True BASIC

The Original BASIC

Statistics Graphics Toolkit

Reference Guide

© 2002-2010 True Basic

TRUE BASIC REFERENCE SERIES:

Statistics Graphics Toolkit

Introduction

The *Statistics Graphics Toolkit*. It provides you with the tools you need to write good-looking, accurate software that has just the features you want. Use the toolkit routines both for doing statistical calculations and for graphic the results, or present the results in textual form. Let the routines handle the details; you just link them together with True BASIC statements.

There are hundreds of routines in this toolkit, and we include the source code for each one so that you can learn how theyy work and modify them to suit your needs. There are also many complete demo programs supplied with the toolkit to show you how to apply them.

Use the capabilities of True BASIC to import data from other applications, such as spreadsheets, into your statistical package.

Included in this Package

STAT1LIB	Main statistics toolkit
STAT2LIB	Statistical graphics
STAT3LIB	Nonparametric statistics
STAT4LIB	ANOVA, Fits, Regression
FRAMELIB	Low-level framing routines
MSGLIB	Mini-*Scientific Graphics Toolkit*
PRTLIB	Low-level routine to print screen
Demo programs	

© 2001 - True BASIC Inc. 04047-0501 USA. *All rights reserved.*

Installing the Statistics Graphics Toolkit

Your *Statistics Graphics Toolkit* disk is not copy-protected, so you can easily install it on your hard disk or local network. We suggest that you create a directory ("folder") especially for this Toolkit, and copy the entire diskette into this directory.

If you install this Toolkit on a network, please respect our intellectual property rights and call us to arrange for a site license.

This Toolkit includes a file named MSGLIB, a stripped-down form of the *Scientific Graphics Toolkit* file SGLIB. If you already own the *Scientific Graphics Toolkit*, change the LOADSTAT and STAT2LIB files to refer to your copy of SGLIB rather than MSGLIB. Then recompile STAT2LIB and replace its old compiled version.

Important Notes

Statistics is a large and complicated branch of learning, and this manual makes no attempt to introduce statistical concepts or explain *when* and *how* the various statistical tests should be used. We strongly recommend that you consult a good statistics book before using tests that you don't fully understand.

Notation

The notation used in statistics is not really standardized. What some people call the *phi* coefficient, for example, is called Cramer's V by others. We have chosen the most common names for use in this Toolkit; to attempt to satisfy all the different notations would be impossible.

Calculations

Worse yet, even the tests themselves are not quite standardized. For example, there are several ways to compute a chi-square statistic. Wherever possible, we have chosen the most common definition of a test as the default but have supplied Set routines that let you switch to other methods.

NOTE: If the results given by a routine in this Toolkit do not match what you expect, the Toolkit may be using a more advanced algorithm than given in your reference books. Nonetheless, check your data carefully!

Two-Tailed Probabilities

All probabilities computed by this Toolkit are *two-tailed*. If you want the one-tailed probability, divide by two.

Using This Toolkit

This Toolkit is designed for general statistics use, with particular attention to graphics and to nonparametric statistical measures. Since it's a large package, we give a "roadmap" below that describes which routines can be useful for given types of work.

Descriptive Statistics

Stats and GroupedStats for general statistics such as mean, standard deviation, etc. DataToFreq, TableToFreq, PlotHist, PlotCum, etc., for histograms and frequency polygons. PlotScat for scatter plots.

Exploratory Data Analysis

LetterValues and BoxPlot. PlotResid and PlotObsResid for graphing residuals. The section on "Data Transforms." ScatPlot with SetLineType("median") for resistant line-fitting and SetGraphType("logy") for semi-log scales. Friedman for nonparametric two-way ANOVA. The entire "Nonparametric Tests" section.

Regressions and ANOVA

The "Line-Fits, Regressions, and ANOVA" section. PlotScat, PlotObs, and their residuals versions. Friedman and KruskalWallisH for nonparametric tests.

Simulations

The "Simulated Distributions" and "Sampling" sections. The "Data Transforms" section may also be useful.

References

The books listed below were used in preparing the *Statistics Graphics Toolkit*, so you can be sure that their concepts and notations come close to matching those in this Toolkit.

Statistics, by D. Freedman, R. Pisani, and R. Purvis. W. W. Norton, 1978. A slow-paced, gentle introduction.

Beginning Statistics with Data Analysis, by F. Mosteller et al. Addison-Wesley, 1983. Clear introduction with interesting examples.

An Introduction to Mathematical Statistics and its Applications, by R. Larsen and M. Marx. Prentice-Hall, 1986. Good textbook for college mathematics courses.

Mathematical Statistics with Applications, by W. Mendenhall et al. Duxbury Press, 1986. Another good math textbook.

Understanding Robust and Exploratory Data Analysis, edited by D. Hoaglin, F. Mosteller, and J. Tukey. Wiley 1983. Superb introduction to modern “robust” statistics.

Exploring Data Tables, Trends, and Shapes, edited by D. Hoaglin, F. Mosteller, and J. Tukey. Wiley 1985. Advanced topics in modern “robust” statistics.

Practical Statistics for the Physical Sciences, by L. Havilcek and R. Crain. American Chemical Society, 1988. Introduction to statistics for scientific work.

Statistical Analysis for Business and Economics, by D. Harnett and J. Murphy. Third edition, Addison-Wesley, 1985. A thorough textbook.

Basic Statistics, by T. Kurtz. Prentice-Hall, 1963. A clear introductory textbook to mathematical statistics.

Applied Statistics: A Handbook of Techniques, by L. Sachs. Springer-Verlag, 1982. A good, thorough collection of descriptions and techniques.

Nonparametric Statistical Methods, by M. Hollander and D. Wolfe. Wiley, 1973. Methodical and relatively easy to read.

Nonparametric Statistical Inference, by J. Gibbons. McGraw-Hill, 1971. Fairly thorough and rather advanced.

Classical and Modern Regression with Applications, by R. Myers. Duxbury Press, 1986. Fairly thorough book on linear and nonlinear regression. Quite readable.

Getting Started

Let's begin with some examples. First, use the *script* LOADSTAT command from the Command window to bring the compiled versions of FRAMELIB and STATLIB into memory. This is not necessary, but makes your programs start much faster.

Loading will take some time. When that's done, call up STATS from your disk and run it. It prints some statistics about the sizes of files in a directory:

```
N = 45  (none missing)
Mean:          13540.62222        Sum:              609328
Variance:        648307571        Sum sq:      28525533139
SD:            25461.88468        RMS:         28587.57049
Low:                   156        Mean dev:    18341.04000
Median:               1227        Coeff var:       1.88041
High:               100958
Range:              100802
```

Figure 43.01: Output of the STATS program.

