how to generate a cumulative distribution.

This function will order the data for the distribution:

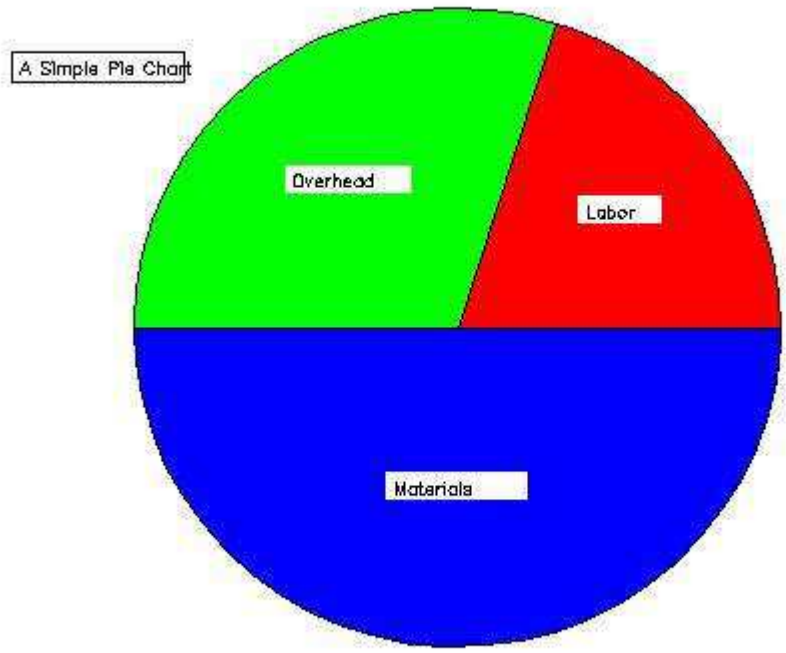
```
∇ Z←DIST X
[1]  Z←X[⍋X], [1.1] (⍳⍴X)÷⍴X
∇
```

And this one will draw a step chart of the distribution:

```
'L' STEP DIST FDATA
```

Drawing Pie Charts

A pie chart shows what proportions the parts of a set bear to the sum of the parts. The whole is shown by a full circle; the parts are sectors of the circle.



Our pie chart was produced by the statements

```
'' PIECHART 2 3 5 WITH 'LaborıOverheadıMaterials'
w←svp
20 60 1 TITLE
```

These Things are Different

A comparison of this example with one of the uses of `SPLOT` in Getting Started and Drawing Line Graphs shows these new elements:

1. There is a new function, `PIECHART`, that uses data in a different way. The numbers 2, 3, and 5 are not interpreted independently of each other. Instead, what is pictured is the ratio that each number has to their total sum. For example, the number 2 tells that "LABOR" takes up 2 parts out of 10.
2. There is a new list of control characters for the left argument.
3. There are no axes for a pie chart, and no annotations. There is a new function, `WITH`, that creates labels.

The following sections describe each of these new items in turn.

PIECHART Draws Pie Charts

To create a pie chart showing the proportion that each number in a set bears to the sum of the set, execute

```
'' PIECHART data
```

Here *data* is a vector containing the set of numbers.

Control Characters for PIECHART

Like SPLOT and CHART, PIECHART uses a left argument of control characters. With these characters you can

- Fill (use F) or fill and edge (use f) the slices of the pie.
- Label the slices with their numeric values. (Use L or N.)
- Label the slices with their relative percentages. (Use P.)
- Draw a box around the labels (use E) and fill the boxes (use B).
- Bound the slices with chords, rather than arcs, so they appear as triangles, rather than as sectors of a circle. (Use C.)
- Label pie segments with text you have placed in a variable named t. (Use T.)

Creating Labels for a Pie Chart

The labels on a pie chart can have up to three parts:

1. Text, entered as an argument of the function WITH (described below).
2. A data value, specified by using the control characters L or N.
3. A percentage value, specified by using the control character P.

WITH

The function WITH combines data and text labels into a right argument for PIECHART. Labels for WITH are similar to those used by LBLX and TITLE in Getting Started and Drawing Line Graphs. They can contain the same delimiters for a new line or a new label. A variable containing labels for WITH may be

- A vector, containing all the labels in a continuous string, with delimiters for new lines and new labels.
- A matrix, with a different label in each row. The labels may contain delimiters for new lines.
- An array of rank 3, with a different label in each matrix of the array. Each row of the matrix is a different line. No delimiters are needed.

Our example shows WITH using a numeric vector as a left argument and a character vector as a right argument. Actually, the order of the two items is not important - the numbers could appear on the right and the characters on the left.

Using USING with WITH

You can change the color of the text portion of a label (but *not* of the data values or percentage values) by using USING with WITH as it is used with TITLE. USING must follow immediately the argument that contains text, as in

```
'' PIECHART 1 1 WITH 'TOP¨HALFɪBOTTOM¨ HALF' USING COLOR 3
```

PIELABEL

If you want to change the attribute values completely between the drawing of a pie chart and the labeling of it, use PIELABEL. Like this...

1. Execute

   '' PIECHART *data*

2. Change the attribute values as you wish.
3. Execute

   data PIELABEL *label*

   where *label* is a variable containing the label you want and *data* is the same as in step 1.

If you use PIELABEL, you cannot use L, N, or P in the left argument of PIECHART.

Using TITLE with PIECHART

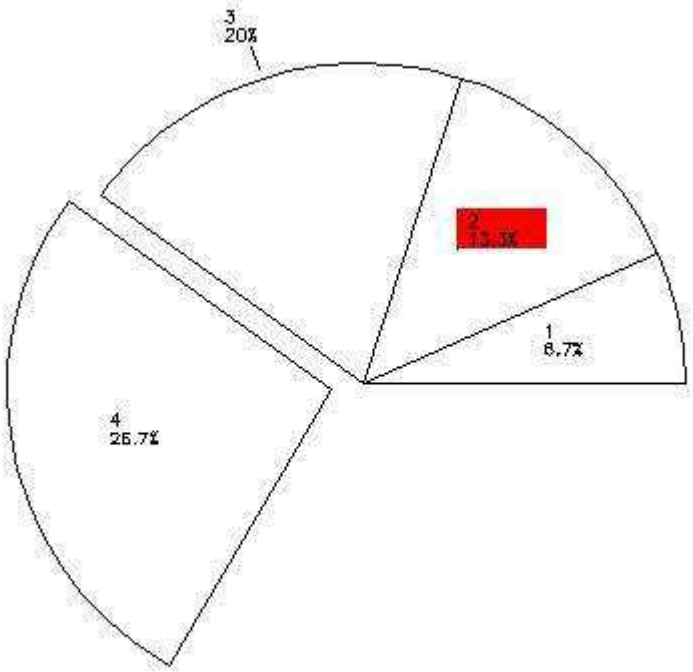
PIECHART does not set a window in problem space or change the scaling viewport. In order to use TITLE, it helps to specify these. In the example that produced out pie chart, the assignment

```
w←svp
```

makes both the window and the scaling viewport equal to the maximum screen size. Then TITLE can use coordinates in virtual space.

Other Controls for PIECHART

There are several other fancy things you can do with the slices in a pie chart. Some of them are illustrated in this chart.



The figure was produced by the statement

```
'LP' PIECHART DATA
```

The variable DATA looked like this:

```
1   0   0   0   0
2   2   0   0   0
3   0  ¯1   0   0
4   0   0  .1   0
5   0   0   0   1
```

In this matrix, each row represents one slice of the pie. The number governing the size of the slice is in column 1 of the row. The number in columns 2 through 5 tell what variations (if any) are used in displaying the slice.

What the Numbers in the Matrix Do

To use one or more of the variations illustrated, do this:

1. Create a 5-column matrix as a right argument for PIECHART. Put the numeric data in the first column and zeros elsewhere.
2. In column 2, put attribute codes for label backgrounds. (You can have filled backgrounds without having boxes around the labels, as in slice 2.)
3. Put any of the following values in column 3. They vary the placement of a label:

a positive number *s*
Puts the label outside the pie. Offsets the label from the slice by *s* times the pie radius.
a negative number
Puts the label outside the pie. Lets `PIECHART` choose the position. (In the figure, slice 3 is outside the pie in a position chosen by `PIECHART`.)

If you leave zero in this column, `PIECHART` will put the label inside the pie if it will fit. Otherwise it will put it outside the pie in a default position, as it did for slice 1.

4. Put any number *s* (positive or negative) in column 4 to offset a slice away from the center of the pie. The amount of offset is *s* times the radius of the pie; a positive offset is directed outward from the center. (Slice 4 in the figure is offset by one-tenth of a radius.)
5. Put a 1 in column 5 to omit drawing the slice and its label. (Slice 5 in the figure is omitted.)

If you are drawing pie charts, you will probably experiment with different settings of these controls to find the variation most pleasing to your eye or most appropriate to your purpose. Just for fun, then, you might like to see what happens when you offset each slice of a pie inward by a whole radius. Try executing

```
'' PIECHART 1,0,0,[1.5]7⍴¯1
```

What `pie` does

The global variable `pie` controls still other aspects of pie charts. The first two elements of this vector contain the coordinates (in virtual space) of the center of the pie, and by changing these you can put your pie wherever you want it on the screen. Other elements control

- The radius of the pie.
- Where the first slice starts. (The default setting starts the first slice at 3 o'clock, as in the figures.)
- The angular span of the whole pie. (The setting is a number of degrees. By setting it to 180, your whole pie will be a semicircle.)
- The attributes of label backgrounds.

`RESTORE` restores the default values of `pie`.

Drawing Polygons

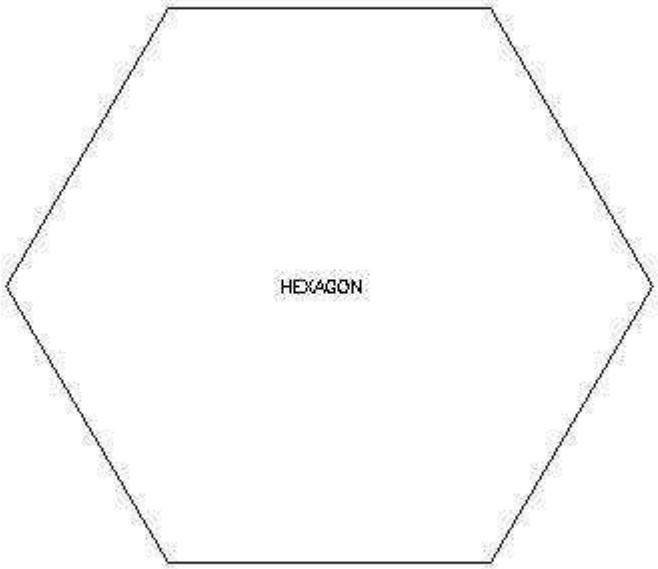
A polygon has nothing much to do with a pie chart, but the `PIECHART` function was easily adaptable to drawing polygons, so here it is.

```
'C' PIECHART 1 1 1 1 1 1
```

draws six equal slices, with chords instead of arcs - a regular hexagon. But the radii are still there.

To get rid of the radii, use `R` in the left argument. Then `PIECHART` will draw a regular polygon, using the first element in the right argument as the number of sides. The following graph was drawn by

```
'R' PIECHART 6 WITH 'HEXAGON'
```



Representing Surfaces

This section tells how to draw the kinds of charts listed below. Each of these charts represents the height of a point above (or below) the (X,Y)-plane.

- Surface Charts.
  The 3-dimensional analogy to a line graph is a network. Like the line graph, the network is made of straight line segments. But the line graph extends over only an interval on the X-axis; the network extends over a rectangle in the (X,Y)-plane. On your display screen or on the printed page we can picture the network only by a 2-dimensional projection. Such is a surface chart.
- Skyscraper Charts.
  These are the 3-dimensional analog to column charts.
- Contour Maps.
  These represent surfaces by contour lines, as in the topographic maps so well-known to wilderness hikers.

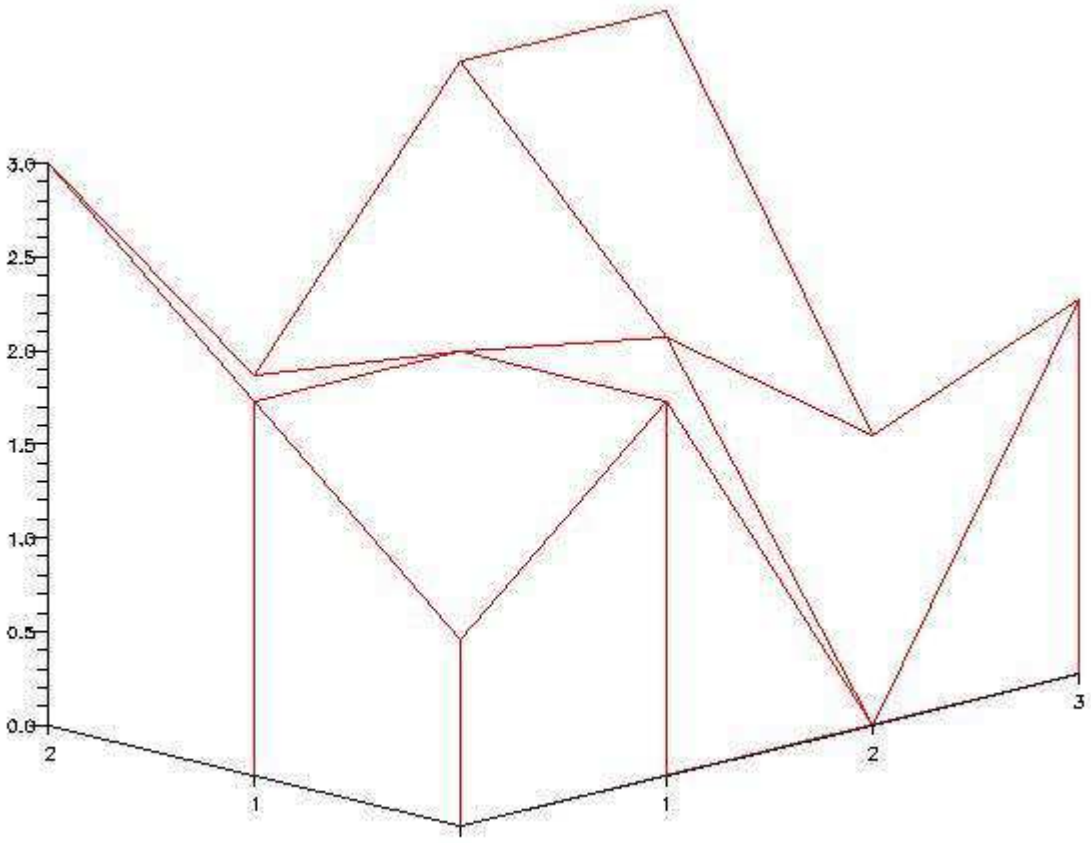
For still other examples of these charts, execute the demonstration functions `WAVEGUIDE, SKYSCRAPER,` and `WGCONT`.

Drawing Surface Charts

Compare the following chart with any of the graphs produced by `SPLOT` in Getting Started and Drawing Line Graphs. The surface chart is the 3-dimensional analog to the line graph. To produce one, we need these new things:

- A new function, called `SURFACE`
- A way to structure data that describes a surface
- A list of control characters for the left argument of `SURFACE`
- Some new functions for putting in axes, labels, and titles (because there are now three axes, instead of two).

The following sections describe each of these things in turn.



SURFACE Draws Surface Charts

The surface chart on the previous page was produced by the statements

```
M←3 4⍴3 1.6 3 3, 2 2 1.8 1, 1 2 0 2
'L' SURFACE M
```

Structuring Data for SURFACE

The heights to be plotted in a surface chart must appear in the right argument of SURFACE as a matrix. The matrix M, plotted in our surface chart, is

```
3  1.6 3   3
2  2   1.8 1
1  2   0   2
```

The function regards the value in the lower left corner of the matrix as the Z-coordinate corresponding to the (X,Y)-coordinates (0,0). It plots that point on the screen *nearest* you.

If the matrix has *n* rows and *m* columns, the function regards the value in the upper right corner as the Z-coordinate corresponding to the (X,Y)-coordinates (*m*,*n*). It plots that point on the screen *farthest* from you.

(The example has 3 rows and 4 columns. The value Z=3, corresponding to X=3, Y=2, is *farthest* from you in the perspective of the figure.)

The X-axis, if not suppressed, extends from the nearest point toward your right. The rows of the matrix extend along the X-axis. The Y-axis extends to your left. The columns of the matrix extend along the Y-axis. The Z-axis appears where the value in the upper left corner of the matrix is plotted, and extends upward.

Control Characters for SURFACE

SURFACE uses a left argument of control characters, much as SPLOT does. In producing our chart, the character L was used to put labels on the axes.

With the control characters, you can also do the following:

- Suppress axes. (Use A).
- Use scaling factors you have defined yourself.
- Extend traces across the surface parallel to the X-axis or the Y-axis. The traces parallel to one axis are drawn at the location of the tick marks on the other axis. (Use X or Y, and see the note on traces below.)

Defining Scaling Factors

To use scaling factors of your own choice, do this:

1. Define a scaling viewport in svp, as you did for SPLOT in Getting Started.
2. Assign the limits on Z-values to sw[1 2], a global variable called the *surface window*.
3. Assign the maximum X- and Y-values to sw[3 4].
4. Use S as a control character in the left argument of the function.

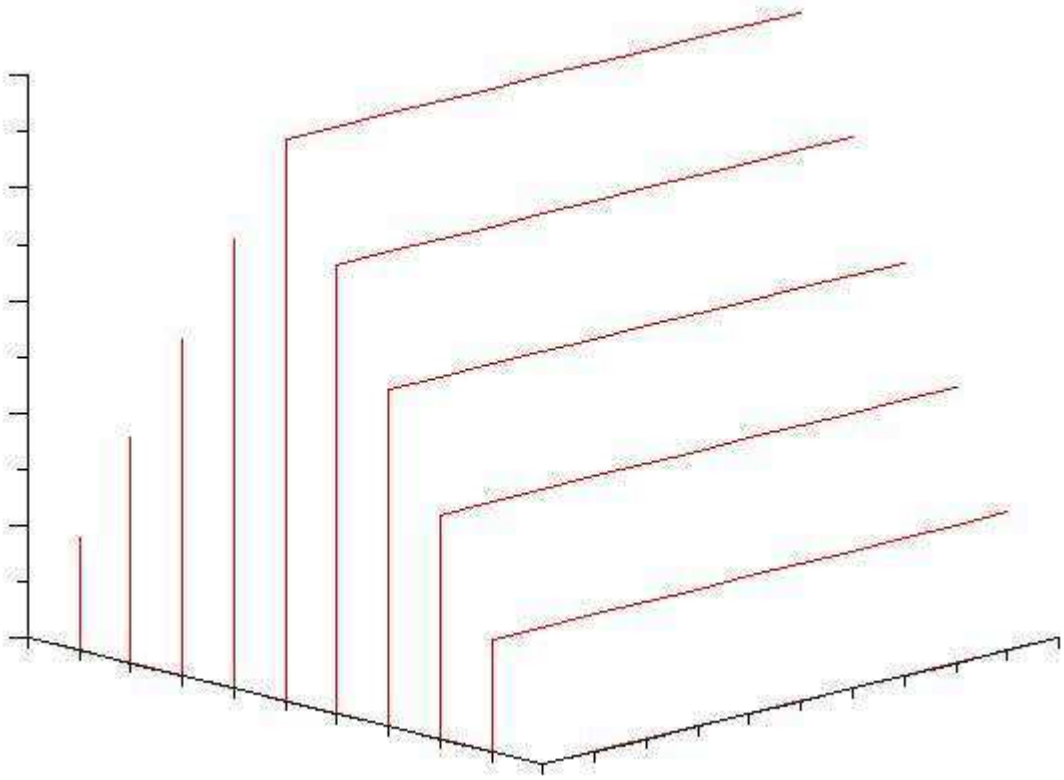
A Note on Traces

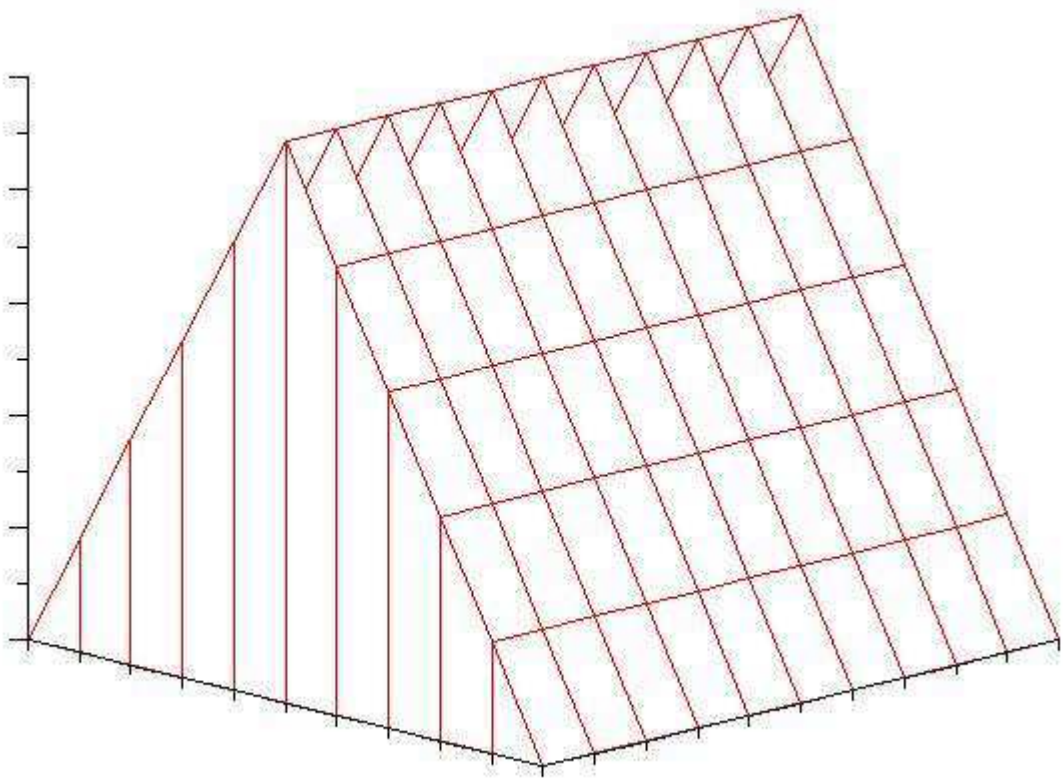
The method of plotting traces plots the X-traces and Y-traces independently. In some cases, particularly in surfaces with sharp ridges, a trace is displayed that should have remained hidden. The effect is illustrated in the following two figures. The first is plotted with X-traces only, and the second with both X- and Y-traces. The statements that produced them were

```
      Q←⍉11 11⍴0 1 2 3 4 5 4 3 2 1 0
      'AX' SURFACE Q
```

and

```
      'AXY' SURFACE Q
```





Drawing Axes, Labels, and Titles for Surface Charts

Axes

You can specify X-, Y-, and Z-axes for a surface chart with the functions `SAXISX`, `XAXISY`, and `XAXISZ`. Each takes a right argument only, which may have the following values:

Scalar 0
: specifies using the default positions for the axis and its tick marks.

Vector
: gives locations for tick marks by their coordinates in the problem space.

You can also specify axes with the function `SAXES`, which takes no arguments and produces default settings for all three axes.

Labels

You can specify labels for the axes of surface charts with the functions `SLBLX`, `SLBLY`, and `SLBLZ`. These work very much like `LBLX` and `LBLY`, described in Getting Started. The left argument can be a scalar 0 or the vector that you used as a right argument of `SAXISX`, (or `SAXISY`, or `SAXISZ`). The right argument contains the label, just as in `LBLX`.

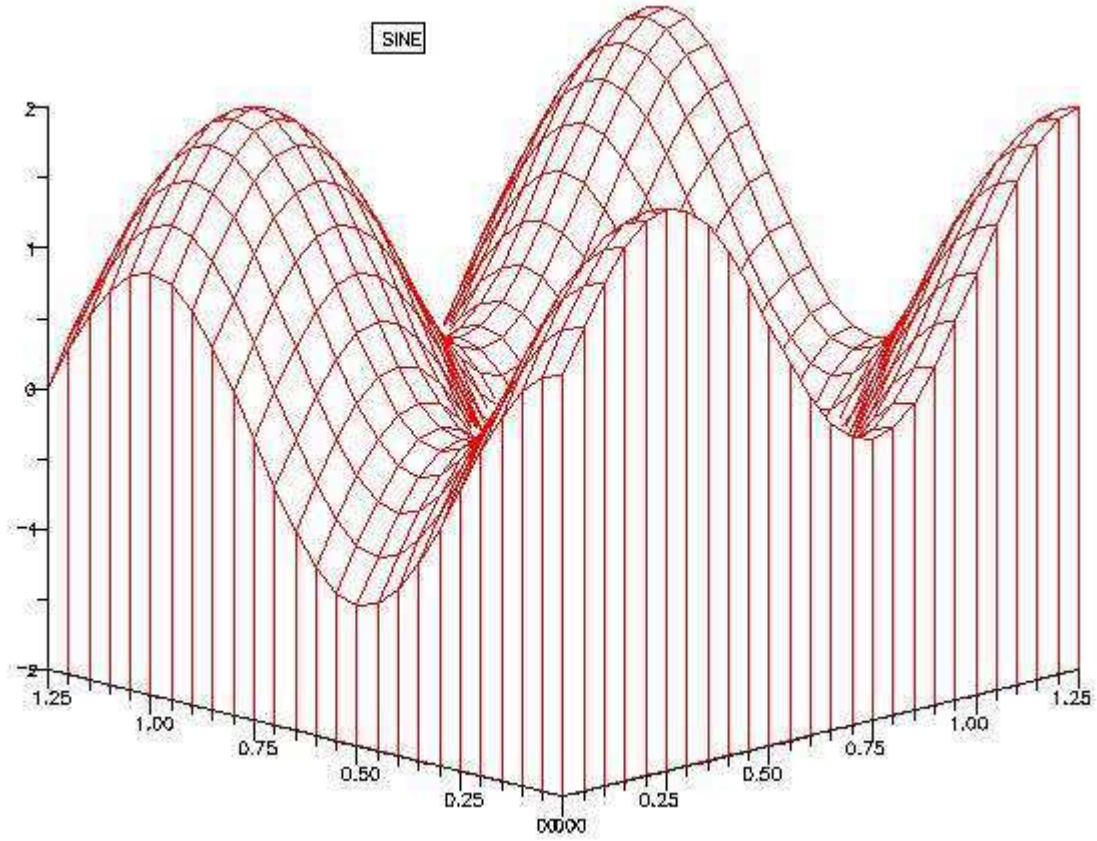
You can also specify labels with the function SLABEL. It takes no arguments and produces default labels for all three axes.

Titles

You can specify titles for surface charts with the function STITLE. This works like TITLE, described in section 1, except that now the first *three* elements of the left argument give the location of the center of the title.

The next chart illustrates the results of these functions. It was produced by the following statements:

```
⎕IO←0
'XY' SURFACE A+⍉A←26 26⍴1○○(⍳26)÷10
0 5 10 15 20 25 SLBLX 0 .25 .5 .75 1 1.25
0 5 10 15 20 25 SLBLY 0 .25 .5 .75 1 1.25
0 SLBLZ 0
5 13 4.75 1 STITLE 'SINE'
```



Writing Text

For a suggestion about using WRITE to place text on a surface chart, see Writing in Three-dimensional Space.

Drawing Skyscraper Charts

A skyscraper chart is another way to show heights above a 2-dimensional base: it represents the same kind of data as a surface chart does. To produce one, you need:

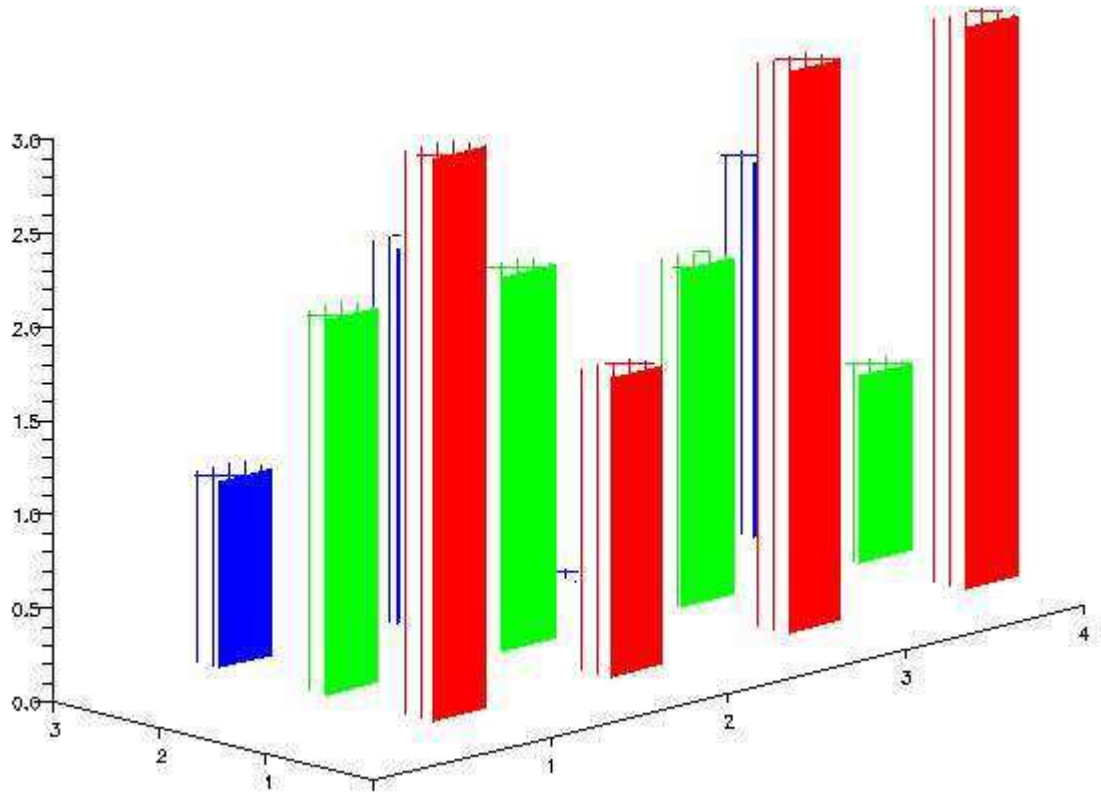
- A new function, called `SS`
- A new list of control characters for the left argument of `SS`

You do not need a new way to structure data.

SS Draws Skyscraper Charts

Our skyscraper chart uses the same data as the surface chart. It was produced by these statements:

```
M←3 4⍴3 1.6 3 3, 2 2 1.8 1, 1 2 0 2
'LF' SS M
```



Structuring Data for SS

`SS` uses a right argument identical to that used by `SURFACE`. Unlike `SURFACE`, `SS` plots the value in the *upper* (not lower) left corner of the matrix at the point *nearest* you on the screen. Furthermore, it regards this value as

corresponding to the (X,Y)-coordinates (1,1), instead of (0,0). This separates the columns of the graph from the axes.

A comparison of the surface and skyscraper charts shows that the first plots the same data as the second, but back-to-front. If this is troublesome, you can easily turn one of them around. Try plotting

```
'L' SS ⊖M
```

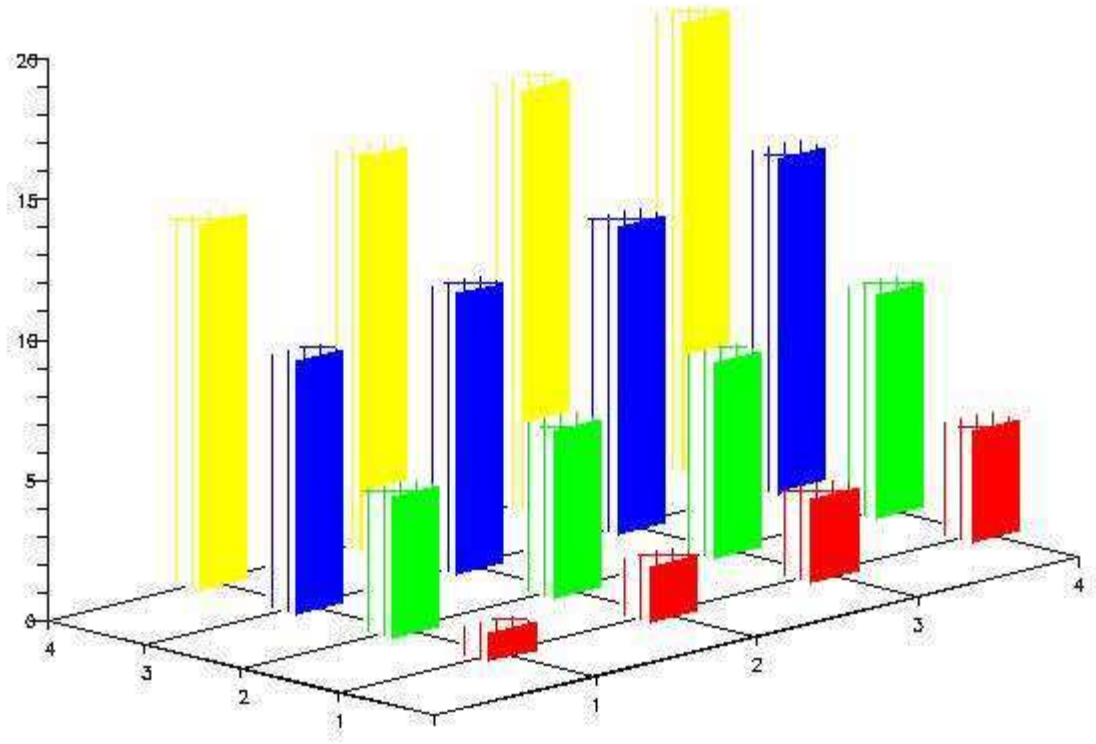
Control Characters for SS

Like `SPLOT` and `SURFACE`, `SS` uses a left argument of characters to control aspects of the output. Again `L` puts labels on the axes.

With the control characters, you can also do the following:

- Suppress axes. (Use `A`).
- Use scaling factors you have defined yourself. (Use `S`, define `svp` as you did for `SPLOT` in Getting Started, and assign the limits on Z-values to the first two elements of the surface window, `sw`.)
- Fill the columns. (Use `F`.)
- Fill and edge the columns. (Use `f`.)
- Draw a base grid. (Use `G`.)

The next chart shows filled and edged columns and a base grid.



Other Controls for SS

On a first pass at skyscraper charts, you can skip this section. But when you want a really elegant chart, you may want to tune your output by changing the defaults for

- Column widths
- Spread factors
- Attribute values for sides and tops of columns

Column Widths

The default widths of columns are 0.3 (30 %) of the distances between the centers of adjacent columns, in both the X- and Y-directions. You can change these values by assigning new values to the 2-element vector `sbw`. `RESTORE` sets both values back to 0.3.

Spread factors

The ratio of the two spread factors is the ratio of a unit distance along the X-axis to a unit distance along the Y-axis. If these distances were the same - that is, if the spread factors were 1 1 - then all the columns in the back of a skyscraper chart would be directly behind some column in the front. A ratio of 5 to 3 seems to minimize obstruction of columns when the default column widths are used, so the default spread factors are 5 3.

You can change the spread factors by assigning new values to `sw[7 8]`. But if you assign them as 1 1, the default of 5 3 will be used.

Attributes for sides and tops of columns

The attribute codes for the right side, left side, and top of a column in a skyscraper chart are specified by elements 9, 10, and 11 of the surface window, `sw`. To change the patterns, assign pattern numbers (not attribute codes) to these elements, as in

```
sw[9 10 11]←5 7 9
```

Drawing Axes, Labels, and Titles for SS

Use the same functions that draw axes, labels, and titles for `SURFACE`, namely `SAXISX, SLBLY, STITLE`, and so on.

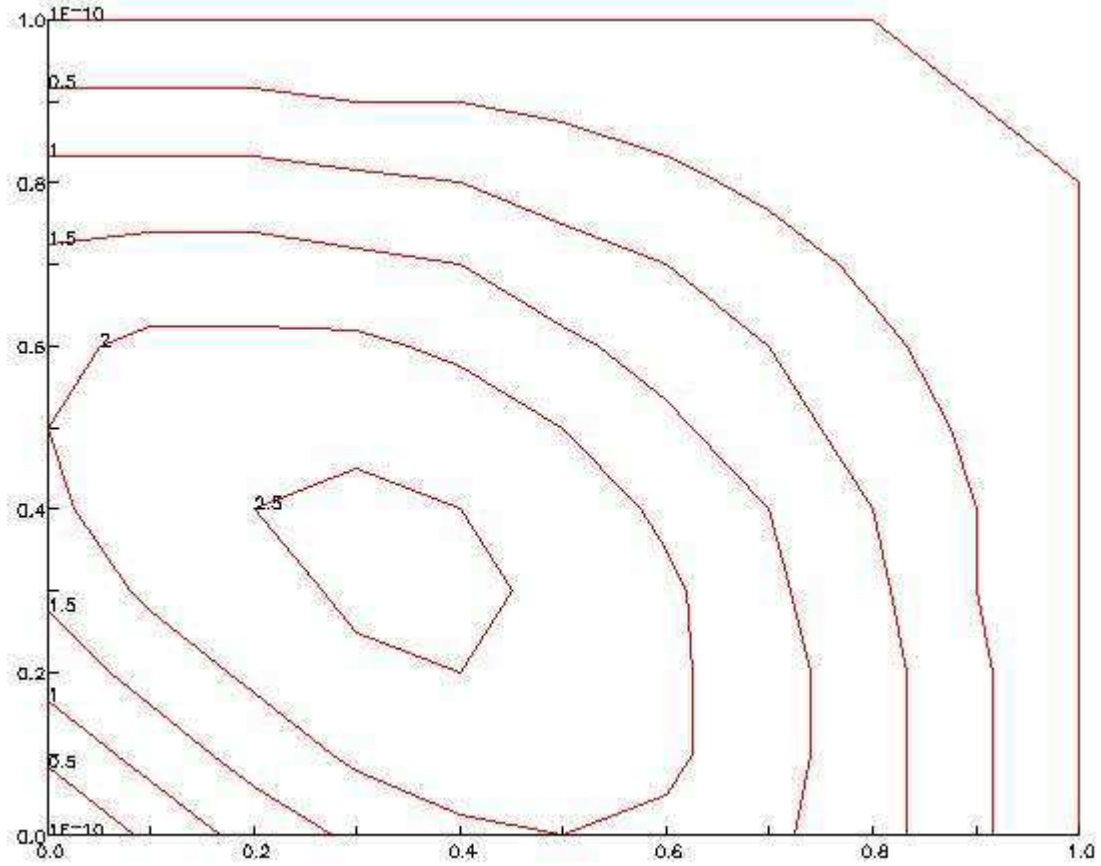
For a suggestion about using `WRITE` to place text on a skyscraper chart, see Writing in Three-dimensional Space.

Drawing Contour Maps

This set of data:

```
0   0   0   0   0   0   0   0   0   0   0
0.6 0.6 0.6 0.5 0.5 0.4 0.3 0.2 0.1 0   0
1.2 1.2 1.2 1.1 1   0.8 0.6 0.4 0.2 0.1 0
1.6 1.7 1.7 1.6 1.5 1.2 1   0.7 0.4 0.2 0
1.9 2.1 2.1 2.1 1.9 1.6 1.3 1   0.6 0.3 0
2   2.3 2.4 2.4 2.3 2   1.6 1.2 0.8 0.4 0
1.9 2.3 2.5 2.6 2.5 2.3 1.9 1.5 1   0.5 0
1.6 2.1 2.3 2.6 2.6 2.4 2.1 1.6 1.1 0.5 0
1.2 1.7 2.1 2.4 2.5 2.4 2.1 1.7 1.2 0.6 0
0.6 1.2 1.7 2.1 2.3 2.3 2.1 1.7 1.2 0.6 0
0   0.6 1.2 1.6 1.9 2   1.9 1.6 1.2 0.6 0
```

can be displayed as in the following chart. The chart is called a contour map.



A contour map depicts a surface by showing the lines of constant height. The lines are represented by their projection on the (X,Y)-plane.

To draw a contour map of a surface, you need:

- A new function, called `CONTOUR`
- A new way to structure data

- A way to specify the heights at which contour lines are to be drawn

CONTOUR Draws Contour Maps

With the data shown assigned to the variable Z, the contour map was produced by these statements:

```
X←0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1.0
Y←1.0 .9 .8 .7 .6 .5 .4 .3 .2 .1 0
SAMP←(Y BY X) OF Z
SAMP[1;1]←1
LEVELS←0 .5 1 1.5 2 2.5
LEVELS CONTOUR SAMP
X AXIS Y
LABEL
```

In addition to the new function CONTOUR, this example contains some other unfamiliar elements:

- The variables SAMP and Z, and the functions BY and OF, are described under Structuring Data for Contour.
- The argument LEVELS is described under Specifying the Heights of Contour Lines.

Structuring Data for Contour

The Height Matrix

The heights of points above the (X,Y)-plane must be given by a matrix, shown below.

Each row of the matrix is a set of Z-values corresponding to a constant value of Y and varying values of X. Each column is a set of Z-values corresponding to a constant value of X and varying values of Y.

Axis Values

In order to plot the Z-values using CONTOUR, the values of X and Y must be attached to the matrix as an extra row at the top, and an extra column on the left. That operation is exactly what the functions BY and OF do, so that after the statement

```
SAMP←(Y BY X) OF Z
```

the matrix SAMP looks like this:

```
0   0   0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
1   0   0   0   0   0   0   0   0   0   0   0
0.9 0.6 0.6 0.6 0.5 0.5 0.4 0.3 0.2 0.1 0   0
0.8 1.2 1.2 1.2 1.1 1   0.8 0.6 0.4 0.2 0.1 0
0.7 1.6 1.7 1.7 1.6 1.5 1.2 1   0.7 0.4 0.2 0
0.6 1.9 2.1 2.1 2.1 1.9 1.6 1.3 1   0.6 0.3 0
0.5 2   2.3 2.4 2.4 2.3 2   1.6 1.2 0.8 0.4 0
0.4 1.9 2.3 2.5 2.6 2.5 2.3 1.9 1.5 1   0.5 0
0.3 1.6 2.1 2.3 2.6 2.6 2.4 2.1 1.6 1.1 0.5 0
0.2 1.2 1.7 2.1 2.4 2.5 2.4 2.1 1.7 1.2 0.6 0
0.1 0.6 1.2 1.7 2.1 2.3 2.3 2.1 1.7 1.2 0.6 0
0   0   0.6 1.2 1.6 1.9 2   1.9 1.6 1.2 0.6 0
```

Tags on Contour Lines

Finally, the number in the upper left corner of the matrix is used to control whether or not the contour lines are tagged with the values of the height they correspond to. The assignment

```
SAMP[1;1]←1
```

puts tags on the contour lines. If the number has any other value than 1, the tags are omitted.

The final form of SAMP is in the GRAPHPAK workspace. To reproduce the figure, execute

```
(0.5×⍳5) CONTOUR SAMP
```

Specifying the Heights of Contour Lines

The left argument of CONTOUR is a vector (LEVELS in the example) that gives the heights for which contour lines are to be drawn.

Specifying Axes, Labels, Annotations, and Titles

You can put axes, labels, annotations, and titles on contour maps with the same functions that were described in Getting Started. *Not*, please note, with the functions that you used with SURFACE or SS, because a contour map has only two axes, not three.

Plotting Saddle Points

At a saddle point on a surface, two separate contour lines touch. You can keep these lines separate by assigning to the variable eps some (normally small) negative and positive deviation, say

```
eps←¯1E¯10 1E¯10
```

CONTOUR adds eps to the contour height wherever a contour goes exactly through one of the grid points.

Performance

CONTOUR uses an iterative technique to find contour lines. The technique allows considerable generality in finding open or closed curves with single or multiple branches, but be warned that it can be fairly time-consuming.

Fitting Curves

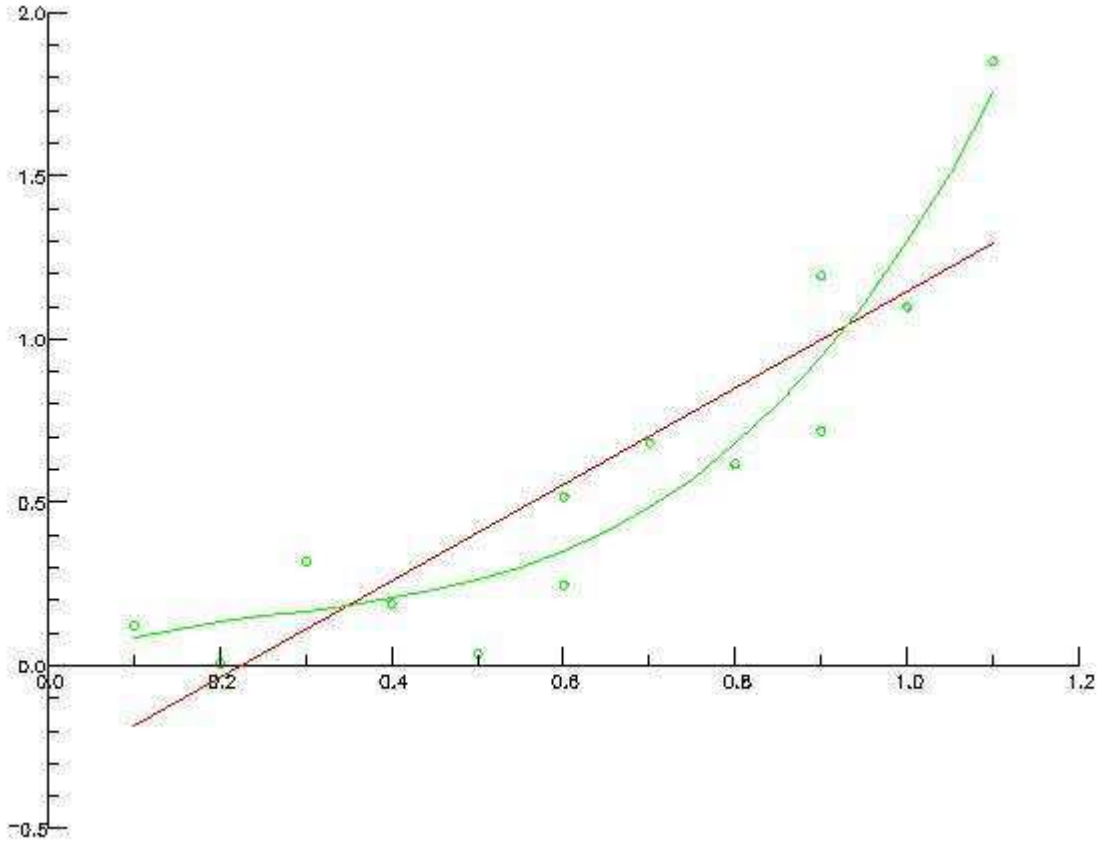
The following graph shows a plot of several points, together with the "best" straight line and the "best" cubic polynomial that can be chosen to approximate them. (Here "best" means a least-squares approximation: The curve chosen represents the function that minimizes the sum of the squares of the differences between the actual values and the function values.)

The graph was produced by the following statements:

```
CLEAR
FIT SL XYTEST
FIT 3 POLY XYTEST
```

Fitting Least-Squares Approximations describes how to use several new functions to fit least-squares approximations of different kinds to sets of points.

Fitting a Spline Curve describes a function that fits spline-like curves, which are required to take on the exact values of a set of points.



Fitting Least-Squares Approximations

When finding a least-squares approximation, the plot of the resulting function may not be what you want most. An expression for the function, a way of calculating other values of it, may be more important to you. All of these things are obtained by the same steps in GRAPHPAK.

Here's what you have to do:

1. Structure your data.
2. Choose from a list the function that describes the kind of approximation you want.
3. Execute

   ```
   CLEAR
   FIT function data
   ```

Structuring Data for a Least-Squares Fit

The data `XYTEST`, used in the example, is in your GRAPHPAK workspace. It looks like this:

```
0.1   0.127
0.2   0.11
0.3   0.324
0.5   0.042
0.6   0.521
0.9   0.723
0.4   0.193
0.6   0.251
1     1.104
0.8   0.622
1.1   1.855
0.7   0.686
0.9   1.198
```

There are two points to make about the data structure:

1. The data is a 2-column matrix, with X-values in the first column and corresponding Y-values in the second.
2. The points do NOT need to be sorted in order of ascending X-value (or in any other order).

Choosing a Function for a Least-Squares Fit

Choose one of the functions below to describe the kind of mathematical function you would like to fit to your data.

`AVG`
: fits a horizontal line, `Y←C`. The height of the line above the X-axis is the average of the Y-values for your data.

`SL`
: fits a straight line, `Y←C1+C2×X`.

n `POLY`

fits a polynomial function, `Y←+/C×X*(0,⍳n)`. It takes a left argument that must be an integer, giving the degree *n* of the polynomial to be fitted.

`EXP`, `LOG`
: fit an exponential function, `Y←C1××C2×X`. The technique used is to transform to semi-log coordinates and find the best fit by a straight line. `EXP` transforms back again and `LOG` doesn't.

`POWER`, `LOGLOG`
: fit a power function, `Y←C1+X*C2`. The technique used is to transform to log-log coordinates and find the best fit by a straight line. `POWER` transforms back again and `LOGLOG` doesn't.

Each of these functions uses a right argument like `XYTEST` (the one used in the example). `POLY` is the only one of the functions that takes a left argument also.

Each of these functions produces a matrix like its right argument. The first column of the matrix is a set of X-values spanning the domain of the input data. The second column is the set of corresponding values of the function. (For `AVG` and `SL`, which plot as straight lines, only two rows are needed.)

`fitmsg`

There is another output from each of these functions, not immediately noticeable. Each function sets the global variable `fitmsg` to an expression that describes the function fitted. For example, after `SL XYTEST`, `fitmsg` becomes

```
      Y←¯0.30498-1.447×X
```

and after `3 POLY XYTEST` it is

```
      Y←(X∘.*0 1 2 3)+.×0.13855 ¯.16441 ¯0.21048 1.5396
```

(The examples in this section have been worked with the printing precision (`⎕PP`) set to 5. If you still have yours set to 10, you will see more digits after the decimal places.)

You can use the expression in `fitmsg` to calculate values of the function for any values of `X` you like. For example, define a function

```
      ∇ FUNC X
[1]   ⎕←⍎fitmsg
      ∇
```

The execution of `FUNC 4` will display the value, for `X` = 4, of the latest function in `fitmsg`.

Or you can let `FIT` do this for you.

FIT Plots a Fitted Function

`FIT` is designed to be used with `AVG`, `SL`, `POLY`, and so on to do several things:

- It plots a graph of the best-fit function, along with a plot of the original data points. (`AVG` and the others save these points in the global variable `fitpts`.)

- It labels axes (you can't suppress this).
- It displays a description of the best-fit function, from `fitmsg`.
- It creates a function called `FITFUN`, which executes the function described in `fitmsg`. For example, executing

  ```
  Y←FITFUN 4 5 6 7
  ```

  assigns to the vector `Y` the values of the best-fit function for `X` = 4, 5, 6, and 7.

You do all this by executing

```
FIT function data
```

where *function* is one of the curve-fitting functions described in this section, and *data* is a 2-column matrix of the type described earlier.

CLEAR Initializes Axes

Before you use `FIT` for the first time, and every time you want to change the axes and display a set of data, you need to initialize things. The function `CLEAR` does this. It also executes `ERASE`. The first use of `FIT` after `CLEAR` will draw the axes and display the data points, as well as display the curve.

Some Variations on Least-Squares Fitting

This page describes some other functions that do odd jobs for `FIT`.

Omitting Points

It's not unreasonable, sometimes, to throw away some data and see what that does to your best fit. `SCRATCH` does this.

If you know the row number of the point

If you look at our least squares plot and decide to omit the fourth point (0.5, 0.042) and the eighth point (0.6, 0.251), execute

```
NEWTEST←SCRATCH XYTEST[4 8;]
```

`NEWTEST` will look like `XYTEST` with points 4 and 8 omitted. Now you can execute

```
FIT AVG NEWTEST
FIT 3 POLY NEWTEST
```

and compare the results with the previous fits.

If you don't know the row number of the point

Rather than examine the matrix and try to identify the points you don't want, execute

```
FIT 3 POLY XYTEST
NEWTEST←SCRATCH ''
```

These statements display the plot of your points and the fitted curve, with the cursor in the middle of your screen. You can now use your cursor as an eraser. To throw away data points:

1. Move the cursor near a point you want to erase. (The cursor doesn't have to touch the point exactly.)
2. Press any function key.
3. Repeat the first two steps for any other points you want to erase.
4. End the process by pressing the Enter key twice.

Now the points you selected are crossed out by "+" signs. The points that are not crossed out are in `NEWTEST`, and you can see what effect your erasing had by executing

```
FIT 3 POLY NEWTEST
```

Plotting in Semi-log and Log-log Coordinates

`EXP` finds a best-fit exponential function by transforming your data to semi-log coordinates and fitting a straight line to the result. `LOG` does the same thing, but omits transforming back to linear coordinates, so that the result plots as a straight line in semi-log coordinates.

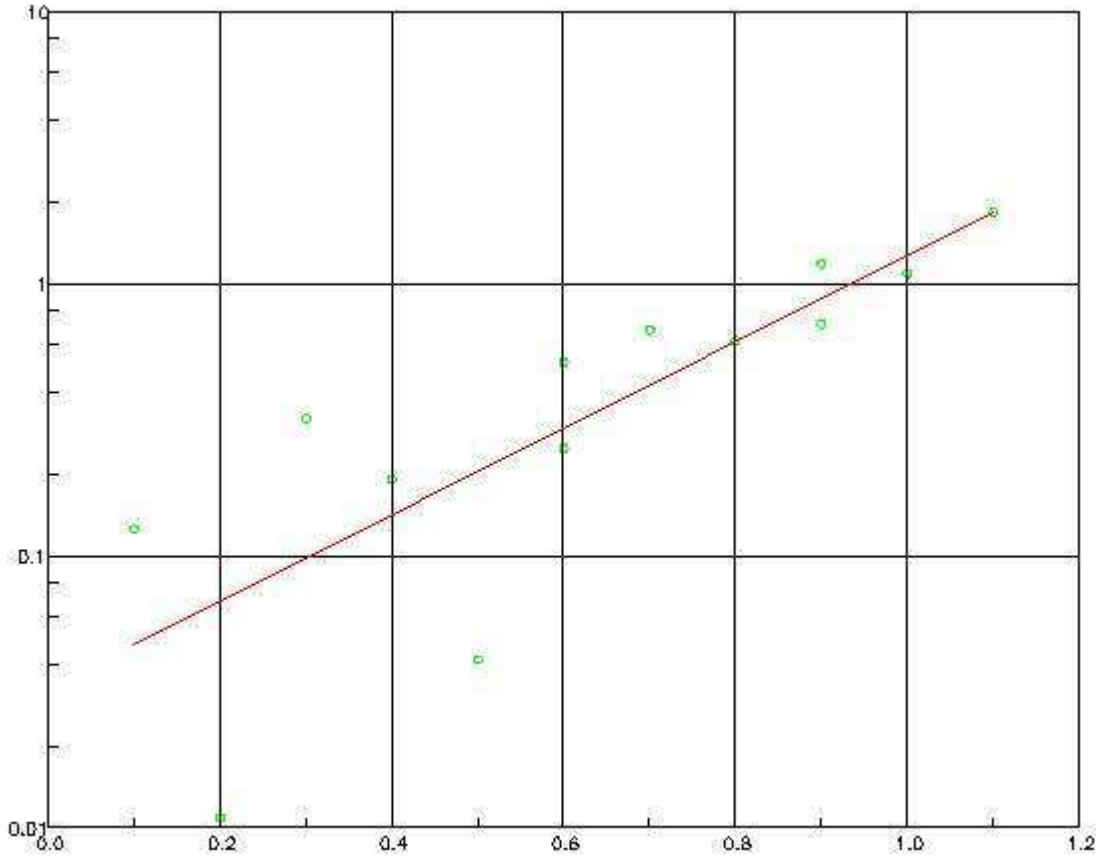
Execute

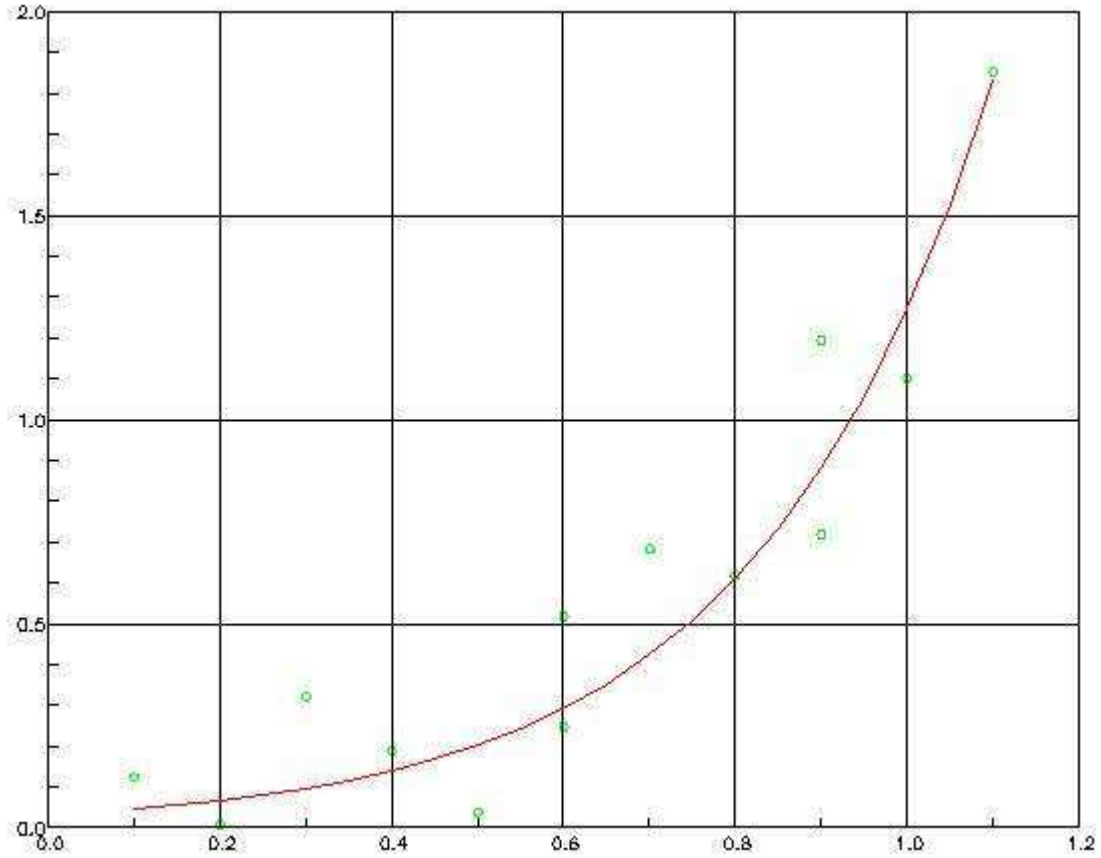
```
FIT LOG XYTEST
```

Similarly, `POWER` finds a best fit by transforming your data to log-log coordinates and fitting a straight line to the result. Then it transforms back again. The `LOGLOG` function does much the same thing, but omits transforming back. Instead, it signals `FIT` to plot the result on logarithmic axes.