NOTE: Numbers shown in this manual may not exactly match the results given on your computer since accuracy of calculations depends on your computer's floating point arithmetic accuracy.

STATS has three important parts. The first is the **library** statement which names STAT1LIB. This lets you use the *Statistics Graphics Toolkit* even if you haven't loaded it before you begin your session.

The second is the *mat read* statement and its associated **data**. This reads the dataset into an array.

The last is the *call PrintStat* statement, which prints statistics about the data in window #0.

Graphing A Histogram

Now call up STATS2 from your disk and run it. It graphs a histogram of the same data:

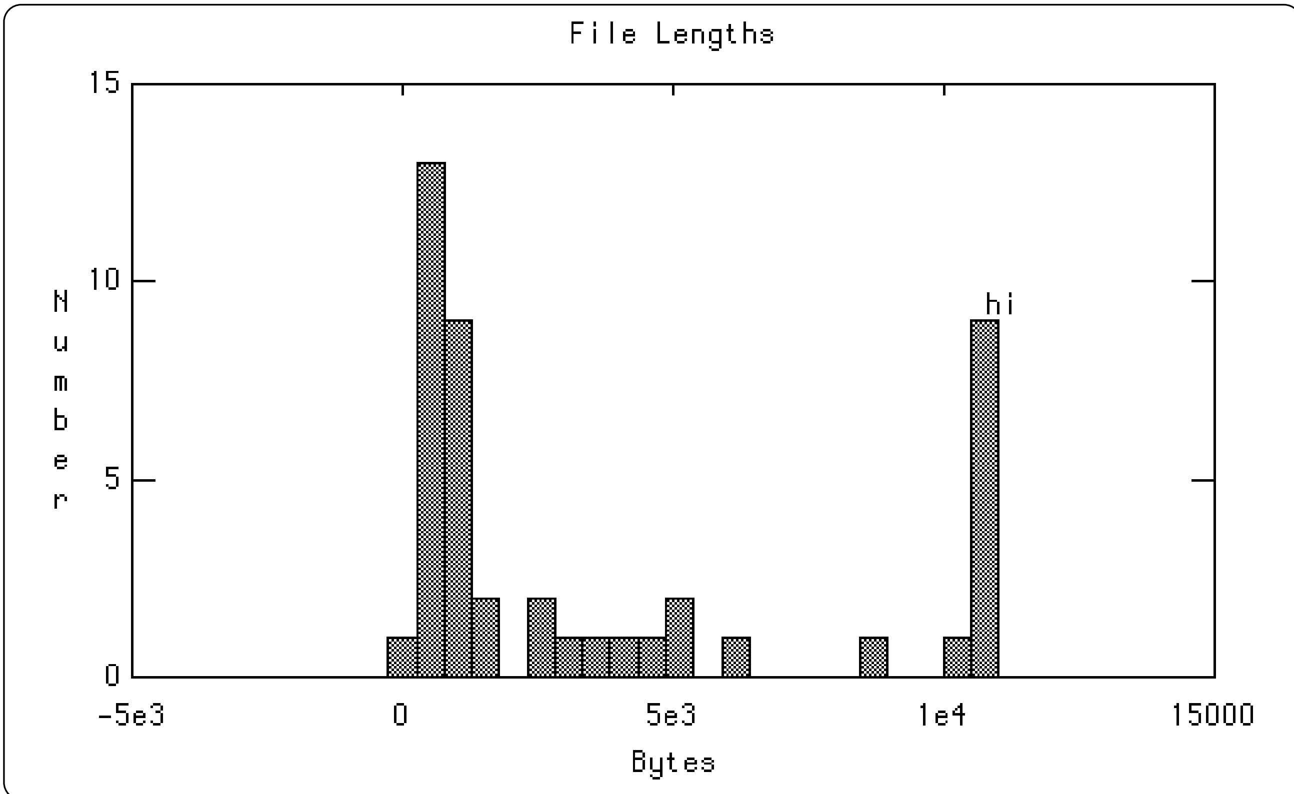

Figure 43.02: Output of the STATS2 program.

NOTE: The illustrations in this manual may not exactly match what appears on your screen. Since the Toolkit automatically adjusts graphs to best fit your computer's screen resolution, it could pick different scales for some graphs. But the illustrations will give you a good idea of what to expect.

STATS2 is like STATS but has three differences: the statements that draw the histogram.

The first is *call SetHistoColor*, which sets the colors of the histogram's bars.

The second is *call SetText*, which defines the graph's title and horizontal and vertical labels.

The third is *call PlotHist*, which draws the histogram with bars running from 0 to 10K at centerpoints that are 512 units apart. The call to PlotHist also contains the color

scheme for this graph: the title color, frame color, and data color.

Looking at the histogram, you can immediately spot several facts that weren't apparent from the printed statistics. For instance, the distribution of file lengths isn't even close to a normal distribution!

Note that many files are bigger than 10K bytes; they are represented by the "hi" bar on the histogram. Such "hi" (and "lo") bars contain all the points outside the chosen range — just so you don't forget that they exist.

Graphing a Scatter Plot with Least-Squares Fit

Now let's turn to graphing data points. Call up the program SCAT from your disk, and run it. Its output looks like this:

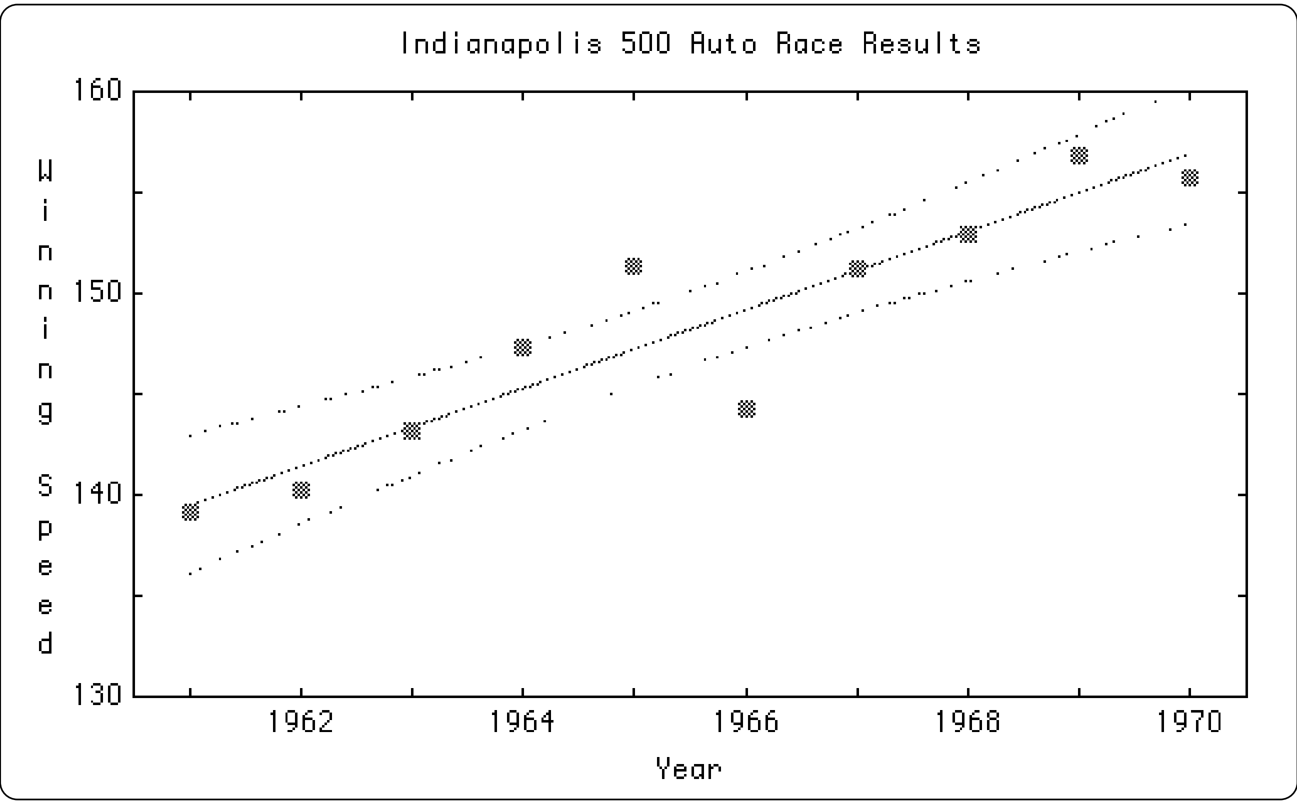

Figure 43.03: Output of the SCAT program.

This program first reads the data points' coordinates into the *x* and *y* arrays.

Then it sets up for the scatter plot. *SetLS* turns on automatic least-squares curve fitting. *SetConfBand* turns on confidence bands for the least-squares line; here the bands are for the 95% confidence interval, drawn in line style 3 (dotted). *SetText* gives the graph's title and labels.

Finally, *PlotScat* draws the scatter plot. Each point is marked in point style 10 (solid block). Line style 0 means that connecting lines between the points are omitted. And finally, PlotScat also gives the color scheme.

Multiple Scatter Plots

It's equally easy to plot several datasets on the same graph. Call up SCATMANY from your disk, and run it.

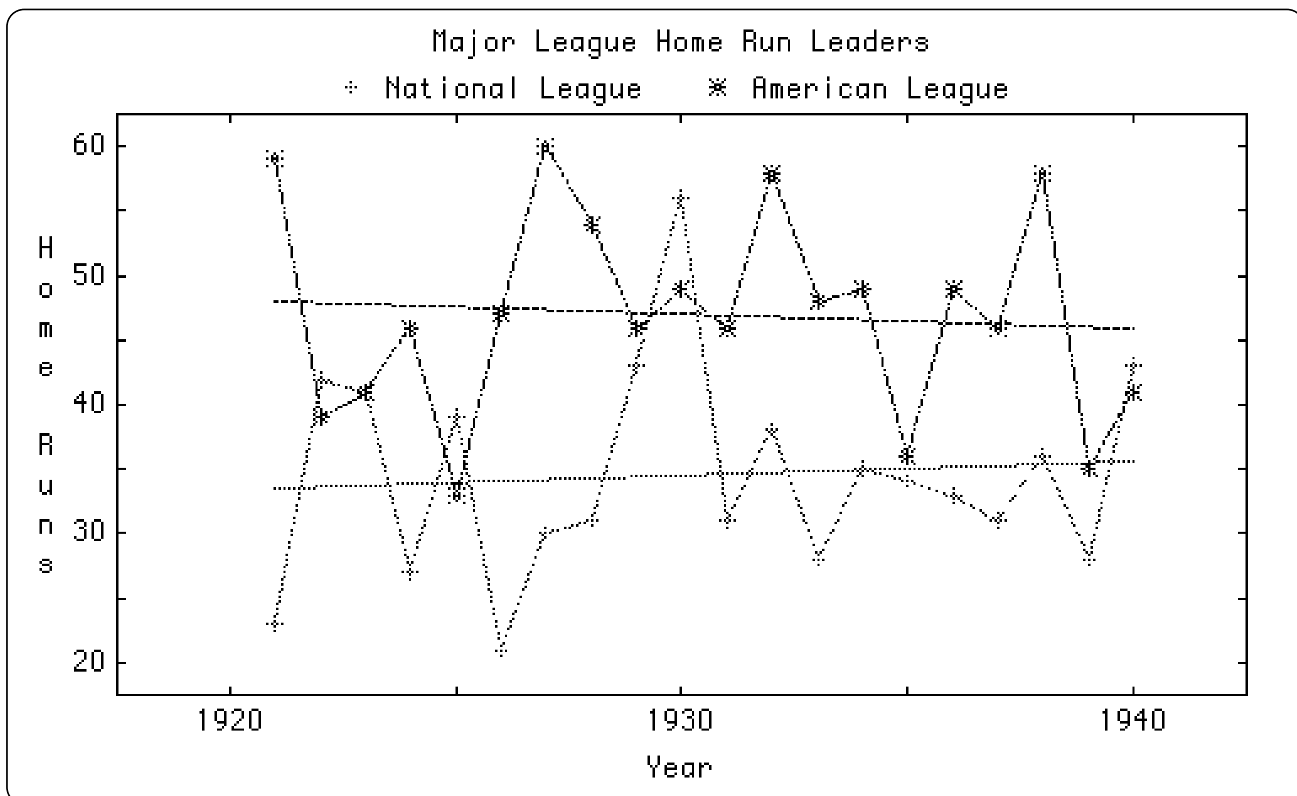

Figure 43.04: Output of the SCATMANY program.

This program plots the number of home runs hit by the leading batters in the National and American baseball leagues between 1920 and 1940. A rather interesting pattern emerges.

SCATMANY is much like SCAT except that the *x*() and *y*() arrays are two dimensional so they can hold information about multiple datasets. In addition, the program defines a *legend$*() array that holds the legends for the two datasets: "National League" and "American League."

PlotManyScat draws all the datasets, picking nice colors and point styles for each set.

As always, you should be wary of blithely fitting least-squares lines to data! Compare

the same data, shown below, with lines fitted by a “resistant” technique that’s less influenced by outlying data points. Now — do you believe either of these linear fits?

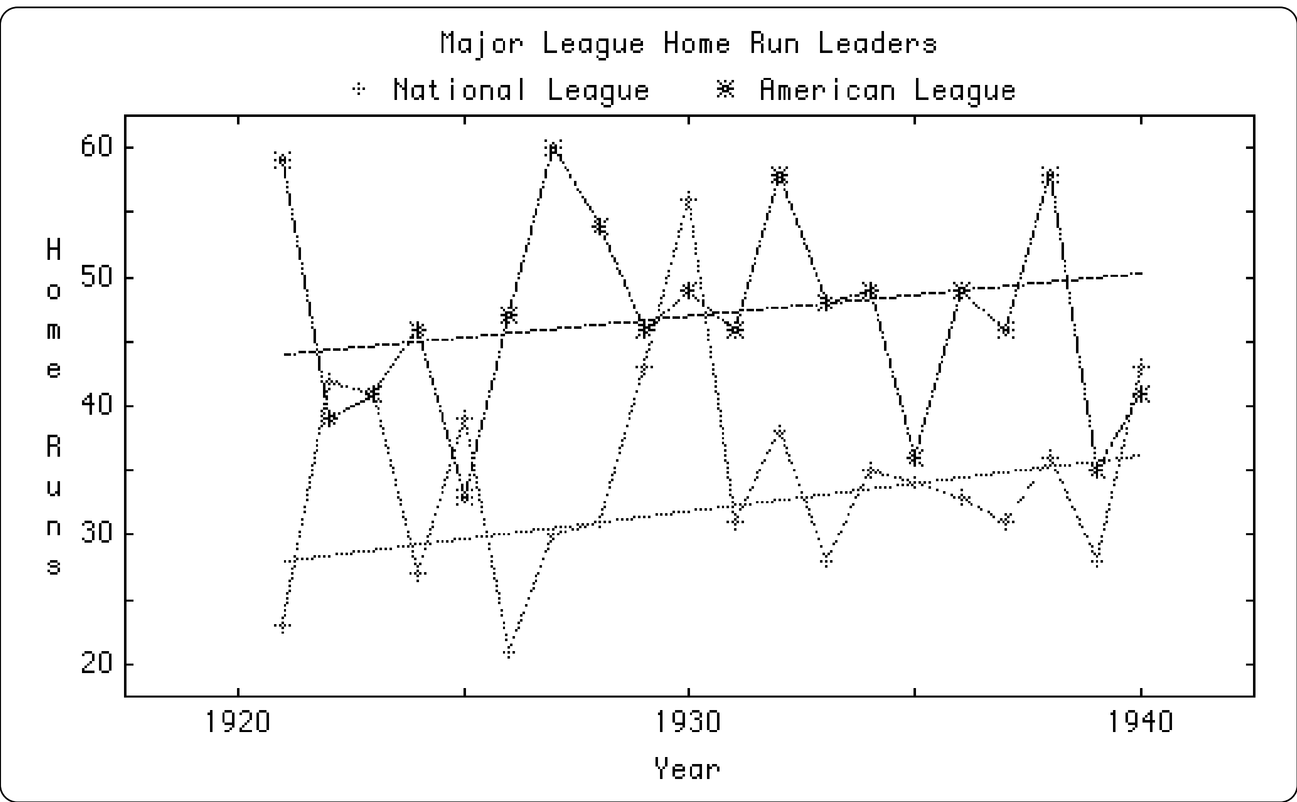

Figure 43.05: Output of the SCATMED program.

Log and Semi-Log Graphs

This Toolkit also makes it very easy to get log and semi-log graphs of your data. For an example, call up SCATLOG and run it.

Figure 43.06 shows the results of SCATLOG. It displays two graphs side-by-side. Both show the same exponential dataset. The left graph uses normal X/Y coordinates; the right uses a log-y scale. It takes just one statement, *call SetGraphType*, to switch to log graphs.

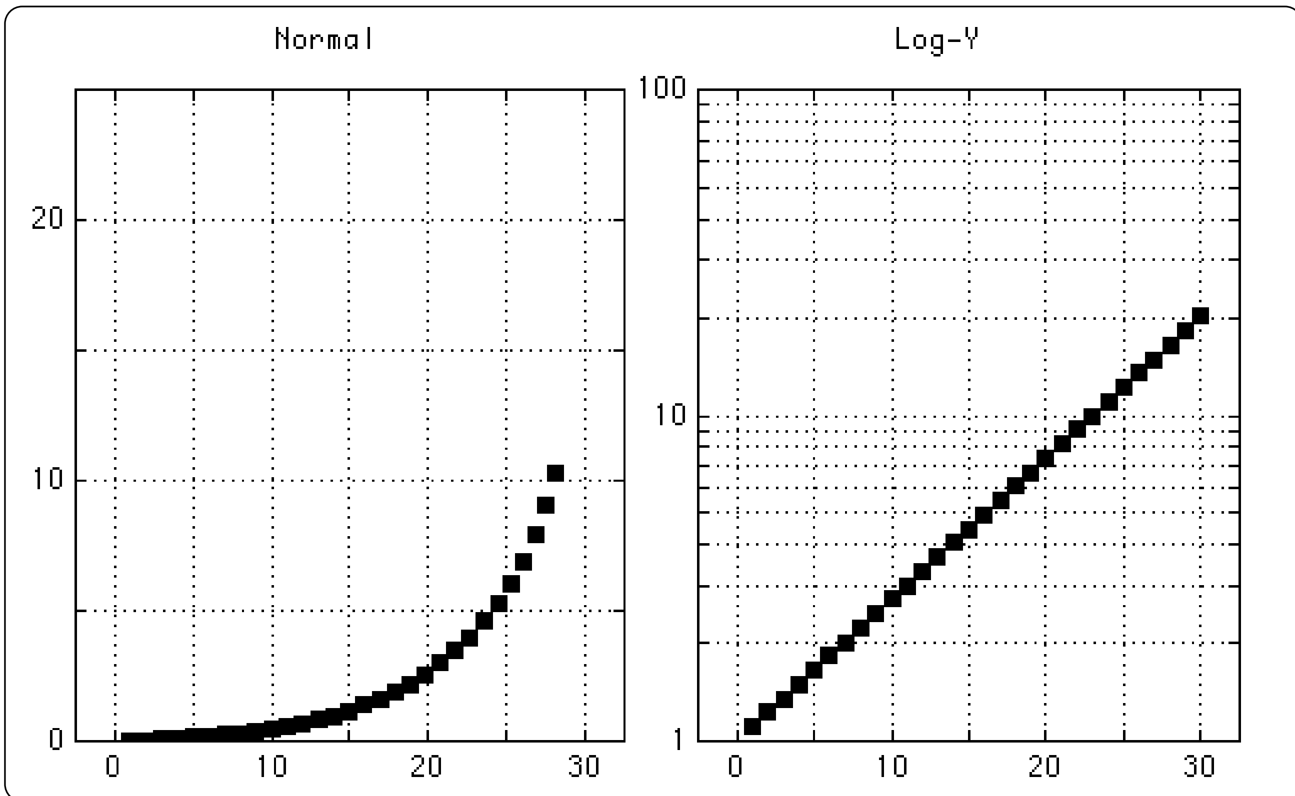

Figure 43.06: Output of the SCATLOG program.