For example, the next two graphs show two ways of displaying the best fit of an exponential function to the data in `XYTEST`. They were produced in succession by these statements:

```
tm←1 1
FIT LOG XYTEST
CLEAR
FIT EXP XYTEST
```



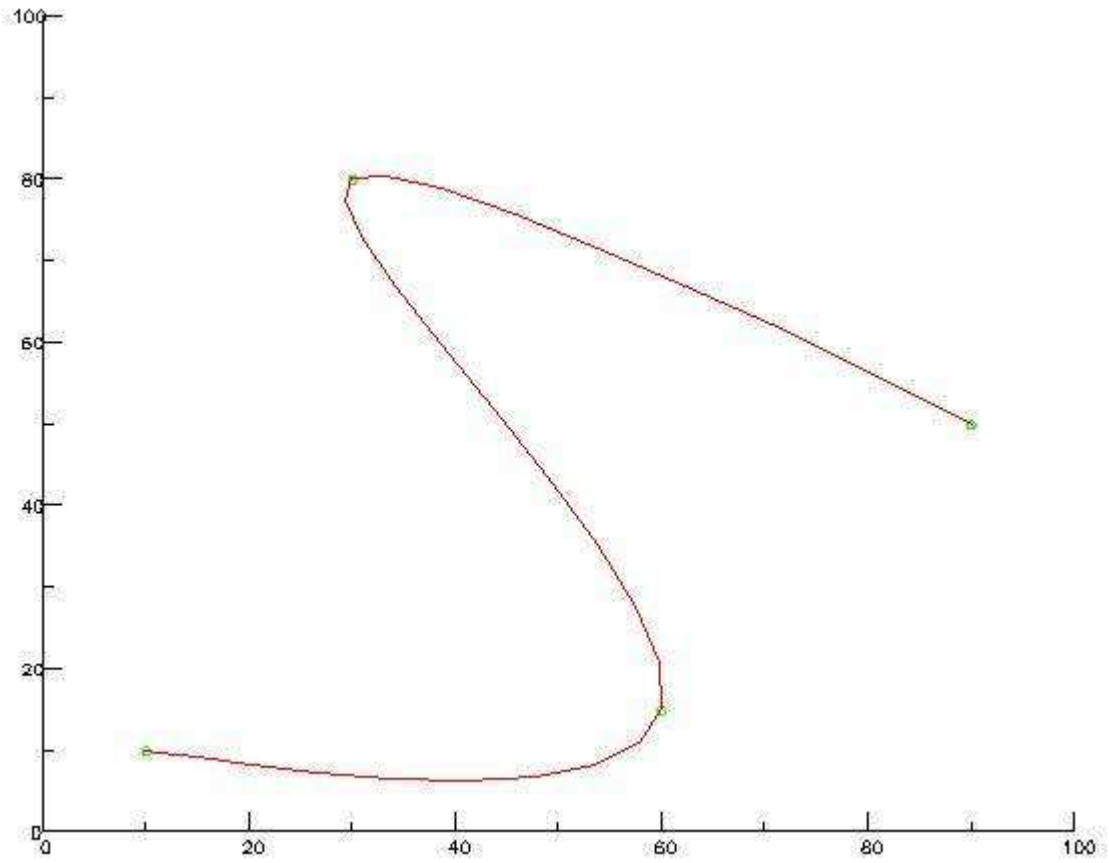


Fitting a Spline Curve

You can also fit to a set of points a curve that is not an approximation, but goes through every point exactly and smoothly. These curves are called *spline curves*. They have the following properties:

- They are made of connected segments, each of which can be described by a rational cubic polynomial.
- Where two segments join, they have the same tangent and curvature. (Or, if you like, their first and second derivatives are the same.) This is the sense in which the curves go through points "smoothly".
- They can be made to have specified tangents at their end points.

The following is a spline curve through the points (10,10), (60,15), (30,80), and (90,50).



SPLINE Fits a Spline Curve

As you might expect, there is a new function to fit spline curves. The example was produced by the following statements:

```
XY←4 2⍴10 10 60 15 30 80 90 50
FIT SPLINE XY
```

If you execute those statements, you will first be asked to enter more data. The example produced the dialog shown below. In each statement, the part following the colon (:) was entered by the user.

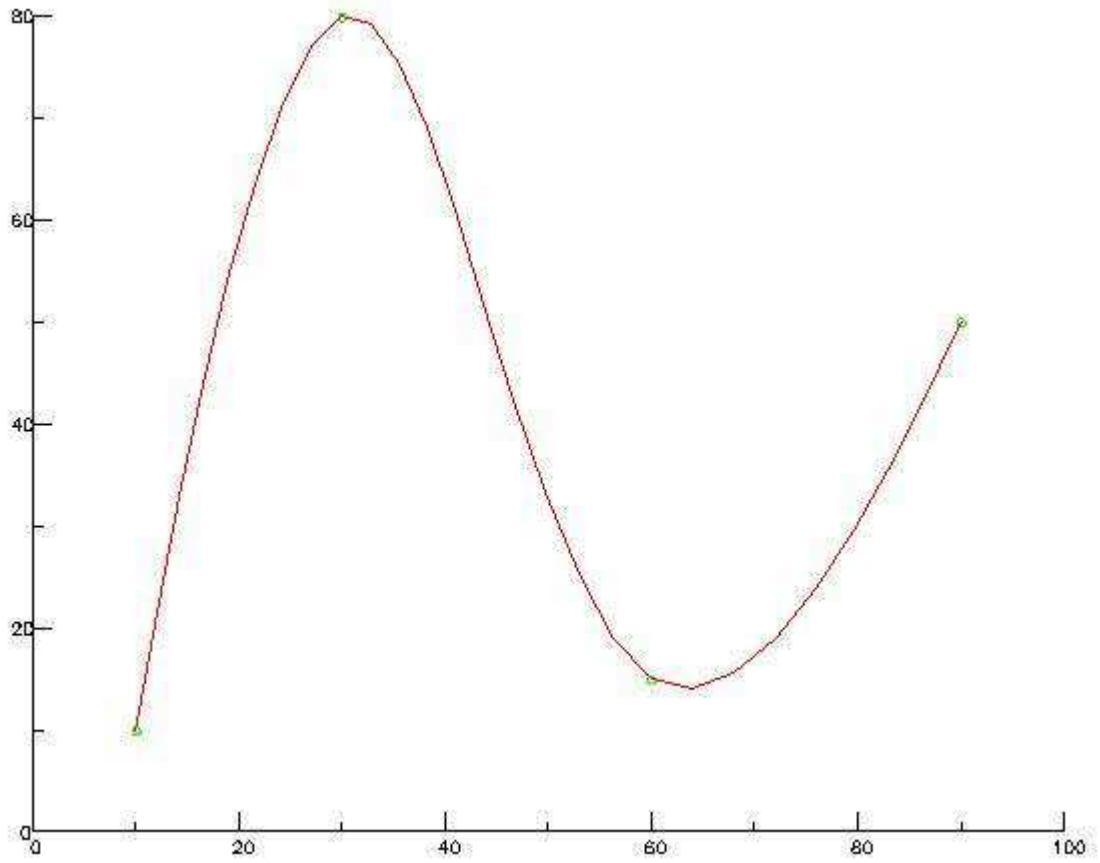
```
SOURCE END CONDITION: FIX 0 1
  SINK END CONDITION: FIX 1 ¯1
   STEPS PER SEGMENT: 10
    4 CONTROL COEFFS: 0
```

Some of the options that can be entered in response to these requests for data are described on the following pages. For a complete description of the technique of fitting spline curves by SPLINE, see D. V. Ahuja, *An Algorithm for Generating Spline-like Curves*, IBM Systems Journal, Vol. 7, Nos. 3 and 4, 1968.

Structuring Data for a Spline Curve

There is one important difference between the data structures for SPLINE and for AVG, SL, and the others. For spline curves, *the order of the points makes a difference*. For example, the graph below shows a spline curve through the same four points as in the previous graph, but in the order (10,10), (30,80), (60,15), (90,50). Each point is in one row of the matrix that is the argument to SPLINE, and the order of the points is the order of the rows.

There is also a minor difference: You need a minimum of four points to fit a spline curve by this technique.



Additional Data for SPLINE

SPLINE requests that you enter additional data, as shown in the example. You can make the following kinds of answers:

- To `SOURCE END CONDITION:` Answer

  `FIX` *m n*
  to specify the 2 direction components of a line that must be tangent to the curve at its beginning.
  `FREE`
  to indicate that the curve has zero curvature at its beginning.
  `CYCLIC`
  to indicate that the tangent and curvature must be the same at the beginning and end points (as would be the case, for example, in a smooth closed curve).

- To `SINK END CONDITION:` Answer with any of the options you can use for the first request. You will be specifying conditions for the end point of the curve. (If you answered `CYCLIC` to the first request, this one will not be made.)
- To `STEPS PER SEGMENT:` Answer with the number of values to be plotted in each segment of the curve. (Each pair of adjacent data points determines a segment.) The more values per segment, the smoother the curve, but also the longer the calculation time.
- To *n* `CONTROL COEFFS:` Answer 0.
  (The technique used allows you to enter up to *n* coefficients with which you can experimentally vary the curve produced - to attempt to eliminate a cusp, for instance.)

Results of SPLINE

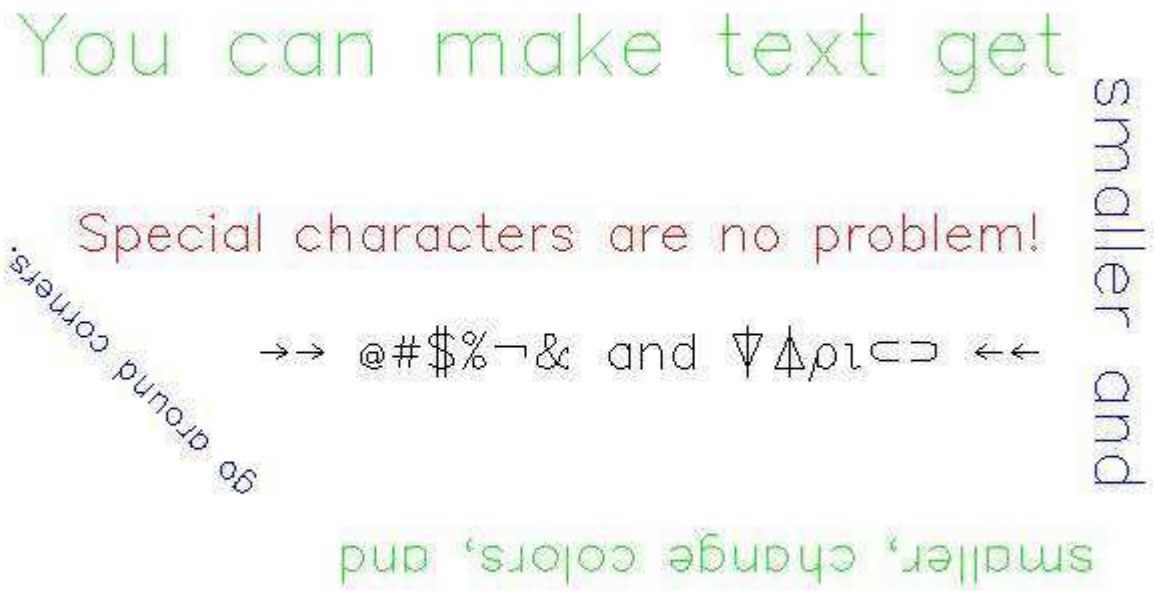
As the examples show, `SPLINE` can be used with `FIT` in the same way as `AVG`, `SL`, and the others can. But it has somewhat different effects on `fitmsg` and `FITFUN`:

- After using `SPLINE`, `fitmsg` will contain a statement that invokes the function `SPL`, which is the same function that `SPLINE` invokes to generate the spline curve.
- And `FITFUN` will contain a set of points from which you can plot the curve. For example, `PLOT FITFUN 0` will plot the same graph.

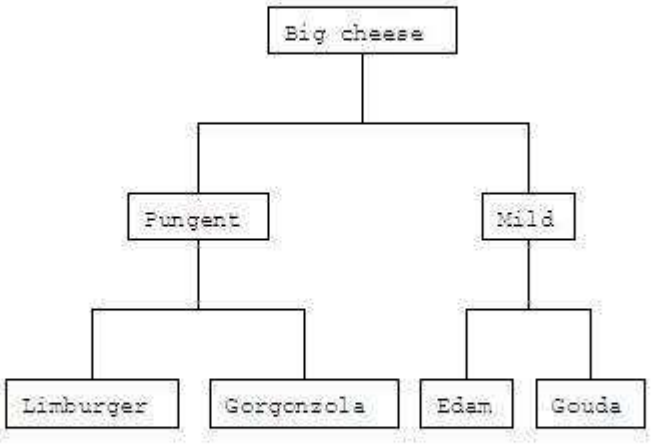
Writing Text and Drawing Flat Pictures

This section tells you

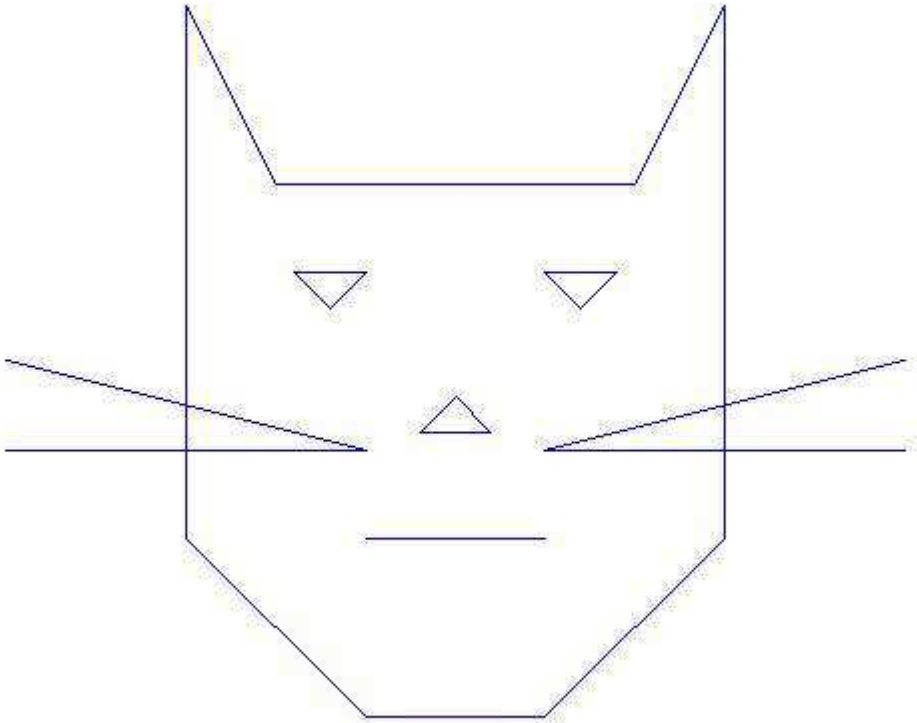
- how to write text on the screen just about anywhere..



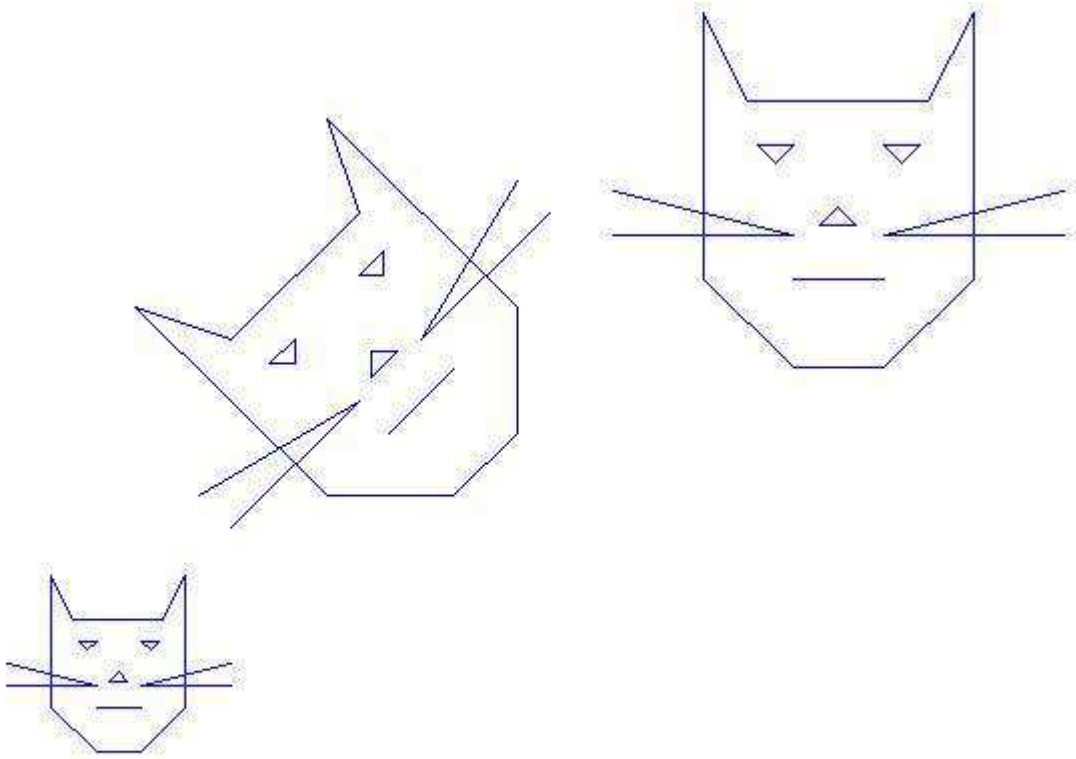
- how to draw hierarchic diagrams or organization charts...



- how to draw pictures on your display screen and reproduce them at will...



- how to fill, magnify, translate, and rotate pictures...



- how to handle writing in three-dimensional space.

Writing Text

In this context, *text* is any string of APL2 characters. Anything else comes under the heading of drawing pictures. Note that the first 64 characters and the last character of `⎕AV` may be interpreted as control characters by some devices.

What you can do in the way of writing characters on your display screen depends on your output device and on the interface to it. GRAPHPAK allows three different ways of writing characters, and what choice is in effect is governed by a value in `dd[9]`. To execute the examples in this section, begin by executing

On workstation systems:

```
OPENGP ⍝ If the graphics window is not open.
VFONT←'GPAKFIGSFONT'
'ROMSIM.AVF' FONTDEFINE VFONT
dd[9]←2
```

On CMS and TSO:

```
dd[9]←2
```

See the discussion in Character Generation to see what differences other choices would make. (`RESTORE` sets `dd[9]` to 0; that's faster, but not as flexible.)

Earlier sections of this manual describe functions that write text for special purposes - labels, annotations, titles, and the like. For more general purposes you need a new function.

WRITE Writes Text

`WRITE` takes two arguments:

- The right argument contains characters that make up the text.
- The left argument is a vector of control values.

For example, execute

```
5 30 3 WRITE 'THIS STARTS AT (5, 30) AND IS 3 UNITS HIGH'
```

Arranging the Text for WRITE

Your text can be a vector (as in the preceding example) or a matrix. If the text is in a matrix, each row of the matrix is a separate line of text.

The Left Argument for WRITE

This is a vector or matrix of control values, with which you can

- Start the text anywhere. The first two values in a row are the coordinates (in virtual space) of the lower left corner of the string of text to be written.
- Specify the size of the characters. The third value gives the height of each character in virtual-space units.
- Slant the text at any angle. The fourth and fifth values do this. If the fourth value is 0, the fifth value is the angle (in degrees) that the line of text will make with the horizontal. If the fourth value is 1, the fifth value is the angle the text makes with the vertical.
- Change the attribute values of the characters. The sixth value, if present, is used as an index into the attribute vector `av`.

To see some of the possible variations, execute

```
50 10 2 0 90 1 WRITE 'GOES STRAIGHT UP'
50 10 2 1 0 2 WRITE 'SO DOES THIS'
50 50 2 0 ¯90 3 WRITE 'GOES STRAIGHT DOWN'
50 50 2 0 180 4 WRITE 'GOES UPSIDE DOWN'
```

In the simplest case, the left argument has as many rows as the right argument has lines of text, and each row controls one line. If the left argument has several rows, and there is only one line of text, the one line will be duplicated for each row of the left argument. If the left argument has one row, and there are several lines of text, the left argument specifies the location of the lower left corner of the block of text.

Setting Things Back Again

Using `WRITE` to write characters of different sizes will change values in the global variable `dd`. `RESTORE` sets these back to initial values, but it may reset more than you want. (For example, it sets `dd[9]` to 0.) To restore just the size of characters, execute

```
0 0 1 WRITE ''
```

(What size this restores characters to depends on the current value in `dd[9]`.)

Changing Color, Style, and Width

You can use `USING` with `WRITE`, as you did with `TITLE`. Execute, for example,

```
10 30 4 WRITE 'ANYTHING YOU LIKE' USING COLOR 1
```

The effect of `USING` is temporary: the graphic output following this display will be rendered with the attribute values previously in force.

`USE` Changes `av`

You can also make a global change to the attribute vector (`av`).

```
USE COLOR 4, STYLE 2, WIDTH 2
```

for example, will assign to av a single attribute code. Until you change av again, anything else you WRITE (or DRAW or FILL) with default attribute values or indexes into av will appear with color 4, style 2, and width 2.

Entering Several Lines of Text

If you are entering several lines of text, you may find it cumbersome to count blanks, catenate successive lines, and shape the whole into a matrix of proper size. An option of the READ function does all this in one operation.

Executing

```
TEXT←6 READ 3 0 10 50
```

does the following:

1. It first positions the cursor at the beginning of the next line on your APL screen, so that you can enter text. End each line by pressing Enter.
2. It stops the operation after you have entered 6 lines. (To enter a variable number of lines, make the left argument of READ a null vector. The operation will stop when you enter a null line.)
3. It writes the text you have entered on your display. The lower left corner of the first character of the first line is at (10, 50).
4. It saves the text matrix in the variable TEXT.

If you want to write the same text somewhere else, you could now execute, for example,

```
30 30 2 0 30 WRITE TEXT
```

The right argument for READ

The values in the right argument of READ in the previous example have the following meanings:

3	means option 3 of the READ function. The other two options are described under DRAW Draws Pictures.
0	means sounding the alarm on the output device to warn you that input is needed.
10 50	give the coordinates of the lower left corner of the first character of the first line on the display screen.

Drawing Hierarchic Charts

The cheese chart in Writing Text and Drawing Flat Pictures is a hierarchic chart. It shows the subordination of some elements to others. It's a very special kind of picture, and something of a diversion in this discussion, but creating one makes good use of the `READ 3` function.

To prepare data for the figure, execute

```
      CHEESE←'' READ 3 0 10 50
```

Then enter the text that will appear in the diagram, as it is shown here. Don't ignore the indentations: they do all the work.

```
Big Cheese
  Pungent
    Limburger
    Gorgonzola
  Mild
    Edam
    Gouda
```

End by entering a null line (press Enter twice).

HCHART Draws Hierarchic Charts

To draw the figure, execute

```
      '' HCHART CHEESE
```

There are other forms that the right argument of `HCHART` can take, but in all cases it is the blank characters at the beginnings of the lines that tell when a line is subordinate to some line above it.

Characters in the left argument allow you to fill the boxes in the diagram, select other fill patterns, and change the style of connecting lines.

The size of the diagram depends on the current setting, in `dd`, of the size of characters. You can change this by using the `WRITE` function.

Drawing Pictures

If you have been executing the examples in this manual, you should by now be thoroughly convinced that a keyboard and a display screen are not the same thing as a pencil and paper. The keyboard and screen will do many of the same things you can do with pencil and paper, but it doesn't do them in quite the same way. On the other hand, the computer will do very easily quite a lot of things you can hardly do with pencil and paper at all. So when you come to wanting to draw pictures, you are perhaps ready to approach the subject by a somewhat roundabout route; you know it will make things easier in the end.

In order to draw pictures on your screen, there must be a way to represent pictures by numeric data in GRAPHPAK. Then there ought to be a convenient way to enter that data. Then at last you could think about changing the data to draw the picture differently - larger, smaller, in different colors, and so on.

The pages that follow describe this approach to drawing pictures:

1. READ Options that Read Cursor Positions tells how you can draw a picture by moving your cursor around on your display screen and recording its position at the end of each move.
2. Structuring Data for a Picture tells what the resulting data looks like in your workspace, and how it represents lines of a picture.
3. DRAW Draws Pictures introduces the functions that draw the picture again.

READ Options that Read Cursor Positions

The function that records several positions of the cursor is option 1 of `READ`. Option 0 of `READ` reads only one position of the cursor. (Option 3 has been described earlier, under Entering Several Lines of Text. Option 2 is reserved.)

READ Reads More than One Cursor Position

For a model of `READ` option 1, execute

```
PICTURE←'' READ 1 0 2
```

This will position the cursor in the middle of your display screen. (Use `VIEW` if you have to.) Now do this:

1. Move the cursor to a point where you want to start a drawing. (Or leave it alone if you're content to start at the middle of the screen.)
2. Press any function key (or a mouse button on workstation systems.)
3. Move the cursor somewhere else.
4. Press any function key or mouse button. Unless something has gone horribly wrong, you have drawn one line of a picture.
5. Repeat the last two steps as often as you like.
   Different function keys and mouse buttons may give different combinations of color and style. What the combination is depends on the current setting of the attribute vector (`av`). You may also find a key that seems to do nothing: it draws a line with the color of the background - black on the display screen, white on the printer. (On the IBM 3279, with the default setting of `av`, it is function key 8.)
6. When you've drawn enough points, press Enter twice.

You now have a drawing on your screen. The variable PICTURE contains a numeric representation of it in your workspace. What that looks like is the topic of Structuring Data for a Picture; for the moment, consider what READ did.

The right argument

The numbers 1 0 2 in the model did these things:

1 named option 1 of READ.

0 specified sounding the alarm whenever you pressed a function key.

2 drew lines. (1 would have meant to draw only points; 0 would have recorded the location but drawn nothing.)

To start the cursor at somewhere other than the middle of the screen, give the coordinates of the starting point as fourth and fifth elements of the right argument of READ 1.

The left argument

The null value specified that you would be through entering points only when you entered a null line (which you did by pressing Enter twice). To stop after *n* points, use *n* as a left argument. (But you will also stop when you enter a null line.)

READ Reads One Cursor Position

Option 0 of READ returns the location the cursor had at the last time the graphics processor received an event. The processor receives an event when you use VIEW. To use READ 0:

1. Execute VIEW.
2. Position the cursor at any point of the graphic field.
3. Press Enter (or any key or mouse button on workstation systems.)
4. Return to the APL screen and execute

```
5.
6.          LOC←0 READ 0
```

Following this, LOC will contain the two coordinates (in virtual space) of the cursor location.

Structuring Data for a Picture

If you executed the READ 1 example, look now at PICTURE in your workspace. It is a 3-column matrix, with each row representing one recorded cursor position. The second and third elements in the row are the coordinates of the cursor position; the first element is the number of the function key with which you recorded the position.

The successive rows of PICTURE record also the order in which you drew the lines or points of your picture. Each row after the first records the movement of the cursor to a new point on the screen. Or, it represents a line from the previous point to the new point. (The first row does not represent a line, but an invisible movement of the cursor to the first point of your picture.)