A Complicated Example

Now let's try something more complicated. Call up OBS from your disk and run it. It shows two views of the same data points (car stopping distances):

The left window shows the raw data points, with a polynomial fit superimposed. This polynomial is of degree 2 — a parabola — but any degree can be used. The right window shows a least-squares linear fit applied to the square root of the *y* measure (stopping distances).

OBS has three major parts. First, it reads data into arrays. There are *missing values* in this data, so it reads the data as text, then calls *TextToNum2* to convert the data, with missing values, to numbers.

Second, it opens the left window. *SetDataStyle* tells what point style to use for the raw data points. *SetPolyFit* gives the degree of polynomial to fit. And *PlotObs* actually draws the graph.

Third, the program opens the right window. It turns off polynomial fitting and turns on least-squares linear fitting. Then it calls *SqrTran2* to apply the square root transform to the *y*(,) array. Finally it draws the resulting graph.

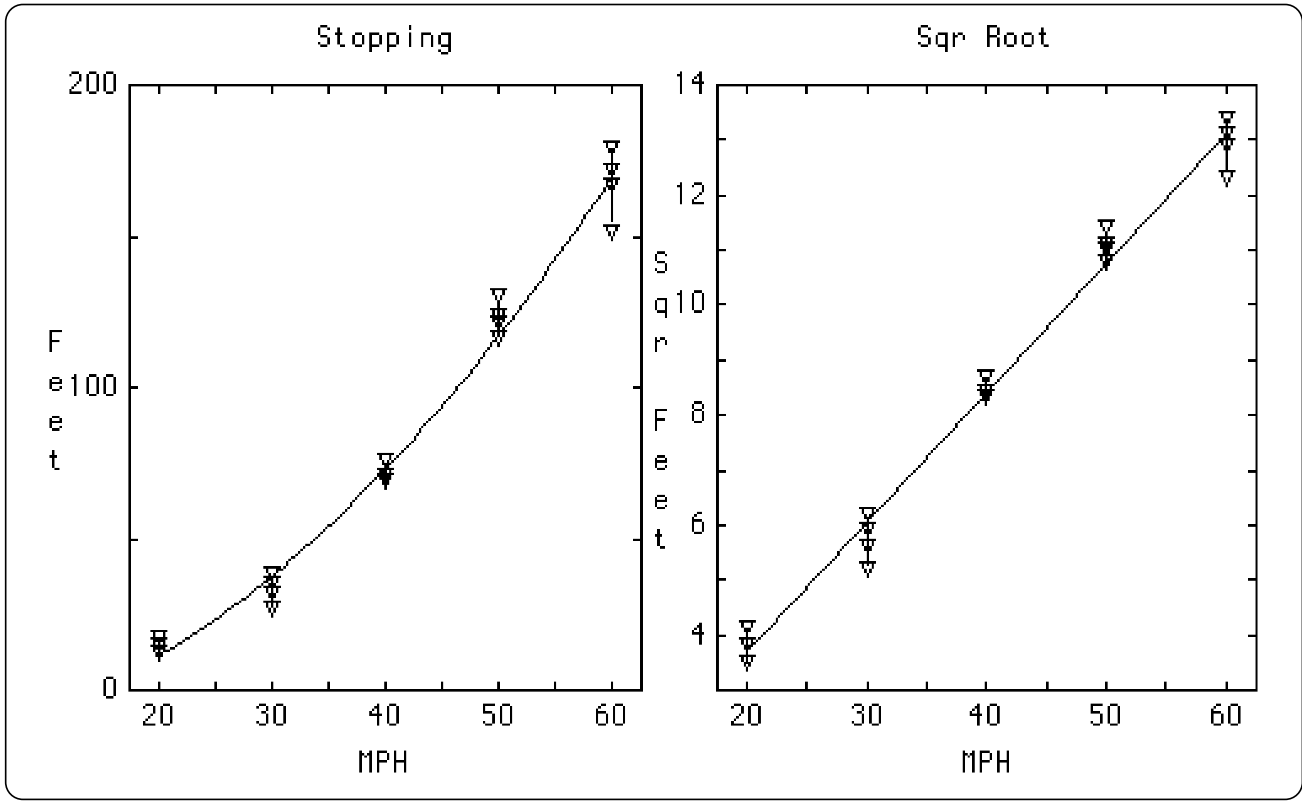

Figure 43.07: Output of the OBS program.

An ANOVA Table

The *Statistics Graphics Toolkit* can also handle ANOVA (analysis of variance) problems. Call up the ANOVA program from your disk and run it.

Source of Variation	Sum of Squares	Degrees of Freedom	Mean Squares (SS/d.f.)
Between samples	59.1762	2	29.5881
Within samples	72.6333	18	4.03519
Total	131.81	20	

F(2,18): 7.33252   Significance probability = .00468

Figure 43.08: Output of the ANOVA program.

This program first *mat reads* three columns of data into a string array *d$*(,); it uses "?" to represent missing data elements, since the columns have different lengths.

Then it converts *d$*(,) to a numeric array *d*(,) by calling *TextToNum2*.

Finally, it uses *PrintAnova* to print the ANOVA table, passing channel #0 for output — the current window.

Multiple Linear Regression

Multiple linear regressions are just as easy. Call up MULTIREG and run it.

```
ANOVA      DF      Sum of Squares      Mean Square
Mean       1        158117.54450
Regression 2         10221.91570         5110.95785
Residual   17         1263.76980           74.33940

F ratio        68.75167         Std. Error          8.62203
Prob(F)      7.12598e-9         Multiple R           .94338
                                R square             .88997
                                Adj R square         .87703

Variable   DF    Estimate        Std. Error          T ratio          Prob(T)

Intercept   1        1.67713
X2          1         .07856          .01880         4.17861           .00063
X3          1        1.79840          .61413         2.92838           .00938

Durbin-Watson D Statistic:        .78355
```

Figure 43.09: Output of the MULTIREG program.

NOTE: If you are using a small Mac screen and find that the right portion of this output does not show on your screen, switch to the small Output font.

It has only two important statements. It *mat reads* data into an array, then calls *PrintMultiRegress*.

"Robust" Statistics

The *Statistics Graphics Toolkit* contains a particularly large collection of nonparametric and other "robust" statistical tools. Nonparametric techniques have been in common usage for the past forty years; the newer robust techniques — such as "box plots" — are just now coming into general use.

For a quick example, call up FANCYBOX from your disk and run it. It displays box plots of the ten largest cities in each of 16 countries using a semi-logarithmic scale.

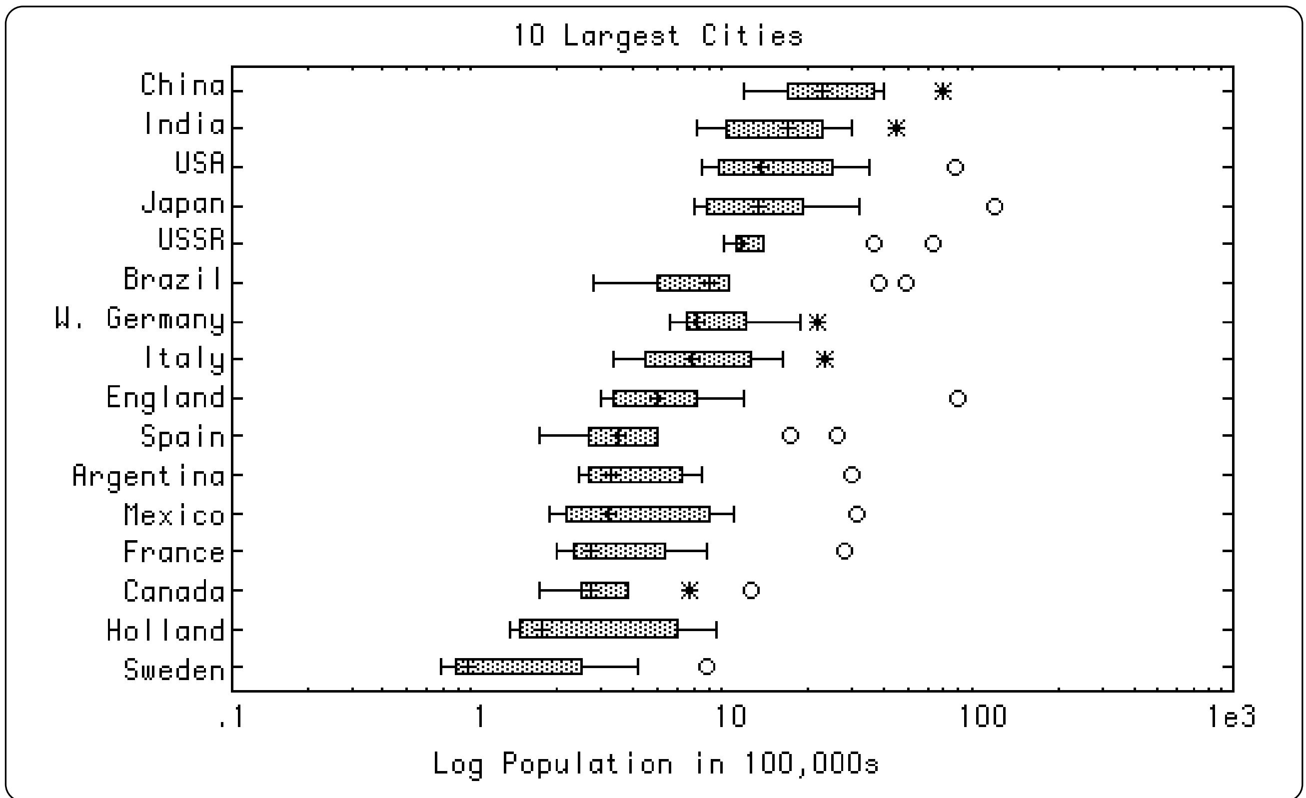

Figure 43.10: Output of the FANCYBOX program.

The Grand Tour

The *Statistics Graphics Toolkit* can draw many more kinds of graphs than we've shown here. For a grand tour, call up STATDEMO from your disk and run it. It's an endless loop, so stop the program when you've seen enough.

Data Management

This section describes routines that make it easy to manipulate rows and columns of matrices, sort a vector, read and print the “missing” value, and import data from other programs — for example, numbers that contain commas or dollar signs.

Missing Values

By default, the *Statistic Graphics Toolkit* sets aside one number to be the “missing value” so you can use datasets that include missing points. If you want, you can change which number is used for the missing value, or tell the Toolkit to disallow missing values entirely.

The missing value is preset to –9.9e99 and its text representation is preset to “?”.

SetHaveMissing (f)

If $f = 0$, disallow missing values. That is, there will be no special number or string that stands for a “missing” data item. Otherwise, there will be a missing value; see SetMissing and SetMissingText to control the numeric and string forms of the missing value.

AskHaveMissing (f)

Return 1 if missing values are accepted, 0 otherwise.

SetMissing (n)

Use n as the missing value. Data items with this value will be ignored whenever statistics are being computed; it’s as if they don’t exist in the dataset. You can use any number n as the missing value.

AskMissing (n)

Return the current missing value in n.

SetMissingText (s$)

Change the missing text value. By default, the missing text is “?”; that is, missing values are converted to question marks by StatStr$, and question marks stand for the missing value in StatVal.

AskMissingText (s$)

Return the current value of the missing text.

Conversion Between Strings and Numbers

This section describes routines that convert between strings and numbers. They correctly convert missing values as well as standard numbers.

In addition, they ignore dollar signs, commas, and asterisks when converting strings to numbers, to make it easier to import data.

def StatVal (n$)

Like True BASIC's Val function, *StatVal* converts a string to a number. However, it ignores spaces, dollar signs, commas, and asterisks. If the "missing" text is a null string, it will convert *any* illegal string into the missing value. But if the missing text is not null, it will convert only that string into the missing value; all other illegal strings will get an error message.

For example, StatVal("$3,230") = 3230. If the missing text is the null string, then StatVal("hi") gives the missing value. If the missing value is "?", then StatVal("?") gives the missing value but StatVal("hi") gives an error.

Exceptions:

735 Can't convert to number or missing value: *xxx*

def StatStr$ (n)

Like True BASIC's Str$ function, *StatStr$* converts a number to a string. However, if the number is the missing value, *StatStr$* produces the "missing" text instead as *TextToNum*.

TextToNum (n$(), n())

TextToNum converts a string array *n$*() to the corresponding numeric array *n*() by using *StatVal* on each element. The two arrays needn't have the same bounds; the target array's lower bound is left untouched and its upper bound adjusted.

Exceptions:

735 Can't convert to number or missing value: xxx

NumToText (n(), n$())

NumToText converts a numeric array *n*() to the corresponding string array *n$*() by using *StatStr$* on each element. The two arrays needn't have the same bounds; the target array's lower bound is left untouched and its upper bound adjusted.