You may have the last point twice

If you used the function keys to record cursor locations, and then used the Enter key twice to stop recording, the last row of your matrix is the same as the one before it. If you have an extra last row, just throw it away; execute

```
PICTURE←¯1 0↓PICTURE
```

More about the first elements

The elements in the first column of PICTURE are attribute codes for the lines. Used as is (for example with DRAW, described later) they are indexes into the attribute vector (av). Thus, a line beginning with the number 5 will be drawn with the fifth set of attribute values.

But you can also make these changes:

- To erase a line, make its first element 0. (The row now describes an invisible movement of the cursor to a new point.)
- To change the attribute codes for a line, change its first element. For example, to change the color and style of the fourth line of PICTURE, execute
-
-
  ```
  PICTURE[5;1]←COLOR 7,STYLE 3
  ```

  (Row 5 describes the fourth line of PICTURE. Row 1 describes an invisible move of the cursor to the starting point, and its first element is not used.)

If you don't have the first elements

You may also have entered data for a picture in some other way than by READ, and that way may not give the first column of indexes into AV. You may not need the first column. You can use many of the functions described next with a 2-column matrix of coordinates only. When you draw the results, you will use default attribute values.

Or, you can attach a first column of attribute codes. For example, execute

```
M←M USING COLOR 3
```

DRAW Draws Pictures

Now that you have entered data describing a picture, assigned to it whatever attribute codes you want, perhaps erased some extra lines, you should be about ready to have the computer make things easy for you.

To draw your picture again, execute

```
DRAW PICTURE
```

To draw it with the same attribute values for every line, execute

```
      DRAW PICTURE USING COLOR 2, STYLE 3
```

If you have been executing all the previous examples, this is the easiest thing you've done in some time. It would be a pity to leave it for something more complicated quite so soon. Why not try these?

```
      DRAW STAR
      DRAW BLIVET
```

STAR and BLIVET are in your workspace mainly to serve as examples for DRAW. Look at the data in STAR to see how simple it is to describe a figure made up of straight lines. (For example, it omits the column of attribute codes.) Look at BLIVET to see how the data for curves must be recorded.

Demonstration functions in GRAPHPAK that draw pictures include APPLE, BLI, FLAG, and FSTAR.

Drawing Smooth Curves

If you looked at the data in BLIVET, you may have been dismayed at the prospect of entering so many points one at a time. You don't always have to do that in order to get a smooth curve. Instead, you could, just for example,

- Generate points by a function
- Draw a smooth curve through a few points using FIT SPLINE

Examples of these two methods follow.

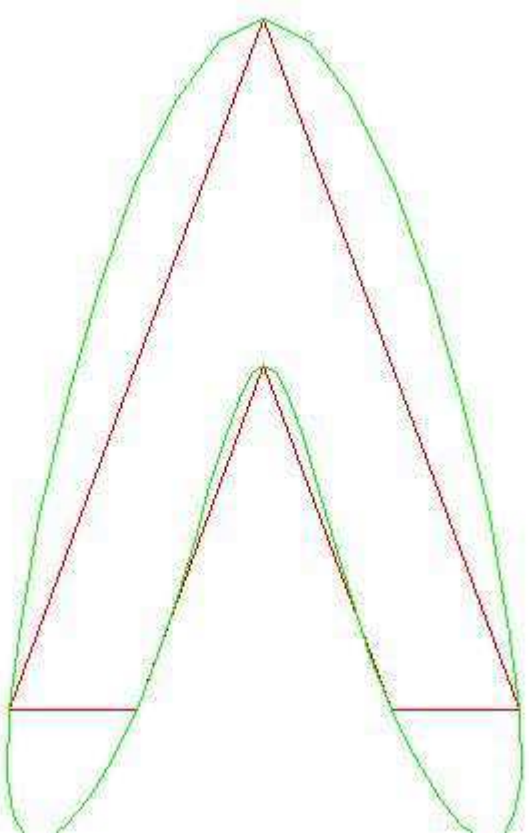
Generating Points by a Function

The following function is a suggestion of what you might do to generate figures. It does not exist in GRAPHPAK.

The function combines option 1 of READ with PIECHART, to generate a circle. Executing the function allows you to enter two points with your cursor. The first point is taken as the center of the circle, and the distance to the second point is taken as the radius. PIECHART draws the circle.

```
      ∇CIRCL;A;pie
[1]   ⍝ Read center and point on circumference
[2]   A←2 ¯2↑2 READ 1 0 1
[3]   ⍝ Set center and radius in pie
[4]   pie←A[1;],(+/(-⌿A)*2)*0.5
[5]   '' PIECHART 1
      ∇
```

Drawing a Smooth Curve through a Few Points



The figure above shows a pattern drawn with straight lines, with a smooth curve drawn through the intersections. To draw something like it, do this:

1. Execute

```
FIG←'' READ 1 0 2
```

   This puts the cursor in the middle of the graphic field allows you to enter the straight-line pattern. As before, you finish by pressing Enter twice.

4. (If you have an extra row on `FIG` now, drop it.)
5. Execute `FIT SPLINE` as shown below. You can use a different number of steps per segment (and other control coefficients).

```
      FIT SPLINE FIG[;2 3]
SOURCE END CONDITION: CYCLIC
   STEPS PER SEGMENT: 8
    9 CONTROL COEFFS: 0
```

11. When this has completed, you can view the result. (It will probably not be the same size as your original figure.)

    If you're not satisfied with the shape of the curve, go back to step 1 and put some more points in the pattern.

12. When you have something you like the looks of, execute

```
14.        SMOOTH←FITFUN 0
```
15. SMOOTH is now a 2-column matrix, containing coordinates of points on the smooth curve. Attach a first column of attribute codes to it if you wish.
16. Execute
```
17.
18.        DRAW FIG
19.        DRAW SMOOTH
```

FILL Fills Polygons

A picture on your screen is made up of straight line segments. Therefore it is a polygon, and can be filled with a solid pattern. For an example, execute

```
FILL STAR
```

FILL uses the same kind of data structure as DRAW does. If the picture has a first column of attribute codes, FILL uses them to determine the fill colors and patterns. (The style value in the attribute code selects a fill pattern.)

If you executed the example, you may have wondered why the center of the star was not filled. The filling technique scans the figure with horizontal lines. As the scanning line crosses a boundary, it turns the fill pattern on if it was off, and off if it was on. Look at the boundaries of STAR (by executing DRAW STAR, for example); you will see that for any horizontal line to penetrate to the center of the figure, it has to cross two boundaries. (The point of intersection of two boundaries counts double.) Hence, to FILL, the center of this figure is outside the polygon it is filling.

FILL requires a closed curve. If your last data point is not the same as the first, FILL will draw an extra line from the last point to the first to close the figure.

If you superimpose one filled figure on another, some of the fill pattern of the first may show through the second (depending, of course, on what kind of fill patterns you use). Whether or not you allow this to happen is controlled by the global variable sf.

Magnifying, Translating, and Rotating Pictures

With the numeric representation of a picture in your workspace, you are almost ready to transform it in as many ways as a photographer transforms a snapshot with a photo enlarger.

Why almost?

`STAR`, for example, is just a set of coordinates in the workspace. If you wanted to shrink the star picture to half its size, what could be easier than multiplying `STAR` by 0.5?

Try it by executing `DRAW STAR×0.5`.

The result may be a little unexpected. The star shrinks, as it ought to, but it also moves off toward the lower left corner of your screen. That is also what it ought to do, but possibly not what you intended. Before trying any more transformations, pause a bit to consider coordinate systems.

Using Problem Space

The variable `STAR` in your workspace represents a picture by a set of coordinates in virtual space. The origin of virtual space - point (0, 0) - is at the lower left corner of your screen, and probably not even inside your scaling viewport. If you divide all the coordinates in `STAR` by 2, and then `DRAW` the result, naturally the picture shrinks toward the origin.

At this point you can either resolve to allow for the location of the origin when calculating transformations, or you can switch to a coordinate system whose origin is where you want it (in the center of the screen, for example). Changing coordinate systems is probably easier, but it involves you in some more discussion of apparatus before going on with drawing pictures.

Changing Coordinates

The new coordinate system you want to change to will define a window in your problem space. You get to choose the window: You can determine its size, its shape, and the location of its origin.

Defining a Window in Problem Space

If you have no particular size or shape in mind, consider the function that follows. It will create a window for you so that

- The window has the same size and shape as the scaling viewport.
- You can pick the origin of the window by moving the cursor.

This is the function definition; it does not already exist in GRAPHPAK.

```
      ∇ CENT
[1]   ⍝ Makes the window equal in size and shape to the
[2]   ⍝ scaling viewport, with a center at the cursor location.
[3]    w←svp-4⍴0 READ 0 1
      ∇
```

To execute CENT:

1. Move the cursor in the graphic field to the point where you want the origin of the window. (Probably this is the center of your picture.)
2. Press Enter. (This records the cursor location and returns you to APL.)
3. Enter CENT.

This defines a window in problem space.

INTO Defines a Transformation

You have a picture drawn on the scaling viewport; its coordinates are in virtual space. You have defined a window in problem space. You next need the definition of a transformation that will map the scaling viewport into problem space.

INTO does that. Execute

```
DS←svp INTO w
```

The result of svp INTO w is a data structure that defines the transformation you need. (Probably you don't care what that structure looks like. If you do, look up INTO in the Function Reference.)

XFM Performs a Transformation

You have a picture drawn on the scaling viewport. You have defined a transformation from the scaling viewport to a window in problem space. You next need to perform that transformation.

XFM does that. Execute

```
WINDOWPIC←DS XFM PICTURE
```

(DS is the result of INTO. All in one step this reads,

```
WINDOWPIC←(svp INTO w) XFM PICTURE
```

Or, Transform PICTURE into WINDOWPIC with the same transformation that maps svp into w.

Results of the Transformation

You now have in your workspace two representations of a picture. It is worthwhile to emphasize their differences:

- PICTURE uses virtual-space coordinates. You can draw it with DRAW.
- WINDOWPIC uses problem-space coordinates. Don't draw it with DRAW - it won't give what you expect.

SKETCH Sketches Pictures in Problem Space

But of course you need a function to draw pictures whose coordinates are in problem space. The function is SKETCH.

For 2-dimensional objects, SKETCH works much like DRAW. Execute

```
SKETCH WINDOWPIC
```

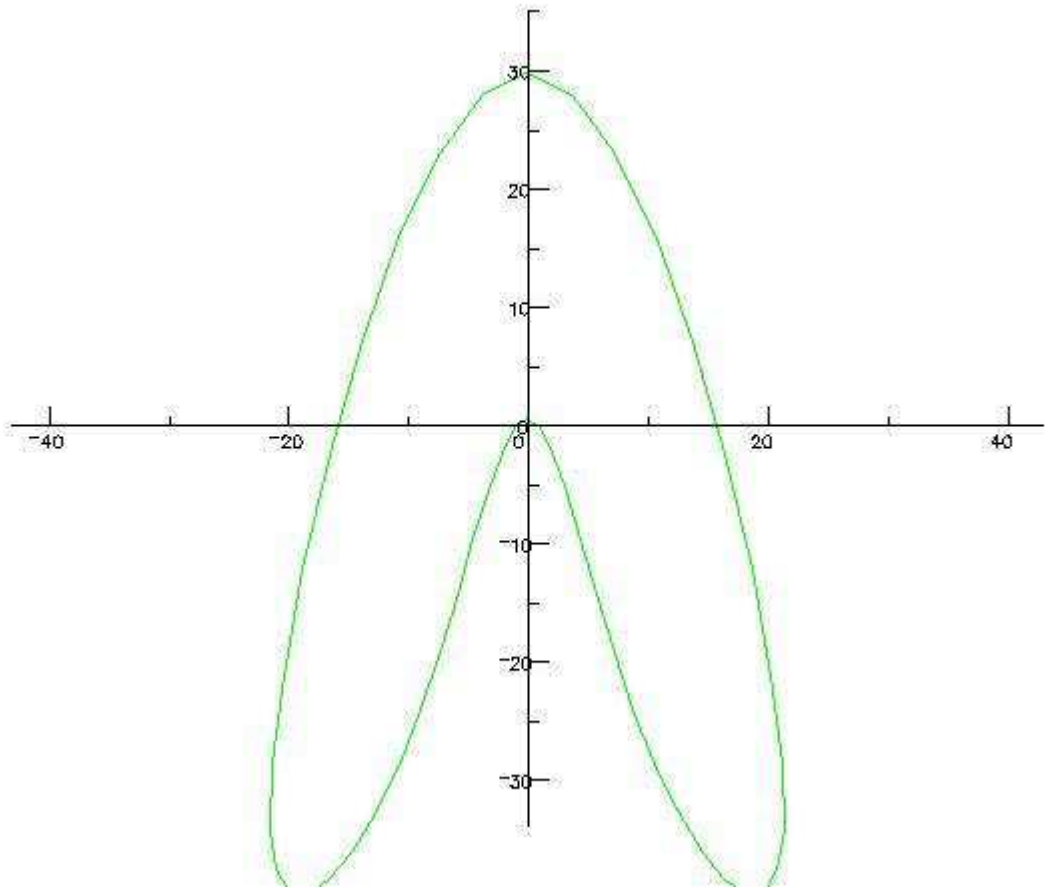
Unlike DRAW, SKETCH absolutely requires a first column of attribute codes. (SKETCH will do things with 3-dimensional objects that DRAW will not do; they are discussed in Drawing Three-dimensional Objects.

Using the New Coordinates

The following figure shows how little has happened. It uses SMOOTH, the curve that was drawn in Drawing Smooth Curves. The new figure was produced by

1. Executing CENT, to choose the origin for a window in problem space.
2. Executing

```
      WINSM←(svp INTO w) XFM SMOOTH
      SKETCH WINSM
      AXES
      LABEL
```



It doesn't look much different. It shouldn't. It's supposed to look exactly the same, except for the axes. But they are important. They emphasize that you now have the picture you want in a coordinate system you have better control of.

Now back to transforming pictures

With a picture represented in problem-space coordinates, and a way of sketching that picture, it's natural to transform the picture by performing mathematical operations on the coordinates. But it's a bit of a nuisance to do this while leaving unchanged the first column of the data, which contains attribute codes.

Some new functions take care of this.

MAGNIFY Magnifies and Shrinks Pictures

Your picture is represented in `WINDOWPIC`. To magnify it by a factor *m*, execute

```
SKETCH m MAGNIFY WINDOWPIC
```

If *m* is less than 1, your picture shrinks.

If *m* is a 2-element vector, your picture will be magnified in the X-direction by the first element and in the Y-direction by the second.

TRANSLATE Moves Pictures

To move your picture on the screen by *x* units in the X-direction and *y* units in the Y-direction, execute

```
SKETCH x y TRANSLATE WINDOWPIC
```

The units are units of distance in problem space.

ROTATE Rotates Pictures

To rotate your picture through an angle *a*, execute

```
SKETCH a ROTATE WINDOWPIC
```

The angle is measured in degrees, and the rotation is counterclockwise.

In this example, `WINDOWPIC` is in problem-space coordinates. They have an origin at a point you chose. When `WINDOWPIC` was rotated, it circled around that origin.

Filling a Transformed Picture

If you have been executing the examples in this section with a picture of your own, you are by now being very careful about whether the data for your picture uses problem-space coordinates or virtual space coordinates. You probably have two sets of data, as the examples here have `PICTURE` and `WINDOWPIC`.

You DRAW one set of data (PICTURE) and SKETCH the other (WINDOWPIC). You could MAGNIFY, TRANSLATE, and ROTATE both sets of data, but only one set puts the result on the screen where you want it (WINDOWPIC). And you could FILL both sets of data, but only one set will give you expected results (PICTURE).

How to Fill the Transformed WINDOWPIC

FILL works on virtual-space coordinates. You have been magnifying, translating, and rotating something in problem-space coordinates. To FILL it, you have to transform back again.

For example, the statement

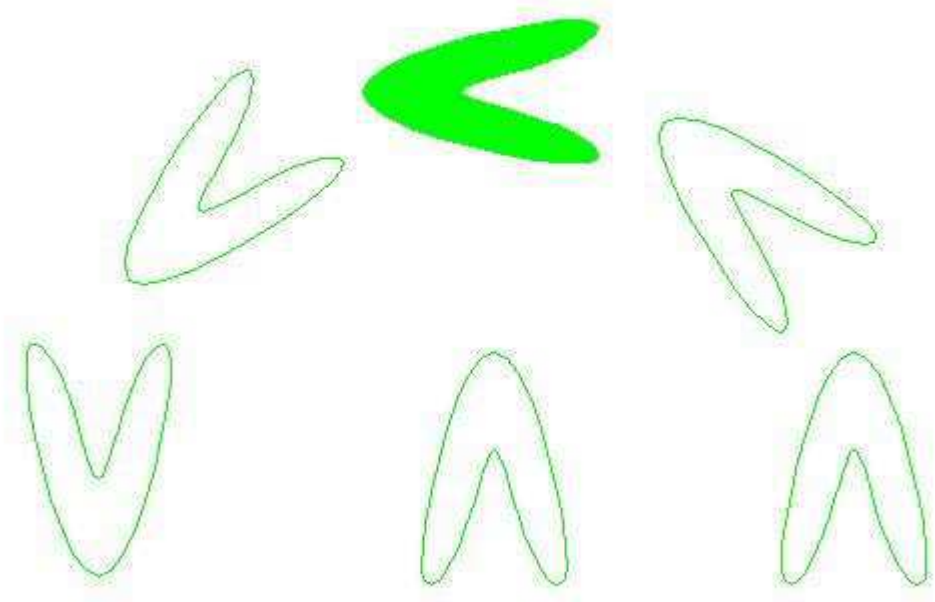
```
FILL (w INTO svp) XFM 90 ROTATE 15 0 TRANSLATE SMOOTH
```

does this:

1. Translates SMOOTH 15 units to the right.
2. Rotates the result through 90 degrees.
3. Transforms the result of that by the same transformation that maps the window into the scaling viewport. (This gives virtual-space coordinates of the object that was translated and rotated.)
4. Fills the object.

Putting it All Together

The next figure might be fun as a puzzle. Could you find the steps that created it? But its purpose is not to puzzle you. It shows that by using the functions described in this section you can put a figure you have drawn anywhere on the screen in any orientation.



The figure was produced by these statements.

```
TINY←0.3 MAGNIFY (svp INTO w) XFM SMOOTH
SKETCH TINY
SKETCH TINYT←15 0 TRANSLATE TINY
SKETCH 45 ROTATE TINYT
SKETCH TOP←90 ROTATE TINYT
FILL (w INTO svp) XFM TOP
SKETCH 135 ROTATE TINYT
SKETCH 180 ROTATE TINYT
```

Writing in Three-dimensional Space

Some of the functions in GRAPHPAK map a 3-dimensional space into the 2-dimensional window. Skyscraper charts do this, for example. Suppose you had plotted such a chart (using the function `SS`) and wanted to write some text on it at a point that corresponded to a triple of X-, Y-, Z-coordinates.

The problem is to capture the transformation that `SS` uses to map 3-dimensional space into the window. The function `SXFM` provides it.

SXFM Maps From Three-dimensional Space to the Windoow

Suppose `M` is a vector of three elements, giving the coordinates of a point in 3-dimensional space. Then

```
SXFM M
```

is a vector of two elements, giving the coordinates of the corresponding point in the window in problem space

(Or, `M` could also be a 3-column matrix, representing many points. `SXFM M` would be a 2-column matrix of the corresponding points.

To write at the point corresponding to `M`, execute, for example,

```
((w INTO svp) XFM SXFM M) WRITE 'PUT IT HERE'
```

In the example, the statement in parentheses, that is the left argument of `WRITE`, does this:

1. Transforms the point `M` into a point in the window in problem space.
2. Transforms the result by the same transformation that maps the window into the scaling viewport

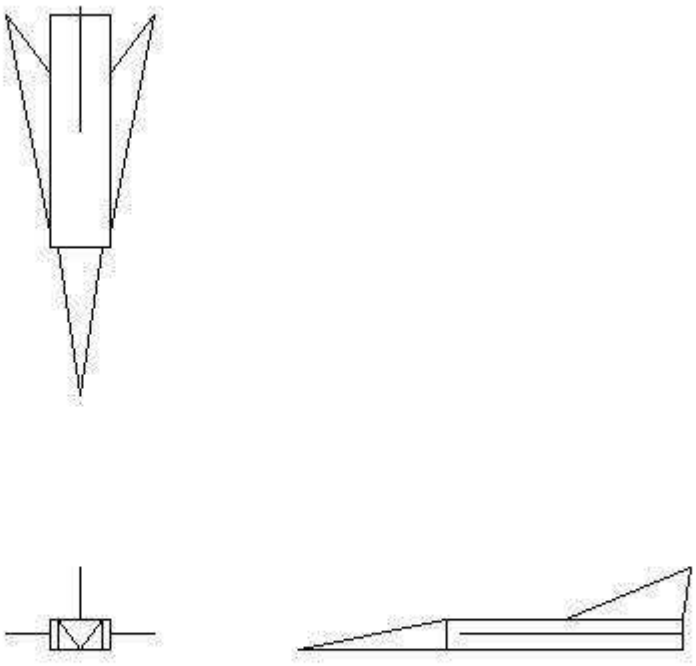
The final result is a point in the scaling viewport.

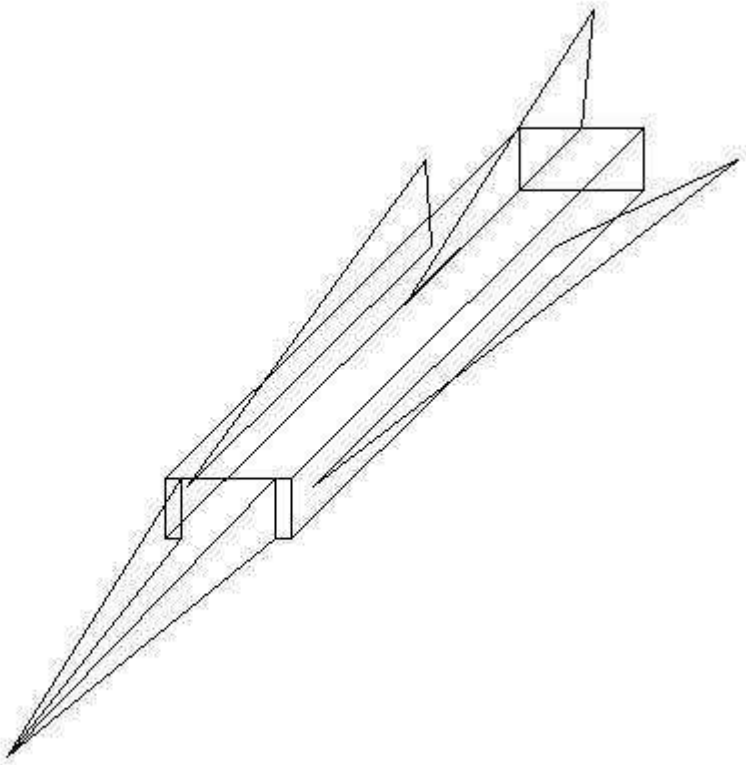
Drawing Three-dimensional Objects

These sections tells how to draw and transform 2-dimensional pictures (projections) of 3-dimensional objects:

- What to Expect
- Structuring Data for a Three-dimensional Object
- SKETCH Sketches Projections on the X,Y-Plane
- Transforming Projections
- Drawing Other Projections

Here are some typical results.





What to Expect

To draw projections of 3-dimensional objects, you need to know about these things:

- A way of representing 3-dimensional objects by a data structure in your workspace
- Functions that draw projections
- Functions that transform projections, by magnifying, translating, and rotating them

If you have read Writing Text and Drawing Flat Pictures, much of this section will be familiar, like drawing pictures. For example:

- Data structures for 3-dimensional objects are like data structures for 2-dimensional pictures, but with one more column of coordinates attached.
- Many of the functions are similar to those described in Writing Text and Drawing Flat Pictures. Some of them are the same - `SKETCH`, `MAGNIFY`, `TRANSLATE`, and `ROTATE`.
- Again you have to be careful about what coordinate system you are working in.

There is one major difference: you do not enter the data for a 3-dimensional object by moving the cursor on the display screen. The display screen is 2-dimensional. The data you enter with `READ` is 2-dimensional. This section assumes that you have some other way to enter data that represents a 3-dimensional object.

Structuring Data for a Three-dimensional Object

The data for a 3-dimensional object must be a 4-column matrix.

- The numbers in columns 2, 3, and 4 of any row are the X-, Y-, and Z-coordinates of a point.
- The number in column 1 of the row is an attribute code used in drawing a line to the point. If the number is 0, it specifies moving the cursor to the point without drawing a line.

(Yes, this is the data structure for a 2-dimensional figure, with a column of Z-coordinates attached.)

The Object in the Examples

In GRAPHPAK there is a description of a 3-dimensional object that is used in all the examples in this section. We have shown two drawings of it. If you have no data of your own, you can re-create the examples in this section by starting with the data in the variable `AP`.

A Note on Coordinate Systems

The data for your 3-dimensional object is a set of coordinates in problem space. Those coordinates have no relationship to points on your scaling viewport until you try to draw a picture of your object.

If you have read Writing Text and Drawing Flat Pictures, this means that you stop working in virtual-space coordinates. All the functions in this section use problem-space coordinates. But if you draw pictures with functions from this section, and then want to fill them or write text around them, you have to think about virtual-space coordinates and the scaling viewport again.

Define a Window

The functions in this section that draw pictures will map the window in your problem space into the scaling viewport. Before going any further, decide what window you want and assign it to `w`.

The examples in this section use the assignment

```
w←¯3 ¯3 3 3
```

RETICLE Shows Your Window

The function `RETICLE` draws an image of the clipping viewport, and a pair of axes centered at the origin of your window. To give yourself a frame of reference for your window, execute

```
RETICLE
```

SCALE Scales Data to Fit

Now you can choose how much of the window your object will take up. The examples that follow begin by making `AP` one-third the size of the window. Execute

```
IM←¯1 1 SCALE AP
```

The left argument of SCALE specifies limits on the size of the object. You can have the same limits on all coordinates, as in the example, or separate limits on each. The example scales the longest dimension of AP to fit between -1 and 1. Its shorter dimensions are scaled down accordingly.

SKETCH Sketches Projections on the X,Y-Plane

By now you must be ready to see what you have. SKETCH sketches a projection of an object on the plane Z=0. This is called an orthographic projection. Execute

```
SKETCH IM
```

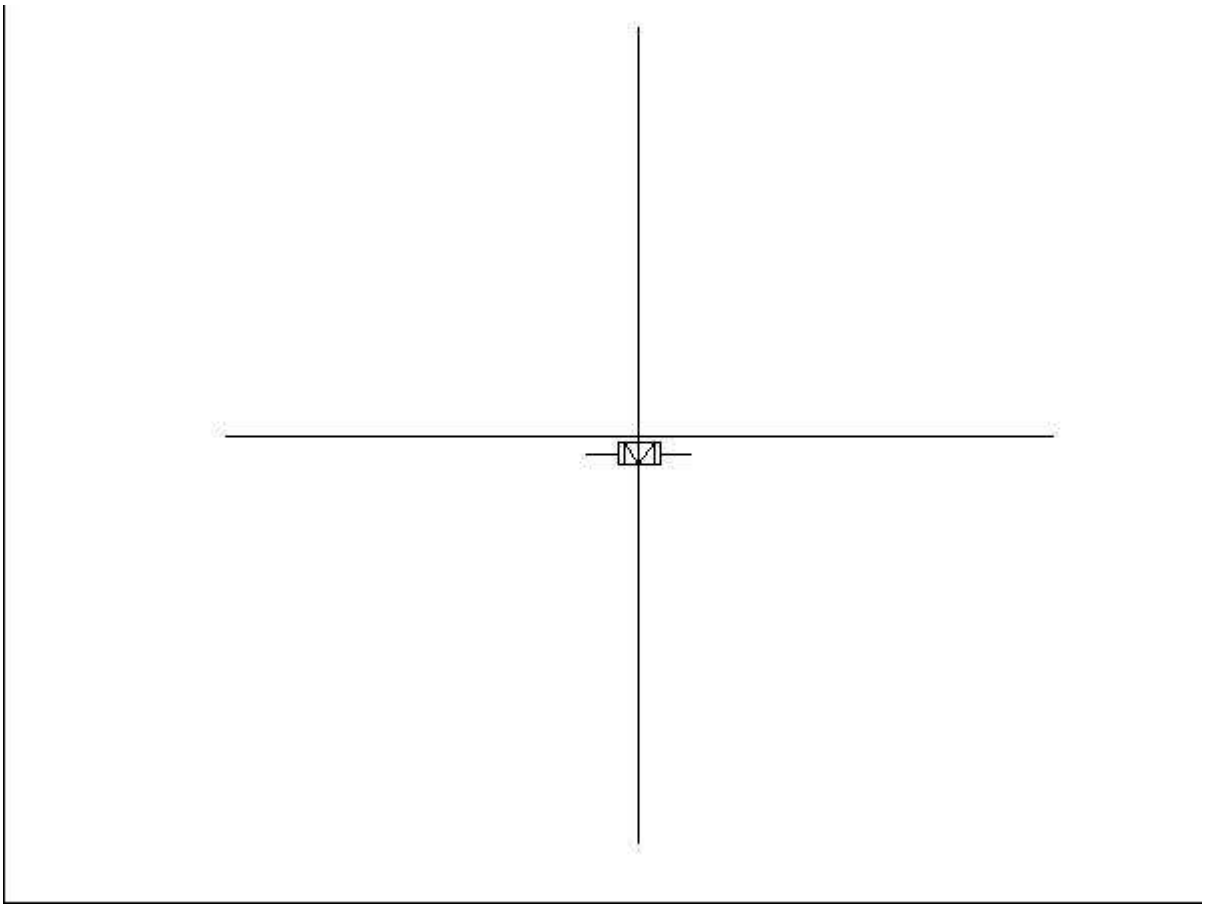
The data here is a scaled down version of AP. It is still a 4-column matrix, with a column of attribute indexes and three columns of coordinates.