TextToNum2 (n$(,), n(,))

TextToNum converts a string array *n$*(,) to the corresponding numeric array *n*(,) by using *StatVal* on each element. The two arrays needn't have the same bounds; the target array's lower bounds are left untouched and its upper bounds adjusted.

NumToText2 (n(,), n$(,))

NumToText converts a numeric array *n*(,) to the corresponding string array *n$*(,) by using *StatStr$* on each element. The two arrays needn't have the same bounds; the target array's lower bounds are left untouched and its upper bounds adjusted.

GetStatFile (#n, n$(,))

GetStatFile reads a text file full of statistics data into an array. The file #*n* must be opened before you call *GetStatFile* by a statement such as OPEN #1: NAME "datafile". Each row and column of the text file data goes into the appropriate item of *n$*(,).

Each line of the text file is treated as a row. Blank lines are ignored. Columns are separated by either a tab character, or two or more spaces. Lines in the file may have different numbers of columns; short rows are "left justified" in the *n$*(,) array. Thus if the first line in a file contains three columns but some line within the file contains only one data item, the last two *n$*(,) entries for that line are the null string. For example:

	File			***n$*(,)**	
Name	Age	Weight	"Name"	"Age"	"Weight"
Allen	34	185	"Allen	"34"	"185"
Byron		170	"Byron"	"170"	""
Darcy			"Darcy"	""	""

These rules let you read SPSS files, Excel files, Lotus 1-2-3 files, and text files created by most common application programs.

Note that all data items are taken as strings; thus you can use files that include column headers, extraneous information, etc. To convert *n$*(,) to a numeric array, changing all nondata items to missing values, try:

```
call SetHaveMissing(1)
call SetMissingText("")
call TextToNum2(n$(,), n(,))
```

See the program GETFILE on your disk for an example.

Exceptions:

7004	Channel isn't open.
8011	Reading past end of file.
8501	Must be text file.

GetRow (a(,), r, v())

Get a row *r* of the 2-D array $a(,)$ into the vector v. This redims v to have Lbound = 1.

GetCol (a(,), c, v())

Get a column *c* of the 2-D array $a(,)$ into the vector v. This redims v to have Lbound = 1.

AppendArray (a(), b())

Concatenation for vectors: $a() = a()$ & $b()$. In other words, the elements of $b()$ are added to the end of $a()$.

AppendRow (a(,), r())

Add a new row $r()$ at the bottom of a 2-D array $a(,)$. The row sizes must match unless the 2-D array has no elements, in which case it will simply be changed to match the new row.

Exceptions:

742	New row doesn't match array's shape in AppendRow.

AppendCol (a(,), c())

Add a new column $c()$ at the right side of a 2-D array $a(,)$. The column sizes must match unless the 2-D array is empty, in which case it will simply be changed to match the new column.

Exceptions:

743	New column doesn't match array's shape in AppendCol.

SortAndCount (d(), s(), nm, nnm)

Sort $d()$, placing the sorted results into $s()$. Missing items will be removed, so $s()$ may be smaller than $d()$. The $d()$ array will not be changed. The number of missing items will be in nm, and non-missing items — i.e., Size(s) — will be in nnm.

Making Graphs

The *Statistics Graphics Toolkit* makes it easy to draw many kinds of graphs. This section describes what all these graphs have in common.

The Frame. A graph's frame includes a rectangular box around the canvas, a title, a horizontal and a vertical label, some marks along the edge of the box, and ticks which join the marks to the box. Complicated graphs can also include legends which identify different datasets. Legends usually go in the frame, just below the title, but you can also put them elsewhere.

The Canvas. The canvas lies inside the frame. It's where the data is plotted.

A Color Scheme. Your color scheme tells how the title, frame, and data display should be colored.

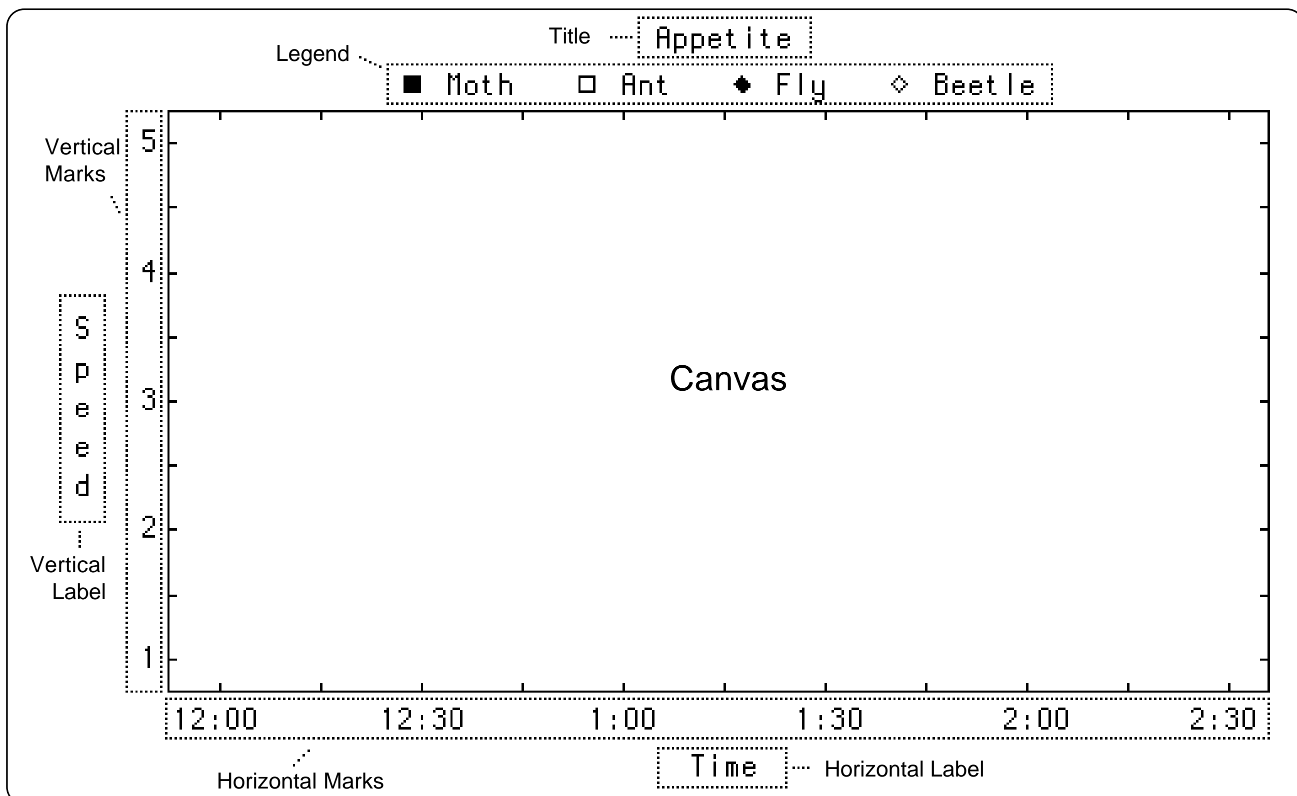

Figure 43.11: Parts of a Statistics Graph.