The plane Z=0 is the plane of the display screen or the paper. The positive end of the Z-axis comes out of this plane toward you. For another view of the situation, execute the demonstration function SPIRAL.

(SKETCH is the same function that was described in Writing Text and Drawing Flat Pictures. If the Z-coordinates of an object are all zero, or are missing, then the object is a flat picture. SKETCH will sketch it.)

The following figure shows the result of these statements:

```
w←¯3 ¯3 3 3
RETICLE
SKETCH IM←¯1 1 SCALE AP
```



Transforming Projections

If you have read Writing Text and Drawing Flat Pictures, you may recognize some of the functions described here. They have some new features, though: their left arguments allow values for three coordinates.

MAGNIFY Magnifies and Shrinks Objects

```
2 MAGNIFY IM
```

will double the size of `IM` in each dimension.

```
0.5 3 1 MAGNIFY IM
```

will shrink `IM` to half its size in the X-direction, triple its size in the Y-direction, and leave it unchanged in the Z-direction.

TRANSLATE Moves Objects

```
2 0 ¯1 TRANSLATE IM
```

will move `IM` two units to the right in the X-direction, leave it unchanged in the Y-direction, and move it 1 unit downward in the Z-direction.

```
2 TRANSLATE IM
```

will move `IM` by +2 units along each axis.

ROTATE Rotates Objects

If you have read Writing Text and Drawing Flat Pictures, this may trip you up. You are not now rotating a picture in a plane, but an object in space. Think of the picture in the plane as being rotated around a Z-axis perpendicular to the plane. Next, think of there being two other axes also.

```
45 ¯30 90 ROTATE IM
```

will rotate `IM` 45 degrees around the X-axis, -30 degrees around the Y-axis, and 90 degrees around the Z-axis, in that order. In each case a positive rotation is counterclockwise about the axis, as seen from the positive side of the axis looking toward the origin.

```
90 ROTATE IM
```

will rotate `IM` 90 degrees about each axis, in the order X, Y, Z.

Example

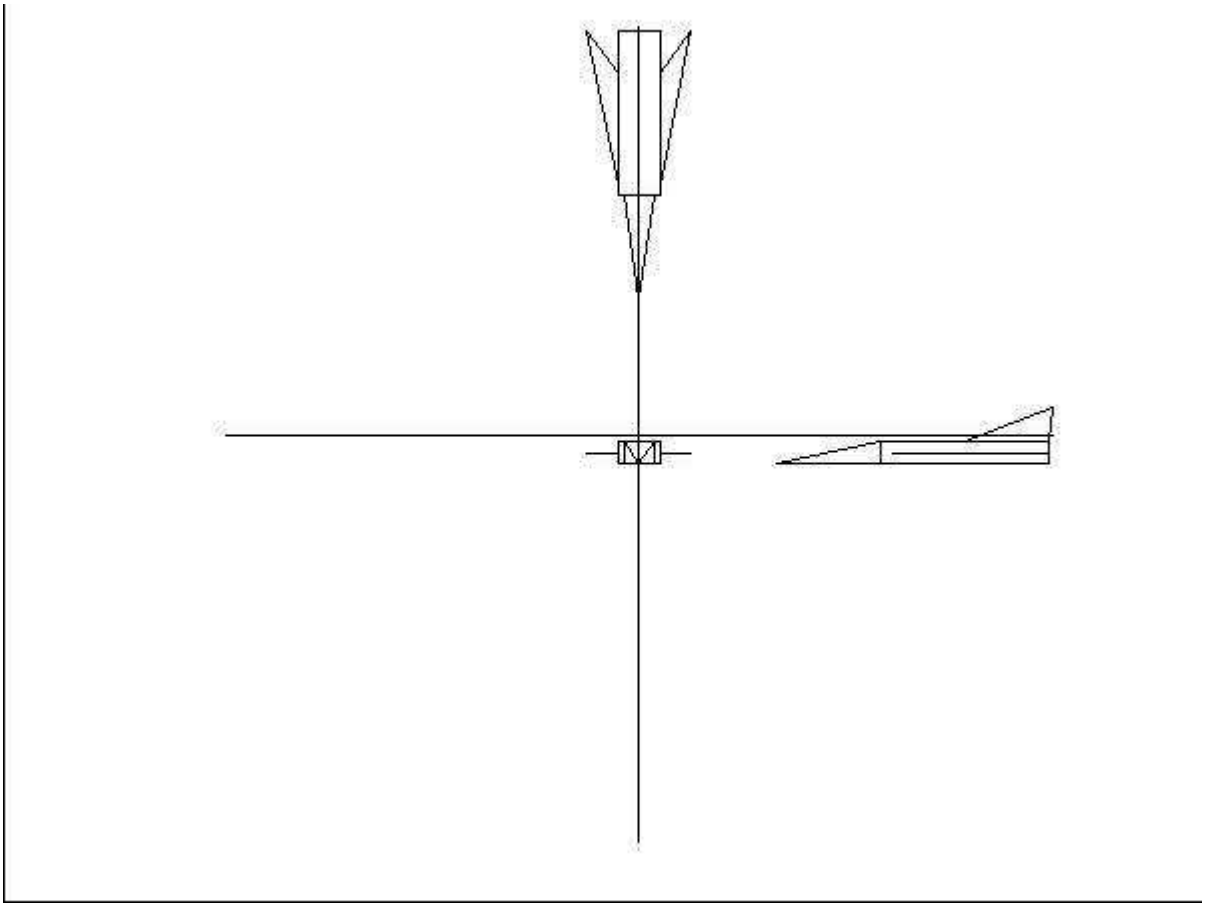
The next picture shows the result of the following sequence of statements:

```
RETICLE
SKETCH IM←¯1 1 SCALE AP
SKETCH 2 0 0 TRANSLATE 0 ¯90 0 ROTATE IM
SKETCH 0 2 0 TRANSLATE 90 0 0 ROTATE IM
```

The first two statements in this example produced the previous picture. Think of that figure as an airplane, its nose pointed toward you along the Z-axis. The Y-axis is on the plane of the screen, extending from bottom to top. The X-axis is also on the screen, extending from left to right.

The third statement in the example rotates the figure 90 degrees clockwise around the Y-axis, so the nose of the plane points to the left. Then it moves the image 2 units away to the right and sketches it. (The result is a projection of the object on the X,Y-plane.)

The last statement rotates the figure 90 degrees around the X-axis, so the nose is pointing down. Then it moves the image 2 units upward and sketches it. (The result is a projection of the object on the X,Z-plane.)



THREEVIEWS Sketches Three Projections

The view of an object shown on the previous page is a common way of looking at a 3-dimensional figure. So common is it that a single function is provided to produce the entire picture in one step.

```
THREEVIEWS AP
```

to do this:

1. Scale `AP` to the window you have assigned.
2. Draw projections of the scaled object on the
   - X,Y-plane
   - Y,Z-plane
   - X,Z-plane

   suitably placed on the display screen.

Drawing Other Projections

SKETCH draws an orthographic projection of a 3-dimensional object. This section tells how to draw other types of projections.

The functions that follow calculate data for projections, but they do not all draw the pictures. The examples show how they would typically be used with SKETCH.

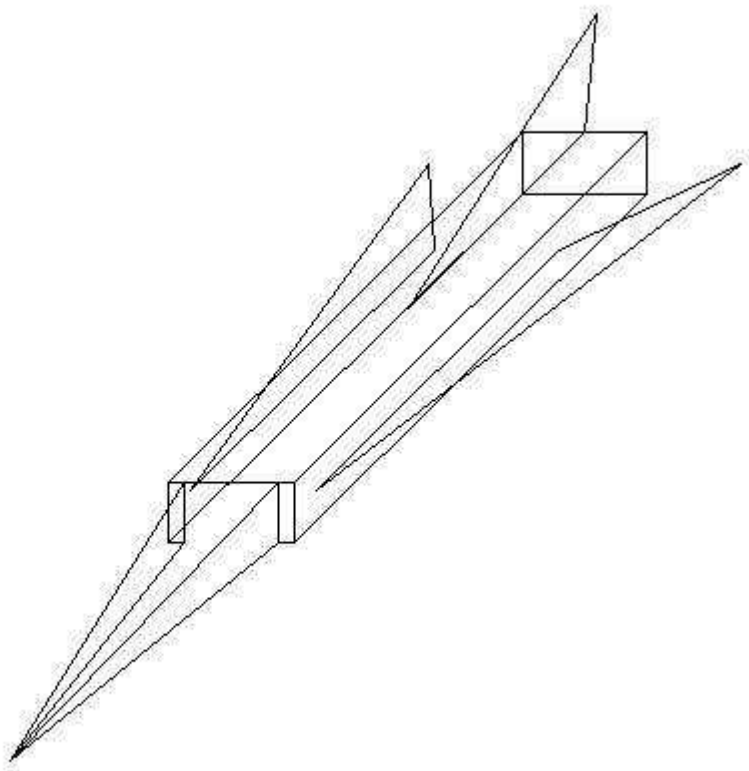
Also, the figures coming up look better with a bigger object than IM. To create the figures, first execute

```
BIGM←3 MAGNIFY IM
```

OBLIQUE Creates an Oblique Projection

The only argument for this function is the 4-column matrix that describes the object. To produce the next picture, execute

```
SKETCH OBLIQUE BIGM
```

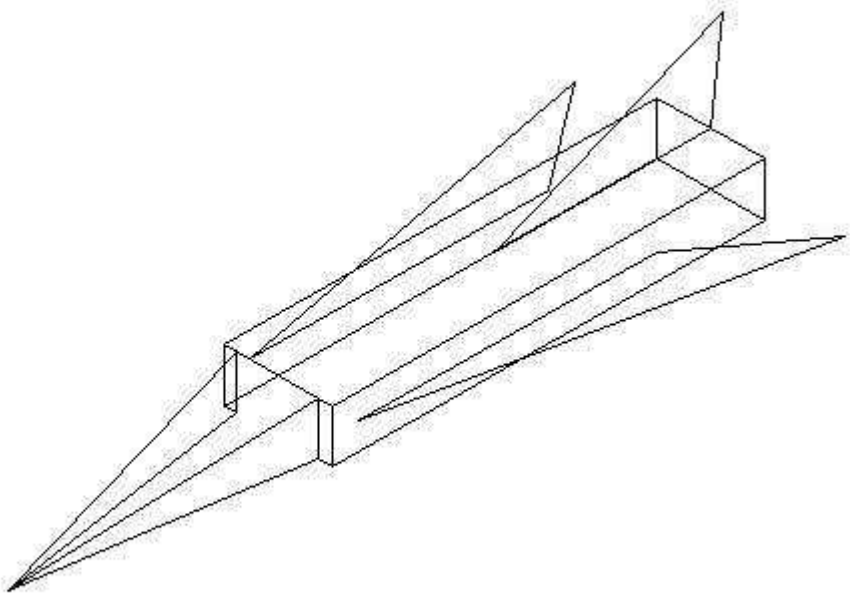


ISOMETRIC Creates an Isometric Projection

In an isometric projection, lines in the X,Y-plane, the Y,Z-plane, and the X,Z-plane are foreshortened by equal amounts.

The only argument for this function is the 4-column matrix that describes the object. To produce an the picture of an isometric projection, execute

```
SKETCH ISOMETRIC BIGM
```



PERSPECTIVE Creates an Perspective Projection

`PERSPECTIVE` creates a perspective projection of an object onto the X,Y-plane, from a vantage point on the Z-axis, perpendicular to the plane of the paper. It needs a left argument to tell how far out along the Z-axis the vantage point is.

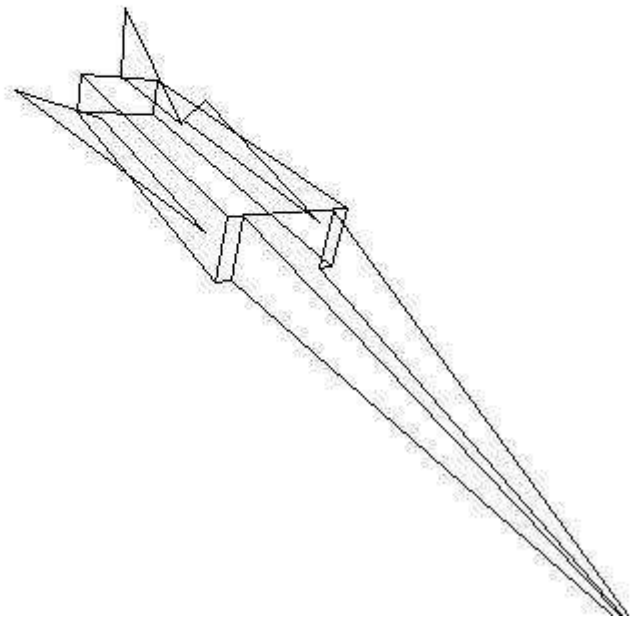
The next few examples make their points better if you are not seeing the object head on. So first turn it a little by executing

```
ROTM←20 30 0 ROTATE BIGM
```

Then, to produce the following picture, execute

```
SKETCH 4 PERSPECTIVE ROTM
```

Here the 4 in the left argument of PERSPECTIVE puts the vantage point at (0, 0, 4).



STEREO Draws Stereo Projections

STEREO draws a pair of stereoscopic views of an object. Each view in a pair is drawn from a slightly different vantage point, as if from a left and right eye. If you can let your eyes go slightly out of focus when viewing the pair, you can perhaps see them as a combined image in which the 3-dimensional effect is evident. (Or you could acquire a stereoscopic viewer.)

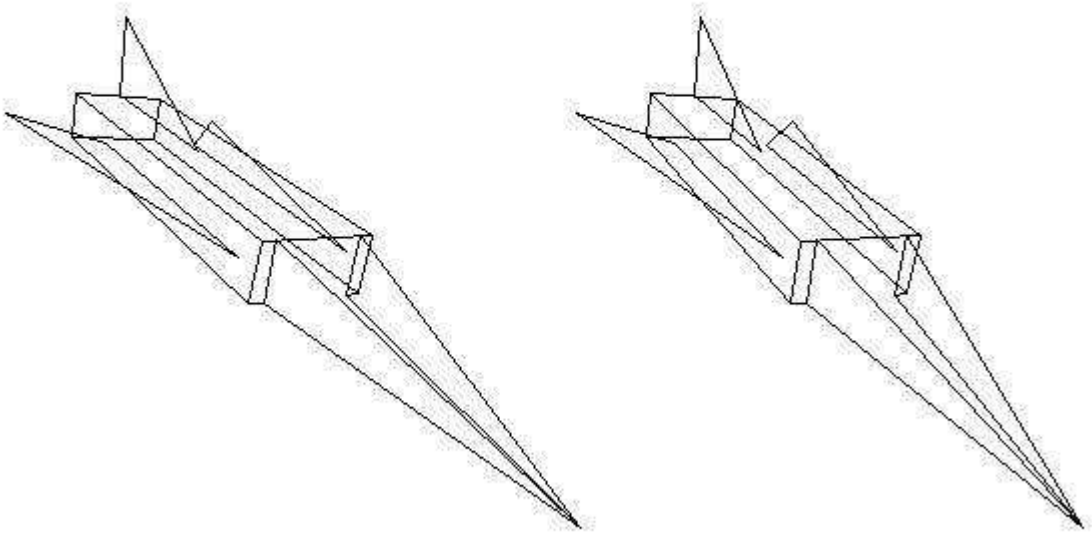
The function requires a vector of 3 elements as a left argument. In order, the elements specify

1. The distance separating two replicas of the object on either side of the Z-axis. (This is the distance between the two eyes that see the object.)
2. The distance of the observer from the origin along the Z-axis. (This corresponds to the left argument for PERSPECTIVE.)
3. The distance between the two views on the display screen.

To create the following picture, execute

```
0.3 6 2 STEREO ROTM
```

This gives a stereoscopic view of the object in the perspective picture, from the same vantage point.



Zooming

Zooming is the effect you get from decreasing the distance between yourself and the object you are observing. There are two ways to produce this effect, and the results are not the same. You can try both ways, and see their differences, by executing the two sequences of statements that follow.

Zoom 1: Moving Yourself Toward the Object

Execute the following statements in succession, and display the result of each.

```
ZM←0.6 ¯1.5 0 TRANSLATE 0 0 ¯20 ROTATE 2 MAGNIFY IM
SKETCH 6 PERSPECTIVE ZM
SKETCH 4.5 PERSPECTIVE ZM
SKETCH 3 PERSPECTIVE ZM
SKETCH 1.5 PERSPECTIVE ZM
```

In this sequence, watch for the distortion that results as the point of perspective moves closer to the X,Y-plane. That point is the vantage point of an imaginary observer. But you will not see what the imaginary observer sees unless you move your eye toward the screen as he moves toward the X,Y-plane.

Zoom 2: Moving the Object Toward You

Execute the following statements in succession, and display the result of each.

```
SKETCH 6 PERSPECTIVE 0 0 1.5 TRANSLATE ZM
SKETCH 6 PERSPECTIVE 0 0 3 TRANSLATE ZM
SKETCH 6 PERSPECTIVE 0 0 4.5 TRANSLATE ZM
SKETCH 6 PERSPECTIVE 0 0 6 TRANSLATE ZM
```

In this sequence, you stay put, as far from the screen as the imaginary observer is from the X,Y-plane. The object comes up and moves past you.

Function Reference

This section contains an alphabetic list of the GRAPHPAK functions that you may need to invoke directly. Each entry includes the function's syntax and descriptions of the function's arguments, results, and effect.

AND	ANNX	ANNY	AVG
AXES	AXIS	BY	CHART
CLEAR	COLOR	CONTOUR	COPY
COPYN	DRAW	ERASE	EXP
FONTDEFINE	FILL	FIT	FIXVP
FREQ	FSSAVE	FSSHOW	GSSAVE
GSLOAD	HCHART	HOR	INTO
ISOMETRIC	LABEL	LBLX	LBLY
LOG	LOGLOG	MAGNIFY	MODE
OBLIQUE	OF	PERSPECTIVE	PIECHART
PIELABEL	PLOT	POLY	POWER
PRINTFILE	READ	RESTORE	RETICLE
ROTATE	SAXES	SAXISX	SAXISY
SAXISZ	SCALE	SCRATCH	SKETCH
SL	SLABEL	SLBLX	SLBLY
SLBLZ	SPLINE	SPLOT	SS
STEP	STEREO	STITLE	STYLE
SURFACE	SXFM	THREEVIEWS	TITLE
TRANSLATE	USE	USING	VER
VIEW	VIEWPORT	VS	WIDTH
WITH	WRITE	XFM	

AND

result ← *left* `AND` *right*

left, *right*
: are numeric vectors of Y-values, or arrays of the type used as right arguments for `PLOT`.

result
: is an array of the type used for `PLOT`.

Effect

The function, together with `VS`, structures data for the right argument of `PLOT` and `SPLOT`.

Example

See Plotting Several Graphs at Once with AND and VS.

ANNX

left `ANNX` *right*

left
: is a scalar giving the vertical distance of an annotation for the X-axis from its default position (1.5 lines below any labels). The value is in problem-space units in the Y-direction, either positive or negative. 0 specifies using the default position.

right
: is a character vector or matrix. If *right* is a vector, it may contain the character ← to delimit new lines. If *right* is a matrix, successive rows contain successive lines of the annotation.

Effect

The function writes the annotation *right* in the position described by *left*. The default position is controlled by the variable `ofx`.

Example

See Creating Axes, Labels, Annotations, and Titles.

ANNY

left `ANNY` *right*

left
: is a scalar giving the horizontal distance of an annotation for the Y-axis from its default position (2 characters to the left of any labels). The value is in problem-space units in the X-direction, either positive or negative. 0 specifies using the default position.

right
: is a character vector or matrix. If *right* is a vector, it may contain the character ← to delimit new lines of the annotated text block. (The annotation effect depends on the shape of the text block and the type of character generation.) If the text block is wider than high:

  - For image characters, each line of the text block appears as a column of letters.
  - For stroked or vector characters, the entire text block is rotated 90 degrees counterclockwise.

  Otherwise, the annotation text block is displayed unchanged. If *right* is a matrix, successive rows contain successive lines of the annotation text block.

Effect

The function writes the annotation *right* in the position described by *left*. The default position is controlled by the variable `ofy`.

Example

See Creating Axes, Labels, Annotations, and Titles.

AVG

result ← `AVG` *right*

right
: is a 2-column matrix of coordinates of points.

result
: is a 2-column matrix of points on the best-fit function.

Effect

The function calculates the best fit to the points in its right argument by a horizontal line. It leaves a character description of this function in the variable `fitmsg` and sets `fitpts` to *right*. (The height of the line above the X-axis is the mean of the Y-values.)

The result matrix is typically used as input to `FIT`.

Example

See Choosing a Function for a Least-Squares Fit.

AXES

```
AXES
```

Effect

Identical to `0 AXIS 0`.

AXIS

X `AXIS` *Y*

X, *Y*
: are independently scalars, vectors, matrices of up to 3 rows, or arrays of rank 3. The left and right arguments of `AXIS` govern, respectively, the X and Y axes to be drawn.

Effect

`AXIS` draws axes for a graph or chart. Each argument controls, for one axis, the following characteristics:

- The location of the axis
- The attribute values of the axis
- The location of tick marks
- The attribute values of the tick marks
- The length of the tick marks

If the argument is

empty
: the corresponding axis is not drawn.

scalar 0
: the axis is drawn at its default position, as close as possible to the value 0 on the other axis. The location of tick marks is determined by default (by the function `TM`), and default attribute values (given by the first element of `av`) are used for both the axis and the tick marks.

a vector
: the elements specify the locations of tick marks (in problem space coordinates). The axis is drawn at its default position with default attribute values.

a matrix of one column
: then the column may have up to 4 elements, which specify respectively

    1. Attribute codes for the axis.
    2. Attribute codes of the tick marks. (They are located by default). 0 specifies no tick marks.
    3. The length of the tick marks, as a fraction of the mean length of the two axes. For example, 0.02 specifies that tick marks are equal to 2% of the mean axis length. 0 specifies using the default length. The default factors are 0.02 for major tick marks and 0.01 for minor tick marks.
    4. The location of the axis, at a coordinate in problem space measured along the other axis. For example, a value of 5000 in *X* would locate the X-axis as near as possible to the value Y=5000.

a matrix of more than one column
: then the matrix may have up to 3 rows, which specify respectively

    1. The locations of tick marks on the axis, as with a vector argument.
    2. The lengths of the corresponding tick marks, as fractions of the length of the other axis.
    3. The location of this axis on the other axis. (Typically all the elements of this row are equal.)

an array of rank 3

then its length in the first dimension is 1. When displayed, the argument looks like a matrix. The extra dimension signals that the first column of the matrix gives attribute codes; the remainder of the matrix is interpreted as a matrix argument.

In the first column, the first element controls the axis, the second controls the tick marks, and further elements are ignored.

Example

See Creating Axes, Labels, Annotations, and Titles.

BY

result ← *left* `BY` *right*

left
: is a vector of Y-values for a contour map.

right
: is a vector of X-values for a contour map.

result
: is a form that can be used as a left argument of `OF`.

Effect

Along with the function `OF`, `BY` structures data into the form required by the right argument of `CONTOUR`.

Example

See Structuring Data for Contour.

CHART

left `CHART` *right*

left

is a character vector whose elements control aspects of the output chart. The characters listed below may appear in the vector in any order, but the order in which digits appear will affect the result.

Character	Meaning
`A`	Do not draw the axes.
`B`	Draw a bar graph (rather than a column chart).
`F`	Fill the bars or columns.
`f`	Fill and edge the bars or columns.
`G`	For multiple sets of data, draw a grouped chart (rather than a stacked chart).
`L`	Label the axes.
`S`	Use the scaling factors implied by the current problem- space window (`w`) and the scaling viewport (`svp`).
`0-9`	Use the corresponding element of the plot attribute vector (`pa`) as an attribute code for the output graph. (0 corresponds to element 10 of `pa`.) Digits in the character vector *left* are taken from left to right to point to codes for the graphs produced.

right

is a numeric vector, or matrix. If *right* is a

vector

its values are taken as Y-values and its indexes as X-values.

matrix

the values in each column are a separate set of Y-values. The indexes of the elements in the column are the X-values.

Effect

The function plots a bar or column chart, using the values of the right argument.

Example

See CHART Draws Bar and Column Charts.

CLEAR

CLEAR

Effect

CLEAR executes ERASE, and controls the drawing of axes by FIT. The first use of FIT after CLEAR will draw a set of axes, plot points, and draw the fitted curve. Later uses will only draw the new fitted curve.

COLOR

result ← `COLOR` *right*

right
: is a scalar or numeric vector of values for the first attribute (*color*) of a set of lines, patterns, or characters on the display screen. Negative values are interpreted by `COLOR` as attribute codes. Positive values are interpreted as new color values, and must precede any negative values.

result
: is a scalar or numeric vector of attribute codes.

Effect

`COLOR` inserts the new color values (positive elements of *right*) into other attribute codes (negative elements of *right*). It uses the positive elements repeatedly to fill all the negative elements, or it creates new negative elements as needed to take all the positive elements.

`COLOR` is typically used with `USE` to change the attribute vector `av` and with `USING` to change an attribute code temporarily. It may be used together with `STYLE`, `WIDTH`, and `MODE`.

Example

See Specifying Color, Style, Width, and Mode.

CONTOUR

left `CONTOUR` *right*

left
: is a vector of Z-values for which contour lines are to be mapped.

right
: is a matrix `M`, with *m+1* rows and *n+1* columns, with the following characteristics:
`M[1;1]`
controls whether tags are written on the contour curve. Tags are written if this value is 1.
`M[1;2]` through `M[1;`*n+1*`]`
is a vector of *n* X-values.
`M[2;1]` through `M[`*m+1*`;1]`
is a vector of *m* Y-values.
`M[I;J]`, for `I` and `J` greater than 1
is the Z-value corresponding to the grid point (Y[`I`],X[`J`]).

Effect

`CONTOUR` draws the contour lines, for the heights given in the left argument, of the function represented by the Z-values in the right argument.

Example

See CONTOUR Draws Contour Maps.

COPY

COPY

Effect

Produces a printed copy of the graphic output.

The variable copyname specifies the output destination of COPY.

On OS/2 and Windows:

copyname can contain either a file specification of the form:

```
[path]filename
```

or a null character vector. If a file specification is supplied, a metafile is written. If a null character vector is supplied, the default printer is used.

On CMS and TSO:

COPY uses the global variable copyctl to control the number of copies and several other aspects of the result. The output is directed to the alternate GDDM device.

On Unix Systems:

COPY is not supported on these operating systems.

COPYN

`COPYN` *destination*

Produces a printed copy of the display screen.

Effect

Assigns *destination* to `copyname` and executes `COPY`.

DRAW

`DRAW` *right*

right
: is a 2- or 3-column matrix. The last two columns in each row are the X- and Y-coordinates of a point in virtual space.

  If *right* has 3 columns, the first element in each row is an attribute code; 0 represents an invisible movement.

  Each row specifies a line to be drawn from the point in the previous row to the new point. The attribute code for the line are given by the first element if the row has three elements, or by the first element of the attribute vector `av` if the row has two elements.

Effect

The function draws the figure represented by *right* in the graphic field. Lines that extend beyond the clipping viewport may be cut off, depending on the variable `sci`.

Example

See DRAW Draws Pictures.

ERASE

ERASE

Effect

Erases the content of a display screen.

Example

See ERASE.

EXP

result ← `EXP` *right*

right
: is a 2-column matrix of coordinates of points.

result
: is a 2-column matrix of points on the best-fit function.

Effect

The function calculates the best fit to the points in its right argument by an exponential function of the form `C1×⋆C2×X`. It leaves a character description of this function in the variable `fitmsg` and sets `fitpts` to the value in *right*. The result matrix is typically used as input to `FIT`.

Example

See Choosing a Function for a Least-Squares Fit.

FONTDEFINE

Note: This function is only supported on workstation systems. Outline fonts (PostScript and TrueType) are only supported on OS/2 and Windows.

font `FONTDEFINE` *name*

font
: Vector font file name ('SCRIPT.AVF') or outline font face name ('Times New Roman') or `¯1` to disassociate name from font (default)

name
: Name for referring to font (maximum 12 characters)

Effect

This function associates a name with a vector font and makes it available to GRAPHPAK. Multiple vector fonts can be defined.

The global variable `VFONT` specifies which font is used when drawing text. `VFONT` is set to the name of a font as defined using `FONTDEFINE`.

The font currently named in `VFONT` is used if `dd[9] = 2`.

Example

The following example writes the word Hello using image, stroked, vector, and outline fonts.

```
)LOAD 2 GRAPHPAK
OPENGP
dd[9]←0
10 10 8 0 0 0 WRITE 'Hello'
dd[9]←1
10 20 8 0 0 0 WRITE 'Hello'
dd[9]←2
'SCRIPT.AVF' FONTDEFINE 'AVF'
'Helvetica'  FONTDEFINE 'Outline'
VFONT←'AVF'
10 30 8 0 0 0 WRITE 'Hello'
VFONT←'Outline'
10 40 8 0 0 0 WRITE 'Hello'
VIEW
¯1 FONTDEFINE 'AVF'
¯1 FONTDEFINE 'Outline'
CLOSEGP
```

FILL

`FILL` *right*

right
is a 2- or 3-column vector or matrix, like the right argument of `DRAW`.

Effect

The function draws in the graphic field a solid representation of the figure defined by *right*. Areas that extend beyond the clipping viewport may be cut off, depending on the variable `sci`.

Fill patterns are given by the first column of *right* if it has three columns, or by the first element of the attribute vector `av` if *right* has two columns. The style value of the attribute code selects the fill pattern. Whether previous patterns are allowed to show through a superimposed pattern is controlled by the variable `sf`.

Example

See FILL Fills Polygons.

FIT

`FIT` *right*

right
is a 2-column matrix of points on a curve.

Effect

The function does the following:

- Draws the points of the curve. (This is typically the result of fitting a curve to a set of points by one of the functions `AVG`, `SL`, `EXP`, `LOG`, `LOGLOG`, `POLY`, `POWER`, or `SPLINE`.)
- Draws the axes and displays the data points to be fitted (but only on the first use of the function after `CLEAR`)
- Displays the description of the curve in `fitmsg`
- Creates a function `FITFUN` that can be used to calculate more points on the curve described in `fitmsg`.

Example

See FIT Plots a Fitted Function.

FIXVP

`FIXVP` *right*

right
: is a 2-column numerical matrix. The first column is a set of X-coordinates (in virtual-space units); the second is a set of Y-coordinates. Together the two columns are a set of coordinates of the vertices of the clipping viewport. The polygon described by *right* must be convex, and the vertices must be given successively in counterclockwise order around the boundary.

Effect

The function constructs a new value for the variable `cvp`, so that the clipping viewport is specified as the polygon defined by *right*.

FREQ

result ← *left* `FREQ` *right*

left

is a numeric vector `L` that controls the class boundaries and the area under the graph, as follows:

Element	Specifies
`L[1]`	whether the resulting frequency distribution is to be normalized (value = 1), or not (value = 0).
`L[2]`	either the class interval size, if there are no further elements in `L`, or the first class boundary, if the other elements follow. If `L` is of length 2, then boundaries are chosen at integer multiples of `L[2]` so as to include all the elements in *right*.
`L[3...]`	additional class boundaries. If there are three or more elements in `L`, the class intervals can differ. Only the elements of *right* between the first and last boundary will be included in *result*.

right

is a vector of measurements.

result

is a 2-column matrix. The first column is a set of class boundaries, and the second is the set of frequencies of measurements for the classes that begin at those boundaries. The first and last rows of the matrix represent classes with width and frequency of zero.

Effect

The matrix in *result* is of the form required as a right argument to the function `STEP`.

Example

See Plotting Frequency Functions.

FSSAVE

Note: This function is only supported on CMS and TSO.

`FSSAVE` *filename*

filename
is the name of a file to be created to hold graphic output.

Effect

The function saves the current content of the graphic field in the file *filename*.

Example

See FSSAVE and FSSHOW.

Note: On TSO, use of `FSSAVE` requires that a partitioned data set with a block size of 400 be allocated to DDNAME ADMSAVE prior to using `FSSAVE`.

FSSHOW

Note: This function is only supported on CMS and TSO.

`FSSHOW` *filename*

filename
is the name of a file that was created by `FSSAVE`.

Effect

The function displays in the graphic field the contents of the file *filename*.

Example

See FSSAVE and FSSHOW.

Note: On TSO, use of `FSSHOW` requires that a partitioned data set with a block size of 400 be allocated to DDNAME ADMSAVE prior to using `FSSHOW`.

GSSAVE

Note: This function is only supported on CMS and TSO.

descrip `GSSAVE` *filename*

descrip
is an optional description of the graphic output to be included in the file.
filename
is the name of an ADMGDF file to be created to hold graphic output.

Effect

The function saves the current content of the graphic field in the ADMGDF file.

Example

See GSSAVE, GSLOAD, and PRINTFILE.

Note: On TSO, use of `GSSAVE` requires that a partitioned data set with a block size of 400 be allocated to DDNAME ADMGDF prior to using `GSSAVE`.

GSLOAD

Note: This function is only supported on CMS and TSO.

`GSLOAD` *filename*

filename
is the name of an ADMGDF file created by `GSSAVE`.

Effect

The function replaces the content of the graphic field with the contents of the ADMGDF file.

Example

See GSSAVE, GSLOAD, and PRINTFILE.

Note: On TSO, use of `GSLOAD` requires that a partitioned data set with a block size of 400 be allocated to DDNAME ADMGDF prior to using `GSLOAD`.

HCHART

left `HCHART` *right*

left

is a character vector whose elements control aspects of the output chart. The characters listed below may appear in the vector in any order, but the order in which digits appear will affect the result.

Character	Meaning
`F`	Fill the boxes in the output chart.
`0-9`	Use the corresponding element of the plot attribute vector (`pa`) to specify fill patterns for the boxes in the output chart. (0 corresponds to element 10 of `pa`.) Digits in the character vector *left* are taken from left to right to point to the attribute codes for the boxes.
-	Draw connecting lines with `STYLE 2`. (The default is `STYLE 1`.)

right

contains the captions to be placed in the boxes of the output chart. It may have any of these formats:

1. A vector, containing the characters `⌶` to delimit captions and ← to delimit lines within a caption. Blanks following the delimiter `⌶` show that a caption is dependent on some previous one.
2. A matrix, in which each row contains a separate caption. The rows may contain the delimiter ← to delimit lines within a caption. Blanks at the beginning of a row show that a caption is dependent on some previous one.
3. An array of rank 3, in which each row of each matrix is a separate line of a separate caption. Blanks at the beginning of the first row of a matrix indicate that a caption is dependent on some previous one.

Effect

The function draws a hierarchic chart, like that shown in the cheese chart in the Writing Text and Drawing Flat Pictures. The size of the chart is dependent on the current setting of the character size in `dd`.

Example

See HCHART Draws Hierarchic Charts.

HOR

`HOR` *right*

Effect

Identical to `0 ANNX` *right* followed by `0 AXIS ''`.

INTO

result ← *left* `INTO` *right*

left
: is a numeric vector, typically of 4 elements and representing a rectangular region in the same way as do the window in problem space (`w`) or the scaling viewport (`svp`).

right
: is a numeric vector of the same length and format as *left*.

result
: is an array of shape 2 3 2 that defines a transformation of the region represented by *left* into the region represented by *right*.

Effect

The function produces a data structure that is suitable for the left argument of `XFM`.

Example

See Magnifying, Translating, and Rotating Pictures.

ISOMETRIC

result ← `ISOMETRIC` *right*

right
: is a 4-column vector or matrix. The last three numbers in each row are the X-, Y-, and Z-coordinates of a point. The first number gives the attribute code for a line drawn to that point.

result
: is a 3- or 4-column vector or matrix, structured as input to `SKETCH`, that represents an isometric projection of *right* onto the X,Y-plane.

Example

See ISOMETRIC Creates an Isometric Projection.

LABEL

LABEL

Effect

LABEL writes default numeric labels for a pair of X- and Y-axes determined by the window in problem space (w) and the scaling viewport (svp).

LBLX

X `LBLX` *right*

X
: is a vector, matrix, or array of three dimensions that describes an axis and its tick marks in the same way as for the function `AXIS`.

right
: is a scalar 0, numeric vector, or character vector or matrix.

Effect

The function writes labels for tick marks on an X-axis described by the left argument *X*. The labels appear below the axis unless the axis is at the top of the scaling viewport, when they appear above it.

If *right* is

0
: the function labels tick marks with the problem-space coordinates of their locations. (The locations are given by the left argument *X*.)

a numeric vector
: the function reformats it as a character vector and uses it as labels.

a character vector or matrix
: the function uses it as labels. *right* may contain APL2 characters for the following special purposes:

  ¨ Begin a new line.

  ⌶ Begin a new label.

  The characters used for these purposes are contained in the variable `dlc`.

Example

See Creating Axes, Labels, Annotations, and Titles.

LBLY

Y `LBLY` *right*

Effect

Similar to the effect of `LBLX`, except that the function writes labels for the Y-axis described by its left argument. The labels appear to the left of the axis unless the axis is at the right of the scaling viewport, when they appear to its right.

Example

See Creating Axes, Labels, Annotations, and Titles.

LOG

result ← `LOG` *right*

right
: is a 2-column matrix of coordinates of points.

result
: is a 2-column matrix of points on the best-fit function.

Effect

The function calculates the best fit to the points in its right argument by an exponential function of the form `C1×⋆C2×X`. It leaves a character description of this function in the variable `fitmsg` and sets `fitpts` to the value in *right*.

The result matrix is typically used as input to `FIT`. The function signals `FIT` to plot the curve as a straight line on semi-log axes.

Example

See Choosing a Function for a Least-Squares Fit.

LOGLOG

result ← `LOGLOG` *right*

right
: is a 2-column matrix of coordinates of points.

result
: is a 2-column matrix of points on the best-fit function.

Effect

The function calculates the best fit to the points in its right argument by a power function of the form `C1×X⋆C2`. It leaves a character description of this function in the variable `fitmsg` and sets `fitpts` to the value in *right*.

The result matrix is typically used as input to `FIT`. The function signals `FIT` to plot the curve as a straight line on log-log axes.

Example

See Choosing a Function for a Least-Squares Fit.

Magnify

result ← *left* `MAGNIFY` *right*

left
: is a scalar, or a vector of 2 or 3 elements.

right
: is a vector or matrix of 2, 3, or 4 columns.

  If *right* has two columns, the numbers in each row are the X- and Y-coordinates of a point.

  If *right* has three or four columns, the last two or three numbers in the row are the X- and Y-coordinates, or the X-, Y-, and Z-coordinates, of a point. The first number gives the attribute code for a line drawn to that point.

result
: is a 3- or 4-column vector or matrix, structured as input to `SKETCH`, that represents the same figure as *right*, expanded by the elements of *left* along each coordinate. If the left argument is a scalar, it is extended to match the number of coordinates in the right argument. Negative elements of *left* produce mirror images for the corresponding coordinates.

Example

See Magnifying, Translating, and Rotating Pictures and MAGNIFY Magnifies and Shrinks Objects.

MODE

result ← `MODE` *right*

right
: is a scalar or numeric vector of values for the fourth attribute (*mode*) of a set of lines on the display screen. Negative values are interpreted by `MODE` as attribute codes. Positive values are interpreted as new mode values, and must precede any negative values.

result
: is a scalar or numeric vector of attribute codes.

Effect

`MODE` inserts the new mode values (positive elements of *right*) into the other attribute codes (negative elements of *right*). It uses the positive elements repeatedly to fill all the negative elements, or it creates new negative elements as needed to take all the positive elements.

`MODE` is typically used with `USE` to change the attribute vector `av` and with `USING` to change an attribute code temporarily. It may be used together with `COLOR`, `STYLE`, and `WIDTH`.

Example

See Specifying Color, Style, Width, and Mode.

OBLIQUE

result ← `OBLIQUE` *right*

right
: is a 4-column vector or matrix. The last three numbers in each row are the X-, Y-, and Z-coordinates of a point. The first number gives the attribute code for a line drawn to that point.

result
: is a 4-column vector or matrix, structured as input to `SKETCH`, that represents an oblique projection of *right* on the X,Y-plane.

Example

See OBLIQUE Creates an Oblique Projection.

OF

result ← *left* `OF` *right*

left
: a matrix of the form produced by the function `BY`.

right
: a matrix of Z-values.

result
: a matrix of the form required for a right argument of `CONTOUR`.

Example

See Structuring Data for Contour.

PERSPECTIVE

result ← *left* `PERSPECTIVE` *right*

left
: is the Z-coordinate of the vantage point for the perspective.

right
: is a 4-column vector or matrix. The last three numbers in each row are the X-, Y-, and Z-coordinates of a point. The first number gives the attribute code for a line drawn to that point.

result
: is a 4-column vector or matrix, structured as input to `SKETCH`, that represents a perspective view of *right*, projected onto the X,Y-plane from the point (0, 0, *left*).

Example

See PERSPECTIVE Creates an Perspective Projection.

PIECHART

left `PIECHART` *right*

left

is a character vector whose elements control aspects of the output chart. The characters listed below may appear in the vector in any order, but the order in which digits appear will affect the result.

Character Meaning

`B` Fill the boxes around labels. If the labels are outside the slices, the attribute values used are the same as those for the slices. If the labels are inside, the attribute code is given by `pie[6]` if that is nonzero; otherwise the label appears in a hole in the filling for the slice.

This parameter is superseded if the second column of *right* is nonzero.

`C` Bound the segments of the chart with chords instead of arcs.

`E` Draw boxes around the labels, using the code in the first element of the attribute vector.

`F` Fill the segments of the chart, using attribute codes in the plot attribute vector.

`f` Fill and edge the segments. For the edges, use attribute codes in the second row of the plot attribute vector if that exists, or, otherwise, the code in the first element of the attribute vector.

`L,N` Label the segments with the numeric values in *right*.

`l` Label only the pie.

`P` Label the slices with percentage values, rounded to one decimal place.

`R` Draw a regular polygon instead of a pie chart. The number of sides is given by the first element in *right*.

`0-9` Use the corresponding element of the plot attribute vector (`pa`) as an attribute code for a segment. (0 corresponds to element 10 of `pa`.) Digits in the character vector *left* are taken from left to right to point to codes for the segments produced.

`T` Label the pie segment with text from variable `t`.

right

is a numeric vector or matrix, or an array of rank 5. If *right* is a

vector

the total size of the pie is proportional to the sum of the elements, and each slice is proportional to one element.

matrix

it can have up to five columns. Elements in the columns are interpreted as follows:

Column 1

Elements determine the sizes of slices as if the column were a vector argument.

Column 2

Elements are attribute codes for the fill pattern for the corresponding label. (The values used override anything specified by the control character `B`.)

Column 3

Elements determine where labels for corresponding slices are placed, as follows

p>0 Put the label outside the pie at a distance of p times the radius of the pie.

0 Center the label in the slice if it will fit; otherwise put it outside the pie in a default position.

p Put the label outside the pie in a default position.

Column 4
A number, p, in this column offsets the corresponding slice by p times the radius of the pie.
Column 5
A 1 in this column omits the corresponding slice; 0 keeps it.
array of rank 5
it can contain both numeric data and labels. The function `WITH` produces data structures of this type.

Effect

The function draws a pie chart.

Example

See PIECHART Draws Pie Charts.

PIELABEL

left `PIELABEL` *right*

left
: is a numeric data array like that used for a right argument of `PIECHART`.

right
: is 0, or a character array of labels of one of the following types. If *right* is a
vector
it may contain `⌶` to delimit labels and ¨ to delimit lines within a label.
matrix
each row is a label. A row may contain ¨ to delimit lines within a label.
rank 3 array
Each matrix corresponds to a label, and each row of the matrix to a line of the label. The rows of the array may not contain delimiters.

If *right* is 0, numeric labels are generated from the values in *left*.

Effect

The function writes labels for piecharts.

Example

See PIECHART Draws Pie Charts.

PLOT

PLOT *right*

right
is a numeric vector, matrix, or an array of special type. If *right* is
a vector
its values are taken as Y-values and its indexes as X-values.
a matrix
the values in its first column are taken as X-values and the values in remaining columns as sets of corresponding Y-values.
an array of special type
it defines a set of several graphs to be plotted. An array of this type has the following characteristics:

- Its rank is greater than one.
- Its length is 1 in every dimension but the last. (When displayed, it looks like a vector.)
- Its elements are used as follows:

  For example, the array shown in the figure would produce two graphs: one with the points (0,7), (1,8), and (2,9); and one with the points (1,30), (2,20). Typically, the array is generated by the functions AND and VS.

```
                          2 3 2 ... 0 7  1 8  2 9  1 30  2 10  ...
                          - - - --- -------------  ----------  ---
                          | | |  |         |              |       |
Number of graphs ----------+ |  |         |              |       |
Number of points in graph 1 -+  |         |              |       |
Number of points in graph 2 ---+ |         |              |       |
  (continue number of points, ----+        |              |       |
  if there are more graphs)                |              |       |
Points of graph 1 -------------------------+              |       |
Points of graph 2 ----------------------------------------+       |
  (continue with points of other ---------------------------------+
  graphs, if there are more)
The rank of the plot array is greater than one
```

Effect

The function plots one or more (superimposed) line graphs specified by the right argument *right*.

In drawing a graph, PLOT

1. Assigns a window (w) in the problem space that will fill the scaling viewport (svp) while yielding convenient locations for tick marks on the axes.
2. Draws the axes with default attribute values.
3. Uses the grid control variable (tm) to determine whether or not to extend tick marks.
4. Labels the tick marks with default attribute values.
5. Looks in the plot attribute vector (pa) for attribute codes for successive graphs. The first element of pa is the code for the first graph, the second element of pa is the code for the second graph, and so on.

Example

See A Model Function: PLOT Draws a Line Graph.

POLY

result ← *left* `POLY` *right*

left
: is an integer, the order of the polynomial to be fitted.

right
: is a 2-column matrix of coordinates of points.

result
: is a 2-column matrix of points on the best-fit function.

Effect

The function calculates the best fit to the points in its right argument by a polynomial. The order of the polynomial is given by its left argument. It leaves a character description of this function in the variable `fitmsg` and sets `fitpts` to *right*. The *result* matrix is typically used as input to `FIT`.

Example

See Choosing a Function for a Least-Squares Fit.

POWER

result ← `POWER` *right*

right
: is a 2-column matrix of coordinates of points.

result
: is a 2-column matrix of points on the best-fit function.

Effect

The function calculates the best fit to the points in its right argument by a power function, of the form `Y←C1+X⋆C2`. It leaves a character description of this function in the variable `fitmsg` and sets `fitpts` to *right*. The *result* matrix is typically used as input to `FIT`.

Example

See Choosing a Function for a Least-Squares Fit.

PRINTFILE

Note: This function is only supported on CMS and TSO.

ctl `PRINTFILE` *filename*

Build a GDDM Print File from the Current Graphic Field.

The function `PRINTFILE` saves the current graphics screen image into a GDDM print file. The image can then be printed as a stand-alone document or by imbedding it in a Bookmaster document.

ctl An optional vector containing 1, 2, or 3 integers. If supplied, the integers have the following meanings:

- `[1]` Type of file to produce. The default value, `0`, causes `PRINTFILE` to produce a LIST38PP file which can be printed as a stand-alone document. A value of `1` causes `PRINTFILE` to produce a PSEG38PP file which can be imbedded in a Bookmaster figure. A value of `1` causes `PRINTFILE` to produce a OVLY38PP file which can be used as an overlay.
- `[2]` Width in millimeters of the image to produce. The default value is 165.
- `[3]` Resolution of the print file. The value controls whether the intermediate GDF file used during the conversion of the image to a PSEG uses 1 or 4 bytes. The argument should be either 1 or 4. The default is 1.

filename A character vector of 1 to 8 characters. It is the name of the file to be built. On CMS, this is the file name of the file; the filetype will be LIST38PP, PSEG38PP, or OVLY38PP. On TSO, it is the second qualifier of a dataset name. The first qualifier will be the current PROFILE PREFIX setting. The third qualifier will be LIST38PP, PSEG38PP, or OVLY38PP. Consult the GDDM manuals for the record format, lrecl, and blksize restrictions concerning GDDM print files.

This function does not produce an explicit result.

When a document type of `1` is specified, the `PRINTFILE` function produces a PSEG38PP file. The text below demonstrates how such a file might be imbedded in a Bookmaster document.

Sample Bookmaster Code

```
:fig place=inline width=page frame=box.
:p.
:artwork name=PSEGNAME.
:p.
:efig.
```

When the `PRINTFILE` function produces a PSEG38PP file, the file is intended to be directed to a 3800 printer. It can be directed to a 3812 style printer, but the Bookmaster document must still be Scripted with a device type of 38PPN.

Because GRAPHPAK does not include image symbol sets for 3800 device types, it is suggested that image symbol sets not be used for generating text in images from which print files will be generated. GRAPHPAK can be directed to use vector symbol sets by setting `dd[9]←2`.

The comments in `PRINTFILE` describe how to modify the function to cause it to produce files useable on other types of devices or direct its output to a ddname on TSO.

READ

result ← *left* `READ` *right*

left

determines the number of entries that will be read. If *left* is
a scalar *n*
then reading continues until *n* entries have been read, or a null line is entered.
empty
then reading continues until a null line is entered.

right

is a numeric vector `R`. The first element of `R` determines which option of the `READ` function is to be used, and the meaning of the other elements of `R` depends on that option. In each case, however, `R[2]` controls the sounding of an alarm when an entry is made: if the value is 0, the alarm will sound.

If `R[1]` is

0 elements beyond `R[2]` are not used.

1 then `R[3]` determines what will be displayed as a result of an entry, as follows:

- 0 no display.
- 1 display a point.
- 2 display a line segment.

`R[4 5]` gives the starting position of the cursor on the screen (in virtual-space coordinates). The default position is at the center of the screen.

3 The character input is accepted from the APL screen. `R[3 4]` gives the position on the graphics field where the text is written, and it gives the position of the lower left corner of the first line of text to be entered.

result

If the option in the first element of *right* is 0 or 1, the *result* is a numeric vector or matrix of three columns, with one row for each entry read. The first column of a row contains the number of the function key by which the entry was made, or 0 if the entry was made by pressing Enter. Columns 2 and 3 contain the coordinates of the location of the cursor when the entry was made.

If the option is 3, *result* is a matrix containing, in successive rows, the lines of text that were entered after executing `READ`.

Effect

The effect of `READ` is controlled by the option in the first element of *right*. If this option is

0 then `READ` returns the coordinates (in virtual space) of a single position of the cursor, the last one recorded. To record a position with this option,

1. Execute `VIEW`.
2. Move the cursor to the desired position and press Enter, or a function key.

3. Execute `0 READ 0`.

1 `READ` returns the coordinates of more than one cursor position. Executing `READ 1` positions the cursor on the display screen; recording begins after that.

3 `READ` accepts input from the APL2 screen. The text you Enter is written to the graphics field and returned as a result.

Example

For options 0 and 1, see READ Options that Read Cursor Positions.
For option 3, see Entering Several Lines of Text.

RESTORE

```
RESTORE
```

Effect

To restore default conditions, the function sets

1. The attribute modification control (`amc`) to 0.
2. The attribute vector (`av`) to a systematic variation through all possible attribute values, by executing `USE ''`.
3. The bar width (`bw`) to 0.8.
4. The clipping viewport (`cvp`) to the screen limits.
5. The character cell size (`dd[7 8]`) to a normal size for the output device. The spacing factors (`dd[5 6]`) are also set.
6. The writing mode (`dd[9]`) to 0 (use image characters).
7. The plot attribute vector (`pa`) to a set of positive integers that avoids pointing to the neutral color in the default setting of `av`, and to the neutral color if the total number of colors is greater than four.
8. The variables `ofx` and `ofy` to the default positions for annotations.
9. The plot character vector (`pc`) to `'○□⋆∘×∇∆'`.
10. The variable `pie` so that the default center of a piechart is in the center of the screen, the default radius is 35% of the shortest dimension of the screen, the first slice starts at 3 o'clock, and the angular span is 360 degrees.
11. The skyscraper bar width (`sbw`) to 0.3.
12. The scaling viewport (`svp`) so that its lower left corner is 8 character spaces horizontally and 3 character spaces vertically from the lower left corner of the screen, and its upper right corner is 3 spaces horizontally and 1 space vertically from the upper right corner of the screen.
13. The grid control (`tm`) to 0 0 (do not extend tick marks).
14. The slopes of axes for skyscraper and surface charts to 0.25.

RETICLE

```
RETICLE
```

Effect

The function draws an image of the clipping viewport and a pair of axes that show the extent of the window in problem space.

Example

See Changing Coordinates.

ROTATE

result ← *left* `ROTATE` *right*

left
: is a scalar or 3-element vector of angles of rotation in degrees.

right
: is a vector or matrix of 2, 3, or 4 columns.

  If *right* has two columns, the numbers in each row are the X- and Y-coordinates of a point.

  If *right* has three or four columns, the last two or three numbers in the row are the X- and Y-coordinates, or the X-, Y-, and Z-coordinates, of a point. The first number is an attribute code for a line drawn to that point.

result
: is a 3- or 4-column vector or matrix, structured as input to `SKETCH`, that represents the same figure as *right*, rotated around the X-, Y-, and Z-axes by the elements of *left*. The rotations are done in the order X, Y, Z.

  If *left* is a scalar, it is expanded to a vector of three elements. But if *right* has only three columns (that is, it represents a figure in the X,Y-plane) the rotation is done entirely in the X,Y-plane around the Z-axis.

Example

See Magnifying, Translating, and Rotating Pictures and ROTATE Rotates Objects.

SAXES

SAXES

Effect

This function is equivalent to the sequence

```
SAXISX 0
SAXISY 0
SAXISZ 0
```

It draws default axes for surface or skyscraper charts.

SAXISX

`SAXISX` *right*

right
: is a scalar 0 or a vector. If it is a vector, its elements give the locations of tick marks on the axis. A scalar 0 specifies using default locations for the tick marks.

Effect

The function draws an X-axis on a surface chart or skyscraper chart.

Example

See Drawing Axes, Labels, and Titles for Surface Charts.

SAXISY

This is nearly identical to `SAXISX`, but draws a Y-axis.

SAXISZ

This is nearly identical to `SAXISX`, but draws a Z-axis.

SCALE

result ← *left* `SCALE` *right*

left
: is a vector giving the maximum and minimum coordinates of the region to which the figure is to be scaled. A 6-element vector gives limits along the X-, Y-, and Z-axes. A 4-element vector gives limits along the X- and Y-axes, for a two-dimensional figure. A 2-element vector is extended to 4 or 6 elements as needed.

right
: is a 3- or 4-column vector or matrix. The first number in each row is an index into the attribute vector. The remaining numbers in the row are the X- and Y-coordinates, or the X-, Y-, and Z-coordinates, of a point.

result
: is a 3- or 4-column vector or matrix, structured as input to `SKETCH`, that represents the figure of *right* scaled to the limits in *left*.

Example

See Define a Window.

SCRATCH

result ← `SCRATCH` *right*

right
is a 2-column matrix representing a set of points to be removed, or is empty.
result
is a 2-column matrix, consisting of the points in `FITPTS` with the designated set of points removed.

Effect

The result of `SCRATCH` is typically used as input to `FIT`. When `FIT` is to be used, the variable `fitpts` contains the set of points to which a fit is desired. (`fitpts` is set by `AVG, EXP, LOG, LOGLOG, POLY, POWER,` and `SL`.) `SCRATCH` does not change the content of `fitpts`, but creates a new (reduced) set of points.

If the argument of `SCRATCH` is empty, the points to be removed can be designated by moving the cursor on the graphic field. For a description of this procedure, see the example.

Example

See Fitting Curves.

SKETCH

`SKETCH` *right*

right
: is a 3- or 4-column vector or matrix. The last two or three numbers in the row are the X- and Y-coordinates, or the X-, Y-, and Z-coordinates, of a point in virtual space. The first number is an attribute code for a line drawn to that point; 0 represents an invisible movement.

  Each row can be considered as a specifying a line to be drawn from the point in the previous row to the new point, using the attribute code given in the first element.

Effect

The function extracts an orthographic projection of the figure defined by *right* onto the X,Y-plane, maps it from the window in problem space into the scaling viewport, and draws it in the graphic field.

Example

See Changing Coordinates and SKETCH Sketches Projections on the X,Y-Plane.

SL

result ← `SL` *right*

right
: is a 2-column matrix of coordinates of points.

result
: is a 2-column matrix of points on the best-fit function.

Effect

The function calculates the best fit to the points in its right argument by a straight line (a linear function). It leaves a character description of this function in the variable `fitmsg` and sets `fitpts` to *right*. The *result* matrix is typically used as input to `FIT`.

Example

See Choosing a Function for a Least-Squares Fit.

SLABEL

SLABEL

Effect

The function writes default numeric labels on a triple of X-, Y-, and Z-axes for surface and skyscraper charts.

SLBLX

X `SLBLX` *right*

X
: is a scalar or a vector that describes an axis and its tick marks in the same way as for the function `AXIS`. (Matrixes and arrays of rank 3 are not allowed as left arguments.)

right
: is a scalar 0, numeric vector, or character vector or matrix.

Effect

The function writes labels for tick marks on an X-axis for a surface chart or a skyscraper chart. Otherwise its effect is similar to `LBLX`.

SLBLY

This is nearly identical to SLBLX, but draws a Y-axis.

SLBLZ

This is nearly identical to `SLBLX`, but draws a Z-axis.

SPLINE

result ← `SPLINE` *right*

right
is a 2-column matrix of coordinates of at least four points.
result
is a 2-column matrix of points on a spline curve through the points.

Effect

The function calculates a spline curve through the points in its right argument. The *result* matrix is typically used as input to `FIT`.

During execution, the function requests additional input from the user, to specify certain characteristics of the curve. For a description of allowable responses and their effects, see Additional Data for SPLINE.

Example

See SPLINE Fits a Spline Curve.

SPLOT

left `SPLOT` *right*

left

is a character vector whose elements control aspects of the output graph. The characters listed below may appear in the vector in any order, but the order in which digits appear will affect the result.

Character	Meaning
`A`	Do not draw the axes.
`F`	Fill the area under the curve.
`f`	Fill the area under the curve and edge the curve.
`L`	Label the axes.
`P`	Plot the points of the graph; do not plot a line.
`p`	Plot the points of the graph connected by a line.
`R`	Plot relative graphs.
`S`	Use the scaling factors implied by the current problem- space window (`w`) and the scaling viewport (`svp`).
`X`	Make the X-axis logarithmic to the base 10.
`Y`	Make the Y-axis logarithmic to the base 10.
`0-9`	Use the corresponding element of the plot attribute vector (`pa`) as an attribute code for the output graph. (0 corresponds to element 10 of `pa`.) Digits in the character vector *left* are taken from left to right to point to codes for the graphs produced.

right

is a numeric vector, matrix, or array. If *right* is
a vector
its values are taken as Y-values and its indexes as X-values.
a matrix
the values in its first column are taken as X-values and the values in its remaining columns as sets of corresponding Y-values.
an array of rank 3 or higher
it defines a set of several graphs to be plotted, as for the function `PLOT`. Typically, the array is generated by the functions `AND` and `VS`.

Effect

The function plots one or more (superimposed) graphs specified by the right argument *right*. The grid-control variable (`tm`) controls the extension of tick marks across the plot to make a reference grid. The plot character vector (`pc`) controls the characters used to plot points (when characters `P` or `p` appear in the left argument).

Example

See A Note on APL2 Functions.

SS

left `SS` *right*

left

is a character vector whose elements control aspects of the output chart. The characters listed below may appear in the vector in any order.

Character	Meaning
`A`	Do not draw the axes.
`F`	Fill the columns.
`f`	Fill and edge the columns.
`G`	Draw a base grid.
`L`	Label the axes.
`S`	Use the scaling factors implied by the current surface window (`sw`) and the scaling viewport (`svp`).

right

is an array `M` of heights to be plotted. If `M` has *m* rows and *n* columns, then `M[1;1]` is the height corresponding to the X,Y-coordinates (1,1), and `M[`*m*`;`*n*`]` is the height corresponding to (*n*, *m*).

If *right* is a scalar or vector, it is treated as a matrix of one row.

Effect

The function plots a skyscraper chart of the data in *right*. The surface window (`sw`) controls not only the data limits in the X-, Y-, and Z-directions, but also the slopes and spreads of the X- and Y-axes and the attribute values for filling columns.

The X- and Y- axes are delimited by the lower and the right and left edges respectively of the scaling viewport. Given the axis slopes, this determines the position on the screen on the far corner of the base rectangle. The vertical screen space remaining between the corner and the upper edge of the scaling viewport is used for the Z-axis. Note that the Z-axis may thus get mapped on a very short or even downward axis if the X- or Y- axis is steep or the height/width ratio of the viewport is small.

Example

See SS Draws Skyscraper Charts.

STEP

left `STEP` *right*

left

is a character vector whose elements control aspects of the output graph. The characters listed below may appear in the vector in any order.

Character	Meaning
`A`	Do not draw the axes.
`B`	Accept a 3-column matrix for the right argument, and plot disconnected steps.
`F`	Fill the area under the graph.
`f`	Fill the area under the graph and edge the graph.
`L`	Label the axes.
`R`	Plot relative graphs.
`S`	Use the scaling factors implied by the current problem- space window (`w`) and the scaling viewport (`svp`).
`X`	Make the X-axis logarithmic to the base 10.
`Y`	Make the Y-axis logarithmic to the base 10.
`0-9`	Use the corresponding element of the plot attribute vector (`pa`) as an attribute code for the output graph. (0 corresponds to element 10 of `pa`.) Digits in the character vector *left* are taken from left to right to point to codes for the graphs produced.

right

is a numeric vector, matrix, or array. If *right* is
a vector
its values are taken as Y-values and its indexes as X-values.
a matrix
the values in its first column are taken as X-values and the values in its remaining columns as sets of corresponding Y-values.
an array of rank 3 or higher
it defines a set of several graphs to be plotted, as for the function `PLOT`. Typically the array is generated by the functions `AND` and `VS`.

If `B` is used in *left*, then *right* must be a matrix of three columns. In this case the first two columns contain the beginning and end X-values for steps, and the third column contains the Y-values.

Effect

The function plots one or more (superimposed) step functions. The grid-control variable (`tm`) controls the extension of tick marks across the plot to make a reference grid.

If the intervals between successive X-values are equal, the plot is extended one interval to the right to emphasize the final Y-value. If the intervals between the X-values are not equal, the final Y-value is indicated by an arrow pointing to the right.

Example

See STEP Draws Step Charts.

STEREO

left `STEREO` *right*

left
is a 3-element numeric vector giving the following distances, in problem-space units.

- The offset of the object to the right and left of the Z-axis in order to produce parallax
- The distance of the observer from the origin (the Z-coordinate of the vantage point)
- The separation between right and left images on the display screen

right
is a 4-column vector or matrix. The last three numbers in each row are the X-, Y-, and Z-coordinates of a point. The first number is an attribute code for a line drawn to that point.

Effect

The function draws the right and left images of a stereoscopic view of the object defined by *right*, from the vantage point given by the second element of *left*.

Example

See STEREO Draws Stereo Projections.

STITLE

This is nearly identical to `TITLE`, except that it writes titles for surface charts and skyscraper charts, and requires the first *three* elements of its left argument to specify the X-, Y-, and Z-coordinates of the center of the title. The meanings of elements 4, 5, 6, ... of the left argument are identical to the meanings of elements 3, 4, 5, ... of the left argument for `TITLE`.

STYLE

result ← `STYLE` *right*

right
: is a scalar or numeric vector of values for the second attribute (*style*) of a set of lines or a fill pattern on the display screen. Negative values are interpreted by `STYLE` as attribute codes. Positive values are interpreted as new style values, and must precede any negative values.

result
: is a scalar or numeric vector of negative attribute codes.

Effect

`STYLE` inserts the new style values (positive elements of *right*) into the other attribute codes (negative elements of *right*). It uses the positive elements repeatedly to fill all the negative elements, or it creates new negative elements as needed to take all the positive elements.

`STYLE` is typically used with `USE` to change the attribute vector `av` and with `USING` to change an attribute value temporarily. It may be used together with `COLOR`, `WIDTH`, and `MODE`.

Example

See Specifying Color, Style, Width, and Mode.

SURFACE

left `SURFACE` *right*

left

is a character vector whose elements control aspects of the output chart. The characters listed below may appear in the vector in any order.

Character	Meaning
`A`	Do not draw the axes.
`L`	Label the axes.
`S`	Use the scaling factors implied by the current surface window (`sw`) and the scaling viewport (`svp`).
`X`	Extend traces across the surface parallel to the X-axis.
`Y`	Extend traces across the surface parallel to the Y-axis.

right

is a matrix `M` of heights to be plotted. if `M` has *m* rows and *n* columns, then `M[`*m*`;1]` is the height corresponding to the X,Y-coordinates (0,0), and `M[1;`*n*`]` is the height corresponding to (*n*-1, *m*-1).

Effect

The function plots a surface chart of the data in *right*. The surface window (`sw`) controls not only the data limits in the X-, Y-, and Z-directions, but also the slopes and spreads of the X- and Y-axes.

Example

See SURFACE Draws Surface Charts.

SXFM

result ← `SXFM` *right*

left
: is a vector of 3 elements, or a matrix of rows of length 3, representing one or more points in the 3-dimensional space used by `SS` and `SURFACE`.

result
: is a vector of 2 elements, or a matrix of rows of length 2, representing the corresponding points in problem space (as determined by `W`).

Effect

The function transforms its argument by the same transformation that `SS` and `SURFACE` use to map from 3-dimensional space into problem space.

Example

See SXFM Maps From Three-dimensional Space to the Window.

THREEVIEWS

`THREEVIEWS` *right*

right
: is a 3- or 4-column vector or matrix, like the right argument of `SKETCH`.

Effect

The function extracts projections of the object defined by *right* on the X,Y-, X,Z-, and Y,Z-planes, and draws them in the graphic field to a suitable scale.

Example

See THREEVIEWS Sketches Three Projections.

TITLE

left `TITLE` *right*

left

is a numeric vector `L` whose elements control the position and attribute values of a title.

Element	Specifies
`L[1 2]`	the X- and Y-coordinates of the center of the title, in problem-space units.
`L[3]`	the attribute code for the border. (0 specifies no border.)
`L[4]`	the attributes of the background fill. (0 specifies no fill.)
`L[5]`	an alternative position for the title.

Value	Meaning
`0`	put the center of the title at `L[1 2]`.
`1`	center the title above the scaling viewport.
`2`	center the title with respect to the X-axis, at Y=`L[2]`.
`3`	center the title with respect to the Y-axis, at X=`L[1]`.
`4`	put the title above the scaling viewport, at X=`L[1]`.
`<0`	`L[1 2]` specifies the location of one of the following:

`¯1`: the lower left corner of the title box.

`¯2`: the center of the bottom edge of the title box.

`¯3`: the lower right corner.

`¯4`: the center of the right edge.

`¯5`: the upper right corner.

`¯6`: the center of the top edge.

`¯7`: the upper left corner.

`¯8`: the center of the left edge.

`L[6]`	whether to produce default legends, as follows: `0`: do not produce default legends. `1`: produce a default legend for each row of *right*.
`L[7 ...]`	attribute values of the default legends. These elements are indexes into the plot attribute vector (`pa`), and are repeated as needed to match the number of legends required. If these elements are omitted, successive elements of `pa` are used instead.

right

is a character vector or matrix. It may contain APL2 characters for the following special purposes:

¨ Begin a new line.

⊂ Begin a graphic identifier.

⊃ End a graphic identifier.

The characters used for these purposes are contained in the variable `dlc`.

The following characters may be used between the special delimiters as graphic identifiers, to display the keys listed:

Character	Displays	Example
P	plot symbol	○
p	plot symbol and line	- - ? - -
F	fill attribute values	÷ ÷ ÷ ÷ ÷
f	fill values and edging	? ? ? ? ?
E	edging values only	□ □ □ □ □
blank, empty	line attribute values	- - - - -

Effect

Produces a title for a graph, with position and other characteristics specified by *left* and text contained in *right*.

Example

See Creating Axes, Labels, Annotations, and Titles.

TRANSLATE

result ← *left* `TRANSLATE` *right*

left
: is a scalar, or a vector of 2 or 3 elements.

right
: is a vector or matrix of 2, 3, or 4 columns.

  If *right* has two columns, the numbers in each row are the X- and Y-coordinates of a point.

  If *right* has three or four columns, the last two or three numbers in the row are the X- and Y-coordinates, or the X-, Y-, and Z-coordinates, of a point. The first number gives the attribute code for a line drawn to that point.

result
: is a 3- or 4-column vector or matrix, structured as input to `SKETCH`, that represents the same figure as *right*, translated by the elements of *left* along each axis. If the left argument is a scalar, it is extended to match the number of coordinates in the right argument.

Example

See Using the New Coordinates.

USE

`USE` *right*

right
: is a scalar or numeric vector or matrix.

Effect

`USE` modifies the attribute vector `av`. If *right* is

empty (`''`)
: the elements of the attribute vector `av` are made to vary systematically through the range of attributes and values specified by `dd[11 12]`.

a scalar
: *right* is extended to match the length of `av`, and attribute codes in *right* replace those in `av`. The values not specified in *right* remain unchanged in `av`.

a vector
: `av` is set to a vector of the same length as *right*. Positive elements of *right* are used as indexes of elements of `av` to be retained (in the positions in which they appear in *right*. Negative elements of *right* are put in the new attribute vector in the positions in which they appear in *right*.

a matrix
: each row is a set of values for different attributes. The last element in each row is the value of the first attribute (*color*), the next to the last element is the value of the second attribute, and so on. `av` is replaced by a vector with one element for each row of *right*.

`USE` is typically used with `COLOR`, `STYLE`, `WIDTH`, and `MODE` to change the attribute vector (`av`).

Example

See Specifying Color, Style, Width, and Mode.

USING

result ← *left* `USING` *right*

left
: is a numeric matrix of 2 or 3 columns that can be used as an argument of `DRAW`, or a numeric or character vector or matrix that can be used as a right argument of `WRITE` or `TITLE`.

right
: is an attribute code.

result
: is a matrix that can be used as an argument of `DRAW` or `FILL`, or an array that can be used as a right argument of `WRITE` or `TITLE`, with the attribute code given in *right*.

Effect

The function specifies an alternate attribute code.

Its effect on attribute codes in *left* is controlled by the attribute modification control variable `amc`. If `amc=0`, `USING` replaces all nonzero code; if `amc=1`, `USING` replaces only negative codes.

Example

See Specifying Color, Style, Width, and Mode.

VER

`VER` *right*

Effect

Identical to `0 ANNY` *right* followed by `0 AXIS ''`.

VIEW

`VIEW`

Effect

Causes the graphpics processor to display the current contents of the graphic field and wait for a user input event. Events caused by function keys have the following effects:

Function	Causes
2	Return to APL (→)
3	Return to APL function (→0)
4	COPY

Any other event causes the same result as function key 3.

On workstation systems:

Any keystroke or mouse button press creates an event.

On CMS and TSO:

Any interrupt key creates an event.

Example

See VIEW.

VIEWPORT

result ← `VIEWPORT`

result

is a 2-column matrix. The first column is a set of X-coordinates (in virtual-space units); the second is a set of Y-coordinates. Together the two columns are the coordinates of successive vertices of the clipping viewport specified by `cvp`.

VS

result ←*left* `VS` *right*

left
: is a numeric vector or matrix of Y-values.

right
: is a numeric vector or matrix of X-values.

result
: is an array of the type used for `PLOT`.

Effect

The function, together with `VS`, structures data for the right argument of `PLOT` and `SPLOT`.

Example

See Plotting Several Graphs at Once with AND and VS.

WIDTH

result ← `WIDTH` *right*

right
: is a scalar or numeric vector of values for the third attribute (*width*) of a set of lines on the display screen. Negative values are interpreted by `WIDTH` as attribute codes. Positive values are interpreted as new width values, and must precede any negative values.

result
: is a scalar or numeric vector of attribute codes.

Effect

`WIDTH` inserts the new width specifications (positive elements of *right*) into the other attribute codes (negative elements of *right*). It uses the positive elements repeatedly to fill all the negative elements, or it creates new negative elements as needed to take all the positive elements.

`WIDTH` is typically used with `USE` to change the attribute vector `av` and with `USING` to change an attribute code temporarily. It may be used together with `COLOR`, `STYLE`, and `MODE`.

Example

See Specifying Color, Style, Width, and Mode.

WITH

result ← *left* `WITH` *right*

left
: is a numeric vector or matrix like that used for a right argument of `PIECHART`.

right
: is a character array like that used for a right argument of `PIELABEL`.

result
: is an array of rank 5 suitable as a right argument of `PIECHART`.

Effect

The function combines numeric data and label data into an argument for `PIECHART`. Its right and left arguments may be interchanged.

Example

See Drawing Bar Graphs, Step Charts, and Pie Charts.

WRITE

left `WRITE` *right*

left

is a numeric vector or matrix whose elements control the position and attribute values of text to be written. Each row of *left* is a vector `L` in which

Element	Specifies
`L[1 2]`	the X- and Y-coordinates (in virtual space) of the lower left corner of the first character to be written.
`L[3]`	the height of characters in virtual-space units. If `L` has fewer than 3 elements, the last specified character size is used. (What heights are possible depends on how characters are generated. For more information, see Character Generation.
`L[4]`	the direction of the line of text, as follows: `0` horizontal `1` vertically upward
`L[5]`	an angle of rotation of the actual text line counterclockwise from the direction set by `L[4]`. (What rotation is possible depends on how characters are generated. See Character Generation for more information.)
`L[6]`	The attribute codes for the text to be written.

right

is a numeric or character vector or matrix containing text to be written. It may also be an array of rank 4 that is a result of the function `USING`.

Effect

The function writes the text in *right* at the location on the display screen, and with the attribute values, specified by *left*.

If both arguments are matrices, each row of the *left* corresponds to a row of *right*.

If *left* is a vector and *right* is a matrix, *left* specifies the location of the last line of *right*. Preceding lines are written above the first line, with the same attribute values.

If *left* is a matrix and *right* is a vector, the text in *right* is duplicated as many times as needed to correspond to all the rows in *left*.

Example

See WRITE Writes Text.

XFM

result ← *left* `XFM` *right*

left
: is an array of three dimensions with two matrixes, representing a transformation of one space into another. Each row in the first matrix gives the coordinates of a point in the first space; the corresponding row in the second matrix gives the coordinates of its image in the second space.

right
: is a numeric matrix of two or three columns, representing a set of points to be transformed. (If the matrix has three columns, the first column gives attribute codes.)

result
: is a matrix of the same shape as *right*, representing a set of transformed points.

Effect

The function transforms a set of points in *right* by the transformation defined by the matrixes in *left*. If the points are represented with a first column of attribute codes, the codes are left unchanged.

Three points, not all on a straight line, are needed to define completely a transformation of one plane into another. If the matrixes in *left* specify more than three points, the transformation is over-defined, and will be computed by a least-squares method. In practice, *left* is typically constructed by the function `INTO`.

Example

See Changing Coordinates.

Variable Reference

This reference section contains an alphabetic list of the variables in GRAPHPAK that you may need to assign values to directly, with descriptions of their content and meaning.

amc	Attribute Modification Control
AP2WGRAPHPAK	Error Messages Array
av	Attribute Vector
bw	Bar Width
co	Color Order
copyctl	Copy Control Vector
copyname	Copy File Name
cvp	Clipping Viewport
dd	Device Dependent Parameters
dlc	Delimiter Codes
eps	Epsilon
gf	Graphic Field
lga	Log Axes
ofx	Annotation Offset From X-axis
ofy	Annotation Offset From Y-axis
pa	Plot Attribute Vector
pc	Plot character vector
pie	Pie Chart Controls
sbw	Skyscraper Bar Width
sci	Scissoring Control
sf	Solid Fill Control
smf	Session Manager Field
svp	Scaling Viewport
sw	Surface Window
tm	Grid Control
VFONT	Vector Font
w	Problem Space Window

amc - Attribute Modification Control

Scalar that controls what the `USING` function does to the first column of its left argument. If `amc=0`, `USING` replaces all nonzero values; if `amc=1`, `USING` replaces only negative values.

AP2WGRAPHPAK - Error Messages Array

Character matrix that contains all the error messages produced by GRAPHPAK.

av - Attribute Vector

Numeric vector that specifies the values of attributes used for graphic output.

Each element of `av` is a negative number that specifies a set of values for one or more attributes. Values are coded in this way:

- The rightmost two digits specify the value of attribute 1 (*color*).
- The two positions to the left of *color* specify the value for attribute 2 (*style*).
- The two positions to the left of *style* specify *width*.
- The two positions to the left of *width* specify *mode*.
- If the number has fewer than seven digits, attribute specifications are omitted in the order *mode*, *width*, *style*.
- The value 00 in any position is interpreted as 01.

For example, the number `¯20005`, as an element of `av`, would specify color 5, style 1, and width 2.

The function `USE` modifies the content of `av`.

bw - Bar Width

Scalar that controls the width of bars produced by the `CHART` function. The value, between 0 and 1, is the width of the bar as a fraction of the distance between successive bar locations. When `bw=1`, adjacent bars touch.

co - Color Order

Numeric vector that contains the correspondence between default colors and color numbers. It is used only in setting a default attribute vector (`av`) with the `USE` function.

Its initial value, as distributed with the GRAPHPAK workspace, is 7 2 4 1 6 5 3 8. This means that the first default color is color number 7, the second is color number 2, the third is color number 4, and so on. For a description of the actual colors, see ATTRIBUTES.

copyctl - Copy Control Vector

An 11-element numeric vector that controls aspects of the output from the COPY and COPYN functions. Its 11 elements have the following default values and meanings:

Element	Value	Meaning
1	0	(None; must be zero)
2	0	No heading on page (1 means put heading on page)
3	1	Number of copies
4	66	Page depth
5	0	Top margin
6	0	Left margin
7	0	Bottom margin
8	80	Characters per line
9	0	Translation code (parameter for GDDM ASTYPE command)
10	47	Number of lines on printer (maximum=80)
11	80	Number of columns on printer (maximum=132)

It should be noted that some care must be exercised in choosing the copy parameters. If image symbols are used, they are the same number of pixels high on the printer as on the display. However, because of scaling, they may not conform to the rest of the picture. For example, if printing is called for on a small area of the paper, the characters will appear large by comparison with the graphical portions. In fact, successive lines of text may overlap. If this happens, one can either choose a larger size paper area or use vector symbols. However, vector symbols may appear distorted because they are computed to the nearest pixel value as on the display.

Values used in this variable must conform to the conventions of the Graphical Data Display Manager. For details, see the Graphical Data Display Manager User's Guide.

Note: The `copyctl` variable is only used on CMS and TSO.

copyname - Copy File Name

This variable specifies the print destination for the COPY function. If copyname begins with a blank or asterisk, COPY will prompt for a destination. The other meanings of its value are operating system dependent:

On OS/2 and Windows:

copyname is either a file specification of the form:

```
[path]filename
```

or a null character vector. If a file specification is supplied, a metafile is written. If a null character vector is supplied, the default printer is used.

On TSO:

copyname is the address of the printer.

On CMS:

copyname is a character vector of up to 8 characters that names a file that will be created and stored. Later it can be printed by the print utility program ADMOPUV.

On Unix Systems:

COPY is not supported on these operating systems.

cvp - Clipping Viewport

Numeric matrix that defines a region of virtual space outside of which displays can be inhibited. The function `VIEWPORT` displays the current contents of `cvp` as a 2-column matrix showing coordinates of the vertices of the clipping viewport. The function `FIXVP` modifies `cvp` to define a region with a given set of vertices. Whether or not display is inhibited outside of the clipping viewport is controlled by the variable `sci`.

dd - Device Dependent Parameters

A 12-element numeric vector that contains device-dependent parameters.

Element	Specifies
`dd[1 2]`	maximum screen size in units of virtual space (0,100).
`dd[3 4]`	actual screen address corresponding to (100,100) in virtual-space units.
`dd[5 6]`	horizontal and vertical space from one character to the next in virtual-space units.
`dd[7 8]`	horizontal and vertical character cell size in virtual-space units.
`dd[9]`	writing mode: `0`: use normal character generator `1`: generate stroke characters `2`: use vector character generator
`dd[10]`	not used.
`dd[11]`	number of attributes available.
`dd[12]`	maximum number of values for each attribute. The numbers are encoded in base 100.

`WRITE` uses `dd[8]` as the size factor if none is given explicitly and always sets `dd[5 6 7 8]` to reflect the last size used.

dlc - Delimiter Codes

A 4-element character vector that contains the delimiter characters.

Element	Initial Value	Delimits
dlc[1]	¨	New lines.
dlc[2]	⌶	New labels.
dlc[3]	⊂	Beginning of key (for TITLE)
dlc[4]	⊃	End of key.

eps - Epsilon

Scalar that modifies the behavior of contours around grid points. It is usually set to a small number.

gf - Graphic Field

A 4-element numeric vector that controls where on the screen graphic output will appear.

Element	Specifies
`gf[1 2]`	row and column of the upper lefthand character position on the screen.
`gf[3]`	number of rows in the graphic field.
`gf[4]`	number of columns in the graphic field.

The value is initialized by `OPENGP` to describe the entire screen.

Note: `gf` is only supported on CMS and TSO.

lga - Log Axes

A 2-element numeric vector that controls whether axes are linear (value = 0) or logarithmic (value = 1). The first element controls the X-axis, the second the Y-axis.

Note: Make sure that `lga` is zero before setting `w←svp` to allow placement of titles in virtual coordinates.

ofx - Annotation Offset From X-axis

A 2-element numeric vector that specifies the default position for annotations on the X-axis. `RESTORE` restores the defaults.

Element	Specifies
`ofx[1]`	`0`: the axis is below the top of the scaling viewport. The default annotation position is below the axis. `1`: the axis is at or above the top of the scaling viewport. The default annotation position is above the axis.
`ofx[2]`	The number of virtual-space units from the default position to the axis.

ofy - Annotation Offset From Y-axis

A 2-element numeric vector that specifies the default position for annotations on the Y-axis. `RESTORE` restores the defaults.

Element	Specifies
`ofy[1]`	`0`: the axis is to the left of the right side of the scaling viewport. The default annotation position is to the left of the axis. `1`: the axis is at or to the right of the right side of the scaling viewport.
`ofy[2]`	The number of virtual-space units from the default position to the axis. The default annotation position is to the right of the axis.

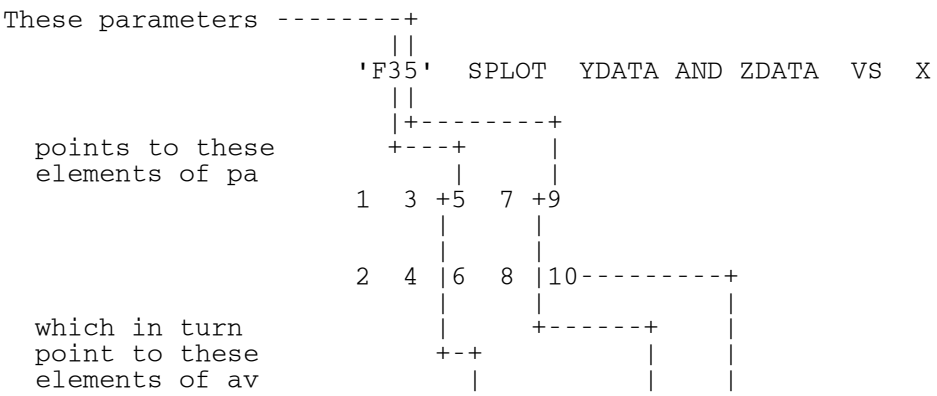
pa - Plot Attribute Vector

Numeric vector or 2-row matrix that contains attribute codes for plotting and charting functions.

If pa is a matrix, then its first row specifies fill patterns and its second row edge attributes.

Functions that use control characters in a left argument (SPLOT, CHART, and so on) can use the digits 1 through 9 and 0 to refer to the corresponding column in pa. (0 refers to column 10 of pa.)

Schematically, the left argument of SPLOT, the matrix pa, and the attribute vector av are related like this:



RESTORE resets pa to either of two values, depending on the number of colors and attributes of the output device.

pc - Plot Character Vector

Character vector that specifies the characters to be used in plotting points by `SPLOT`. Its initial value, as distributed with the GRAPHPAK workspace, is `'○□⋆∘×▽△'`. The initial value is reset by `RESTORE`.

pie - Pie Chart Controls

A 6-element numeric vector that controls location, size and other features of piecharts.

Element	Specifies
`pie[1 2]`	X and Y coordinates of the pie center in virtual-space units.
`pie[3]`	pie radius in virtual-space units. (Default is 35% of the shortest dimension of the screen.)
`pie[4]`	starting angle in degrees, measured from 3 o'clock. Default is 0.)
`pie[5]`	the angular span in degrees. (Default is 360.) If the value is negative, the slices of the pie will be plotted in clockwise order.
`pie[6]`	an attribute code for filling label boxes placed inside pie slices.

sbw - Skyscraper Bar Width

A 2-element numeric vector that specifies the width of columns for skyscraper charts in the X direction (first element) and Y direction (second element). Each value, between 0 and 1, specifies the width of the column as a fraction of the distance between column locations.

sci - Scissoring Control

Numeric scalar that controls the display of points, lines, and filled areas located outside the clipping viewport defined by `cvp`.

Value Meaning

`¯1` display all points. Results for points outside `cvp` are invalid.

`ÿ0` replace points outside `cvp` by the nearest boundary point.

`.5` discard lines wholly or partially outside `cvp`.

`ÿ1` discard parts of lines or filled areas that fall outside `cvp`.

sf - Solid Fill Control

Scalar that controls whether fill patterns will (value = 0) or will not (value=1) be allowed to show through a superimposed pattern. If earlier patterns are not allowed to show through, the area to be filled is first filled with a neutral background. This means passing twice as much data to the graphics processor, and could be noticeably slower. The initial value as distributed with the workspace is `sf=1`.

smf - Session Manager Field

A 4-element numeric vector that controls what portion of the screen is given to the APL2 Session Manager.

Element	Specifies
`smf[1 2]`	row and column of the upper left-hand corner of the session manager field.
`smf[3]`	number of rows in the field. (Must be at least 3.)
`smf[4]`	number of columns in the field. (Must be at least 40 and no more than 79.)

Note: `smf` is only supported on CMS and TSO.

svp - Scaling Viewport

A 4-element numeric vector that specifies the screen area in which charts are displayed. Its elements are coordinates in virtual space. The first two specify the lower left corner of the scaling viewport, and the last two the upper right corner. For a diagram of its elements, see The Scaling Viewport.

Functions whose output is fitted to the scaling viewport include `PLOT, SPLOT, CHART, STEP, PIECHART, CONTOUR, SURFACE,` and `SS`.

sw - Surface Window

An 11-element numeric vector that controls surface and skyscraper plots.

Element	Specifies
`sw[1 2]`	data limits in Z direction.
`sw[3 4]`	maximum X and Y coordinates.
`sw[5 6]`	slope of X and Y axes.
`sw[7 8]`	spread factors of X and Y axes, used to minimize the obstruction of columns in the skyscraper function `SS`. (The default is 5 3.)
`sw[9 10 11]`	attribute values for right side, left side, and top of a skyscraper column.

tm - Grid Control

Scalar, or vector of one or two elements, that controls the extension of tick marks across the output of `PLOT`, `CHART`, `SPLOT`, and `STEP` to make a reference grid.

Value	Means
`0` or `0 0`	do not extend tick marks.
`1` or `1 1`	extend tick marks in both directions.
`1 0`	extend X-axis tick marks vertically.
`0 1`	extend Y-axis tick marks horizontally.

VFONT - Vector Font

Specifies the font to use when `dd[9] = 2`.

For information about using `VFONT` see FONTDEFINE.

w - Problem Space Window

A 4-element numeric vector that defines the window in problem space. Its elements are coordinates in problem space. The first two specify the lower left corner of the window, the last two the upper right corner. For a description of the window in problem space, see The Window in Problem Space.

Installation and Customization

This reference section describes some things you might want to do to tailor GRAPHPAK to your own needs. It includes

- What to Do with Symbol Sets
- Saving Space in a Private Copy of GRAPHPAK
- Changing the Session Manager and Graphic Fields

For details specific to CMS and TSO, see the Installation and Customization Manual for your operating system.

What to Do with Symbol Sets

Note: This section applies to TSO and CMS only.

When you first get the GRAPHPAK workspace, it contains a set of tables that describe character symbols to your graphics device. These tables are called the symbol sets.

When GRAPHPAK is installed, store the symbol sets in an auxiliary storage location accessed by all users.

How to Install the Symbol Sets

1. On TSO, allocate data sets for the symbol sets. (This isn't necessary on CMS.)
2. Execute `WRITESS`.

For a description of the symbol sets, see Symbol Sets.

Using your own Symbol Sets

If you have tried to execute a GRAPHPAK function, and got the message

```
SYMBOL SETS DO NOT EXIST ON AUXILIARY STORAGE
```

then the symbol sets were not loaded to an auxiliary storage unit that is available to you. You can store your own copy of the symbol sets by using `WRITESS` in the same way as described above.

If you cannot get the symbol sets from auxiliary storage, a GRAPHPAK function will attempt to load them from the workspace you are using. If they are not in the workspace, processing will halt and a message will be displayed. If this happens, try to copy the group `GPSYMBL` from the public library copy of GRAPHPAK. If you cannot obtain the group `GPSYMBL`, you can still use GRAPHPAK, but you will be using a default symbol set that does not contain the APL characters.

It is possible to operate with the symbol sets in your workspace only. But the symbol sets will not be available to the printer for hard copy if they are not in auxiliary storage. Also the symbol sets take up, needlessly, about 25,000 bytes of storage in your workspace. Hence you will probably want to keep the symbol sets out of your working copy of GRAPHPAK. You can remove them by erasing the group `GPSYMBL`, as described in APL2 Group Variables.

Saving Space in a Private Copy of GRAPHPAK

To customize GRAPHPAK for your own use, load the workspace and save it in your private library. For example,

```
)LOAD 2 GRAPHPAK
)SAVE GRAPHPAK
```

If you wish to save storage space when running GRAPHPAK, consider one or more of the actions listed below.

- Remove the comments.

  The function `DECOMMENT` simplifies this task: Enter `DECOMMENT` and reply `YES` to the question it asks.

  You may at some time want to modify functions in GRAPHPAK, or obtain an understanding of some function that is more thorough than this manual can provide. The comments are valuable for these purposes, and you will probably want to keep in your installation at least one copy of GRAPHPAK that retains the comments.

- Erase the group `GPDEMO`.

  ```
  )ERASE (GPDEMO)
  ```

  This group contains demonstration programs.

- After the symbol sets have been placed on auxiliary storage, erase the group `GPSYMBL`.

  ```
  )ERASE (GPSYMBL)
  ```

  Note: Symbol sets are only supplied in TSO and CMS.

Changing the Session Manager and Graphic Fields

Note: This section applies to TSO and CMS only.

The area of your screen in which graphic output is displayed is called the *graphic field*. If you are using the APL2 session manager, the graphic field has to be distinguished from the area of your screen that is under control of the session manager, called the *session manager field*.

The examples in this manual have been created using the session manager, with both the session manager field and the graphic field the size of the entire screen. In this situation, the screen is usually the session manager field: you can enter APL2 statements on the entire screen under control of the session manager. Likewise numeric and character output will display on the entire screen. Graphic output does not display until you execute `VIEW`. After executing `VIEW`, the screen is the graphic field, and displays graphic output. When you press Enter, the screen becomes the session manager field again.

Setting the Fields Used in this Manual

To set the graphic and session manager fields to the sizes used for the examples in this manual, enter

```
SMFIELD 1 1 32 79
GRFIELD 1 1 32 80
```

(The examples were created on an IBM 3279 Display Terminal, Model 3. On this model, with these settings, both fields are of maximum size and overlap. Use `VIEW` to switch from the session manager field to the graphic field.)

The function `SMFIELD` sets the session manager field. `GRFIELD` sets the graphic field. Use `SMFIELD` first; `GRFIELD` uses the session manager field settings to decide whether to share pages.

The arguments of these two functions are read as follows:

- The first two elements of the argument give the row and column on your screen of the upper left corner of the field. (The top row of your screen is row 1. The left edge is column 1.)
- The third element is the number of screen rows in the field.
- The fourth element is the number of characters in each row.

The session manager field must be at least three rows deep, and not less than 40, nor more than 79, characters wide.

These two functions set, respectively, the global variables `smf` and `gf`. By examining these variables you can tell what settings your fields have at any time. (You cannot change the field settings by assigning values to the variables; the assignment does not communicate the change to the session manager.)

Other Possibilities

For an alternative setting of the fields, try this:

```
SMFIELD 1 1  6 79
GRFIELD 7 1 26 80
```

This setting will put the top six lines of your screen under control of the session manager, leaving the last 26 lines for the graphic field. You can enter at the top of your screen commands that produce graphic output (`PLOT`, `DRAW`, and so on), and the results will display at the bottom of your screen immediately. After the first use of `VIEW` in a session, you don't have to use it again.

As this example suggests, there are numerous ways to divide your screen between the session manager and the graphic fields. Experiment a bit to find the one you like best.

For further information on the APL2 session manager, see the *APL2 Programming: System Services Reference*.

APL2 Group Variables

The GRAPHPAK functions use, and share, many routines that are not explicitly named or described in *APL2 GRAPHPAK: User's Guide and Reference*. Hence you cannot safely erase a function - it might be used by some other routine. Neither can you expect to copy a single function from GRAPHPAK to another workspace and have it perform as it would in GRAPHPAK. What you *can* copy is an APL2 group variable. See *APL2 Programming: Language Reference* for a description of how to use group variable to copy group objects into workspaces.

The functions and variables in GRAPHPAK are organized into seven group variables. One of these is the group variable `GPDEMO`; it is independent of the other group variables and may safely be erased if you don't want it. The other six group variables are interdependent, and the following principles apply to using them:

- *Don't erase any objects listed in the variables.* They may be needed by other groups, for example.

  ```
  )COPY 2 GRAPHPAK (GPBASE)
  ```

  copies all the names in the matrix `GPBASE`.

- *You may copy any of these group variables into another workspace.* Objects in other group variables it includes will be copied with it.

The interdependent groups in GRAPHPAK are these:

`GPBASE`
: contains fundamental drawing and writing functions, including `WRITE, READ, FILL,` and `DRAW`. It is contained in each of the other group variables.

`GPGEOM`
: contains the descriptive geometry functions described in Drawing Three-dimensional Objects. It contains `GPBASE`.

`GPPLOT`
: contains functions for plotting, including `PLOT` and `SPLOT` described in Getting Started. It contains `GPBASE`, and is contained in the last three group variables in this list.

`GPCHT`
: contains functions for drawing charts, including `CHART, SURFACE, SS, PIECHART, STEP,` and `HCHART`. It also contains `GPPLOT` and `GPBASE`.

`GPFIT`
: contains the functions for fitting curves described in Fitting Curves. It also contains `GPPLOT` and `GPBASE`.

`GPCONT`
: contains functions for drawing contour maps, including `CONTOUR`. It also contains `GPPLOT` and `GPBASE`.

Device Dependencies

This reference section lists elements of GRAPHPAK that contain information that is dependent on particular types of hardware.

- Device-dependent Functions
- Device-dependent Variables
- Symbol Sets
- Character Generation
- ATTRIBUTES

The descriptions given for CMS and TSO are specific to the IBM 3278 and 3279 Display Terminals and the IBM 3287 Printer with the Programmable Symbol Set feature.

Device-dependent Functions

The functions listed here are coded to use a particular type of device, and may need modification if a different type is used.

`CGWRT`
: Creates the orders needed to place character generator text on a display, and passes the orders to `OUTPUT`.

`COPY, COPYN`
: See the descriptions of these two functions in Function Reference.

`ENCODE`
: Takes the coordinate and attribute information passed to `DRAW` and codes it into the graphic orders needed by the output device. It transforms from virtual space to real space.

`FILLENCODE`
: Takes the coordinate and attribute information passed to `FILL` and codes it into the graphic orders needed by the output device.

`FSSAVE, FSSHOW`
: See the descriptions of these two functions in Function Reference.

`GRFIELD, SMFIELD`
: See the descriptions of these two functions in Changing the Session Manager and Graphic Fields.

`INDEV0, INDEV1, INDEV2, INDEV3`
: These functions perform the several options of `READ`. The numbers in the function names may serve as device types. If functions are to be written to support other devices, the following conventions are suggested for device types:

  0 a position indicator, like a text cursor or current beam position.

  1 a position indicator controlled by the user, like the cursor on a keyboard.

  2 reserved.

  3 arbitrarily positioned text input from a keyboard.

`OPENGP`
: Initializes several items; it

  - Initiates sharing with the graphics processor.
  - Sets the aspect ratio of the screen in `dd[1 2]`.
  - Sets the number of attributes supported by the graphics device and the number of values possible for each attribute, in `dd[11 12]`.
  - Sets the clipping viewport in `cvp` to the boundaries of the actual screen.
  - Sets the graphics field in `gf` to fill the entire screen.
  - On CMS and TSO loads the symbol sets into GDDM from auxiliary storage. (The symbol sets describe character symbols to the graphics device. See What to Do with Symbol Sets for more information.

`OPENGP145`
: Initializes the same items as `OPENGP` and opens the graphic window in an AP 145 window. The argument to `OPENGP145` is passed to the AP 207 OPEN command. For further information, consult the function's comments and the DEMO_GRAPHPAK function in the DEMO145 workspace in public library 2.

  `OPENGP145` is only available on Windows.

OUTPUT

Passes data to the graphics processor for output via the variable IN.
It is possible to use OUTPUT to directly issue commands to the graphics processor.

On workstation systems:
Auxiliary processor 207 is the APL2 workstation graphics processor. AP 207 is documented in the APL2 User's Guide for your operating system.
To issue an AP207 command, assign the command(s) to the variable IN and execute OUTPUT.

```
IN←'VIEW' ''
OUTPUT
```

On CMS and TSO:
Since, in general, the GDDM auxiliary processor requires numeric data to be passed through CTLs and literal data through DATs, a coding scheme is used. The last element of IN contains the number of elements of IN which are to be passed to CTLs, while the remainder of the elements are turned into literal characters by indexing into ⎕AV (origin 0). The function GEP determines the entry points to the GDDM program through a table look-up procedure. For example, to find the entry point for the GDDM program GSCOPY, execute

```
A←GEP 'GSCOPY'
```

A will contain the number corresponding to the call of GSCOPY. If the called GDDM program has numeric parameters, they are appended to the numeric call and passed through CTLs. If the called GDDM program has literal parameters, they are passed through DATs.

For example, to execute the GDDM command to save the current segment and save it in a file called PICTURE, execute (in origin 0)

```
IN←GEP 'FSSAVE'
IN←IN,⎕AV⍳8↑'PICTURE'
IN←IN,1
OUTPUT
```

PRINTFILE

Note: This function is only supported on CMS and TSO.

See the description of this function in Function Reference.

Device-dependent Variables

The variables listed here contain coordinates in real space that depend on the type of the output device. They may need modification if a different type is used.

`apFSM`
: Scalar that contains the number of the graphics auxiliary processor (126 on mainframes and 207 on workstations.)

`apSM`
: **Note:** This variable is only supported on CMS and TSO.

  Scalar that contains the number of the APL2 Session Manager auxiliary processor (typically, 120).

`cg∆r`
: Defines the rotation capabilities of the character generator. If there are no rotation capabilities, this variable may be undefined.

`cof`
: **Note:** This variable is only supported on workstation systems.

  Defines the meanings of colors 7 and 8.

  The default value of `cof` is 0. The value 0 associates colors 7 and 8 with the the auxiliary processor 207 color names NEUTRAL and BACKGROUND. The meanings of NEUTRAL and BACKGROUND are device and operating system dependent. For example, on Windows, BACKGROUND is white on both screen and printer. On AIX, BACKGROUND is black on the screen and white on the printer.

  Setting `cof` to 1 associates colors 7 and 8 with the the auxiliary processor 207 color names WHITE and BLACK. `cof` should be set to 1 if you need to guarantee white or black colors regardless of device.

`cvp`
: See the description of this variable in Variable Reference.

`dd`
: See the description of this variable in Variable Reference.

`mode`
: **Note:** This variable is only supported on workstation systems.

  Defines the video mode.

  `mode` is a three element integer vector. The three elements are defined as follows:

  `[1]` requested video mode
  `[2]` video mode before last setting
  `[3]` current video mode

  Consult the APL2 User's Guide for your system to determine video modes supported by your system's auxiliary processor 207.

Symbol Sets

This section applies to TSO and CMS only.

GRAPHPAK functions use the following symbol sets for supporting different types of devices and sizes of characters. The last character of the image symbol set names indicates with what type of device the symbol set is used. Consult the *GDDM Base Programming Reference* for further information about symbol sets.

APL1ISSA	9 x 16
APL1ISSC	9 x 12
APL1ISSG	9 x 12
APL1ISSK	20 x 24
APL1ISSN	8 x 16
APL1ISSR	9 x 12
APL2ISSA	18 x 32
APL2ISSC	18 x 24
APL2ISSG	18 x 24
APL2ISSK	24 x 48
APL2ISSN	16 x 32
APL2ISSR	18 x 24
APLFVSS	Scalable vector symbol set

Character Generation

GRAPHPAK allows three types of character generation. The type in effect is controlled by the global variable `dd[9]`.

Image Characters

```
dd[9]←0
```

These characters are defined on a pixel by pixel basis.

On workstation systems:

Image characters are built into auxiliary processor 207. The characters are a fixed size.

On CMS and TSO:

Image characters are stored in the symbol sets APL1ISSC and APL2ISSC. They can be modified by using the GDDM interactive symbol editor utility.

The first of these fonts has characters about 1.7 virtual-space units high. The second has characters about 3.4 units high. When image characters are used, the size of characters written with `WRITE` will be either 1.7 or 3.4 units, whichever is closer to the specified height.

The numeric values of the character heights depend on the size and aspect ratio of the graphic field. The sizes are 1.7 and 3.4, where 32 screen lines equal approximately 71 virtual unit space units. With a smaller graphic field, the figures become larger, and `WRITE` may well select the smaller of the two fonts unless the size parameter is considerably greater than the usual 3.4.

`RESTORE` always resets `[5 6 7 8]` to the current nominal size of the smaller font.

Image characters cannot be rotated or magnified.

Stroked Characters

```
dd[9]←1
```

These characters are constructed by `WRITE` within GRAPHPAK, and are written by sending the strokes to the graphics processor. They can be both rotated and magnified.

Vector Characters

```
dd[9]←2
```

These characters are stored in font definition files. They are similar to the stroked characters, but are generated within the graphics processor instead of within APL2, and are correspondingly faster to write.

On workstation systems:

Vector characters are stored in APL2 vector font files. APL2 vector fonts are defined using the FONTDEFINE function. The variable VFONT specifies which defined vector font to use when drawing text.

On OS/2 and Windows, outline font files (PostScript or TrueType) can also be used as vector fonts.

On CMS and TSO:

Vector characters are stored in the symbol set APLFVSS, and can be modified by the GDDM interactive vector symbol utility.

ATTRIBUTES

Color

The device-dependent interpretation of color values is given in the list that follows. For each color is given its color number; the index of the element of the attribute vector `av` that specifies that color when `av` is set to default values; and the actual color displayed.

Number	av Index	Name
1	4	blue
2	2	red
3	7	pink (magenta)
4	3	green
5	6	turquoise (cyan)
6	5	yellow
7	1	neutral (white on display, black on printer)
8	8	background (black on display, white on printer)

The correspondence between the `av` indexes and the color numbers is determined by the variable `co` (color order).

Fill Patterns

The interpretation of fill patterns is determined by the graphics processor. GDDM provides 16 patterns. The number of fill patterns supported by auxiliary processor 207 is operating system and machine dependent. The AP207 QUERY command can be used to determine the number of fill patterns supported. When filling is called for, the style value of the attribute code may range from 1 to the maximum to select a fill pattern.

Messages

This reference section lists the messages the user can receive from either GRAPHPAK or an auxiliary processor when using APL2. These messages are stored in the global variable `AP2WGRAPHPAK` in the GRAPHPAK workspace.

The global variable `MORE` can be used to see the message number or any additional information available about an error. If you encounter an error message when using GRAPHPAK, enter `MORE` to view the message code and any additional information available

The messages are listed here in order by message number. With each appears further explanation and, where appropriate, suggestions for corrective action.

Some messages are operating system dependent.

AP2WGRAPHPAK001

"AND" LENGTH ERROR

Arguments of `AND` are not conformable. For example, in the expression `A AND B VS C`, the three vectors must have the same length.

AP2WGRAPHPAK002

RIGHT ARGUMENT OF "AND" MUST BE A HOMOGENEOUS GROUP

If the left argument of `AND` is a vector, the right argument must consist of curves all of the same length.

AP2WGRAPHPAK003

IF RT ARG OF "AND" IS A "VS" GRP, LEFT CANNOT BE A MATRIX

In the structure `Y1 AND Y2 VS X`, `Y1` cannot be a matrix. `Y1` must be a vector that has the same length as `X`.

AP2WGRAPHPAK004

0 IS INVALID ATTRIBUTE PARAMETER

Zero cannot be used as the right argument of `COLOR, STYLE, WIDTH,` or `MODE`.

AP2WGRAPHPAK005

ATTRIBUTE LENGTH ERROR

If two or more of `COLOR, STYLE, WIDTH,` or `MODE` are used in combination, their right arguments must have the same length or one must be a scalar.

AP2WGRAPHPAK006

NOT ABLE TO PRODUCE FUNCTION "FITFUN"

The function `FITFUN` is probably suspended. Check the state indicator by entering `)SI`. A list of function names and statement numbers will be displayed, some indicated by asterisks (*). Enter → as many times as there are asterisks in the list.

AP2WGRAPHPAK007

CURSOR OUTSIDE OF WINDOW

The `READ` function was used to return the location of the cursor but the cursor was outside of the graphics field.

AP2WGRAPHPAK008

SYMBOL SETS DO NOT EXIST ON AUXILIARY STORAGE

The symbol sets have not been found. An attempt will be made by the function to load the symbol sets from the workspace. However, if this is done, the symbol sets will not be available to the printer utility and default symbol sets will be used. (See Symbol Sets. for further discussion.)

AP2WGRAPHPAK009

DEVICE NOT SUPPORTED

A call of `READ` (graphical input) was made for an unsupported device.

AP2WGRAPHPAK010

ARGUMENTS OF "WITH" MUST BE OF OPPOSITE TYPE

`WITH` is used to pass numeric data and alphanumeric data to the `PIECHART` function. Its two arguments may appear in either order, but if one is numeric the other must be literal.

AP2WGRAPHPAK011

ARGUMENT OF "USE" MUST HAVE RANK LESS THAN 3

The function `USE` expects a scalar, vector, or matrix argument.

AP2WGRAPHPAK012

LEFT ARGUMENT OF "USING" MUST BE A MATRIX

The left argument of `USING` must be a matrix of data that can be passed to `DRAW` or `FILL`. For example,

```
DRAW DATA USING COLOR 5
```

AP2WGRAPHPAK013

"VS" ERROR

Length problem. The arguments of the function `VS` are not of the same length. Check the lengths and try again.

AP2WGRAPHPAK014

CANNOT DO LOG PLOT OF NON-POSITIVE NUMBERS

All data for a logarithmic plot must be positive, and at least one of your values is negative or zero.

AP2WGRAPHPAK015

"B" ON LEFT ONLY POSSIBLE IN "STEP" WITH 3 COLUMN ON RIGHT

If the control character `B` is used in the left argument of `STEP`, then the right argument must be a 3-column matrix. The first column is the vector of x-values to begin each interval, the second column is the vector of x-values to end each interval, and the third column is vector of y-values. (See STEP Draws Step Charts for further description.

AP2WGRAPHPAK016

INVALID "PIECHART" ARRAY

The right argument of `PIECHART` must be a numeric vector, a matrix, or a special rank 5 array produced by the `WITH` function.

AP2WGRAPHPAK017

SYMBOL SETS NOT FOUND IN WORKSPACE

Go back to an original version of GRAPHPAK to find the symbol sets. You can use GRAPHPAK even if the symbol sets are not found, but a default symbol set will be used that does not have the APL characters.

AP2WGRAPHPAK018

WARNING - IMAGE SYMBOL SET IN USE

A call to `PRINTFILE` was made to produce a PSEG from an image which used image symbol sets. `PRINTFILE` produces a file suitable for printing on Family 4 devices. No APL image symbol set supporting these devices is shipped with APL2. The text portions of the image will probably appear too small when printed. Set `dd[9]←2` to force GRAPHPAK to use a vector symbol set (which is device independent), reconstruct image, and recall `PRINTFILE`.

AP2WGRAPHPAK021

GDDM AP RETURN CODE ERROR:

The GDDM auxiliary processor (AP 126) detected an error and did not pass the information on to GDDM. The number following the message is the return code from the auxiliary processor. An explanation of the return code is given in the *APL2 Programming: System Services Reference* for your system.

AP2WGRAPHPAK022

GDDM RETURN CODE ERROR:

When this message occurs, it is followed on the next line by a message from GDDM about the last error it detected. If more than one error was detected by GDDM, previous errors are identified only by the GDDM error number; information about these can be found in the *GDDM User's Guide*.

If one passage of information from AP 126 to GDDM results in several errors, the order in which these errors are listed is not necessarily the order in which the corresponding GDDM calls were listed in AP 126.

APL processing is stopped and control is returned to the APL interactive session through a branch out (→).

AP2WGRAPHPAK023

GDDM RETURN CODE WARNING

As in message AP2WGRAPHPAK022

APL2 processing continues.

AP2WGRAPHPAK024

SESSION MANAGER AP RETURN CODE ERROR

The values passed to the APL session manager are invalid. The field defined must be at least 3 lines high and 40 characters wide and cannot exceed the limits of the screen.

AP2WGRAPHPAK025

GDDM AP NOT SHARING

The GDDM auxiliary processor (AP126) is not active; it must be active for you to produce a graphic display. (APL2 was probably not installed correctly.)

AP2WGRAPHPAK026

SYSTEM ABEND DETECTED IN FSQERR PROBABLY INSUFFICIENT SPACE

The amount of storage is not large enough to complete processing. Contact your system administrator.

AP2WGRAPHPAK027

DEVICE *name* NOT SUPPORTED

The specified device is not supported by GRAPHPAK. Specify a supported device or contact your system administrator.

AP2WGRAPHPAK028

MUST BE MORE THAN *number* DATA POINTS

You have specified an insufficient number of data points. Reissue the command with more data points than *number*.

AP2W207x

ERROR AP2W207*x* HAS OCCURRED

Auxiliary processor 207 has returned error code *x*. An explanation of the AP 207 return codes is given in the APL2 User's Guide for your system.

Glossary

annotation
: A caption on the axis of a plot.

array
: In APL, a systematic arrangement of elements. Arrays include scalars, vectors, matrices, and structures of higher dimension.

attribute
: A visual characteristic of a portion of a drawing. Common attributes are color, line style, line width, and fill pattern.

attribute code
: A code that represents, in one number, a value for each of the attributes.

bar chart
: A chart made up of horizontal bars.

character generator
: A device (software or hardware) that contains character shapes and can display them upon being passed a code.

clipping
: The truncation of a graphical object so that only the portion on one side of a line is displayed.

clipping viewport
: A polygon which limits the area to be displayed. Graphical objects or portions of graphical objects outside the clipping viewport are not displayed.

column chart
: A chart made up of vertical bars.

contour plot
: A set of curves that connect points of the same height value. The most common example is a topographic map.

curve fitting
: Finding a mathematical representation that in some sense comes closest to a set of experimental data. This is usually achieved by a least-squares procedure, but some methods represent curves that pass through every data point.

device dependent
: Depending on the particular display device. This can refer to code that varies according to the device or to the appearance of a picture that varies.

device-dependent function
: A function that uses data about a particular device.

empty vector
: A vector of zero length.

fill pattern
: A pattern with which an area is filled.

filling
: The rendering of an area with a uniform color or as a given pattern.

global variable
: In APL, a variable that is created by the execution of a function and erased when execution is complete is said to be **local** to that function. A variable in a workspace that is not local to any function is global.

histogram
: A graph that displays how many members of a set of data fall in each of several class intervals.

index

In APL, a number or set of numbers that tells the position of an element in an array. The index of an element of a vector is a single number; for example, the index of `7` in the vector `4 115 7 22` is `3`. There is one number in the index of an element for each dimension of the array. For example, if `A` is an array of rank 3, the number in the second matrix, fourth row, and third column is `A[2;4;3]`.

label
: Caption that identifies a tick mark.

latent expression
: In APL, a system variable containing an APL expression that is executed as soon as a workspace is activated.

legend
: A part of a title on a graph that identifies each set of data by the color or style in which it is displayed.

local variable
: In APL, a variable that is created by the execution of a function or operator and erased when execution is complete is said to be local to that function or operator.

matrix
: In APL, an array of two dimensions.

pie chart
: A rendering of proportional data as sectors of a circle.

pixel
: Smallest addressable point on a graphics device.

plot
: A graphic representation of a set of data.

polygon
: A figure composed of a set of straight lines. The end point of the last line coincides with the beginning point of the first line. Lines of a polygon can cross each other.

problem space
: The space of all points that can be represented by coordinates of the type used in describing the data for a problem.

rank
: In APL, the rank of an array is the number of its dimensions.

real space
: The set of distinguishable points on the display area of an output device.

scalar
: In APL, an array of zero dimensions, comprising a single element.

scaling
: Changing coordinate systems.

scaling viewport
: A subset of virtual device space that defines the portion of the display screen in which a plot is drawn.

scissoring
: Clipping at the edges of the clipping viewport to leave only the portion of a graphical display that is within the viewport.

shape
: In APL, vector giving the number of elements in each dimension of an array. The shape of `A` is given by `⍴A`.

skyscraper chart
: A chart made up of vertical prisms (skyscrapers) on a 2-dimensional base.

spline curve fitting
: Finding a mathematical representation of a curve that goes through each point in a set of data.

step chart

A plot that, instead of connecting adjoining points with a straight line, connects them with a horizontal and then a vertical line.

surface chart
: A chart of a function of two variables that shows Z-values along lines of constant X-value and of constant Y-value.

tick mark
: A small line drawn on the axis of a plot to indicate location of certain values.

transformation
: A change in representation from one coordinate system to another.

vector
: In APL, an array of one dimension.

virtual device space
: A coordinate system running from 0 to 100 in both the X- and Y-directions.

window
: A subset of problem space that includes the data for a particular problem.

writing
: Rendering characters on an output device.