Titles and Labels

Before you draw a graph, you should specify its title and horizontal and vertical labels by calling *SetText*.

SetText (title$, hlab$, vlab$)

SetText sets the current text for the title, horizontal label, and vertical label. If any of these texts are null strings, that space is added to room for the data display.

AskText (title$, hlab$, vlab$)

AskText returns the current texts of the title, horizontal label, and vertical label. It returns the null string for those labels which don't yet have values.

Color Schemes

Whenever you draw a graph, you must give a color scheme. It tells what colors to use for the graph's title, frame, and data.

Usually a color scheme gives at least three colors. The first is the title's color. The second is the frame's color. (The frame includes the horizontal and vertical labels.) And the remaining colors are used, in sequence over and over, for the data values.

For example, a color scheme "white blue red yellow green" gives a white title, blue frame, and data colors which cycle from red to yellow to green and back to red again.

If you give fewer than three colors, the last color is used to fill out the remaining colors. So the color scheme "red green" gives a red title, green frame, and green data. The color scheme "white" draws everything in white. The null color scheme "" simply uses the current foreground color for everything.

What Color Names Can You Use?

You can use any True BASIC color in the color scheme. In particular, you might want to use "background" if you've used *SetCanvas* to change the graph's canvas color. Then you can use the real background color to get a reverse-video effect.

You can also use colors such as "1" or "13" to access any of your computer's colors, even if they lack True BASIC names. For example, the color scheme "white 1 12" gives a white title, frame in color #1, and data in color #12.

Canvas Colors

The *SetCanvas* routine controls the canvas color. For example, call SetCanvas("red") gives you a red canvas. As in color schemes, you can use numbers like "13" as well as color names. Once you've set the canvas color, all graphs will use this color until your program stops or you call *SetCanvas* again. For more information, see the "Advanced Graph Control" section.

Your Computer's Modes

Most computers have several graphics modes. You can use the *Statistics Graphics Toolkit* in any graphics mode. If you don't pick a mode, the Toolkit will switch to mode "GRAPHICS" before it begins to draw. (Exception: it will use an EGA mode on IBM PCs and PS/2s with EGA cards.)

To control the mode, use True BASIC's *set mode* statement to switch to that mode before you call any Toolkit routine. For example, use the *set mode "medres"* statement on an IBM PC with EGA adaptor to draw in the medium resolution mode.

Many graphs cannot be drawn in low or medium resolution. The textual labels don't fit in the space allowed. Therefore, you may wish to switch to a high resolution before drawing a graph. On an IBM PC with a Color Graphics Adaptor, for instance, you may wish to *set mode "hires"* before you call any Toolkit routines.

Point and Line Styles

Point styles are used for graphing data points. The line styles can be used to connect data points, add confidence bands, or show fitted curves. You can also use GraphPoint and GraphLine to draw points and lines wherever you wish. They're described in the "Advanced Graph Control" section.

Available Point Styles

Point styles are numbered from 1 to 13. You may also use style 0, which means that no point should be drawn. The visible styles are:

Style	Appearance	Style	Appearance
1	dot	8	down triangle
2	plus	9	diamond
3	asterisk	10	solid box
4	circle	11	solid up triangle
5	X	12	solid down triangle
6	box	13	solid diamond
7	up triangle		

Available Line Styles

Line styles are numbered from 1 to 4. You may also use style 0, which means that no line should be drawn. The visible styles are:

Style	Appearance
1	solid
2	dashed
3	dotted
4	dash-dotted

AskMaxPointStyle (n)

AskMaxPointStyle returns the maximum number of point styles currently supported by the *Statistics Graphics Toolkit*. At present, this number is 13. See the previous sections in this section for more information.

If more point styles are added to later versions of the Toolkit, this routine will return a larger number.

AskMaxLineStyle (n)

AskMaxLineStyle returns the maximum number of line styles currently supported by the *Statistics Graphics Toolkit*. At present, this number is 4. See the previous sections in this section for more information.

If more line styles are added to later versions of the Toolkit, this routine will return a larger number.

Printing A Graph

Every computer has a different way to print its screen on a printer. Usually you must press some combination of keys on the keyboard, and the computer then takes a snapshot of the screen onto the printer. See your *True BASIC User's Guide* to find out how to do this on your computer.

You can also take a snapshot from within a program. Just add **library "prtlib"** to your program. Then *call PrtSc* whenever you want to print the screen image.

NOTE: Many computers require some kind of special preparation before you can print a picture on your printer.

Often you must run some utility program before you start running True BASIC. (On the IBM PC or compatibles, *Call PrtSc* simply mimics the Shift-PrtSc keystroke combination, and will not work without a driver installed.) Read your computer's *Owner's Manual* to find out what you must do to capture a screen on your operating system.

Simple Statistics

This section describes how to find simple descriptive statistics about a set of numbers: the mean, median, root mean square, and so forth.

Stats (data(), st())

Analyze a dataset *data*() and return the array *st*() full of statistics. You can use the following functions to index this *st*() array; see the "Subscript Functions" section for more help with using these functions. If there are no non-missing elements in *data*(), Stats gives an error message.

st_n	number of elements
st_nm	number of missing elements
st_nnm	number of non-missing elements
st_sum	sum of items
st_mean	mean (average)
st_ssq	sum of squares
st_var	variance
st_sd	standard deviation
st_sem	standard error of the mean
st_med	median
st_low	lowest value
st_hi	highest value
st_range	range
st_rms	root mean square (quadratic mean)
st_md	mean absolute deviation
st_cvar	coefficient of variance
st_wmean	Winsorized mean

By default, the variance, standard deviation, and coefficient of variance are computed with denominator equal to the number of non-missing items minus 1, *nnm*–1 . To use *nnm* instead, *call SetSD(0)* before calling *Stats*.

GroupedStats (freq(,), st())

GroupedStats analyzes a frequency dataset *freq*(,) and returns the same *st*() array as *Stats*. Frequency datasets give rougher values than raw datasets since information was lost in grouping the data. Winsorized means aren't defined for grouped data, so the

simple mean is returned in its place. Also, the extreme ends of the appropriate intervals — not the centerpoints — are returned as the lowest and highest values.

By default, grouped statistics are computed without Sheppard's correction. If you wish to use the correction, call *SetSheppard(1)* before calling *GroupedStats*. As with *Stats*, call *SetSD(0)* before calling *GroupedStats* if you want to compute variance, etc., with denominator *n* instead of *n*–1.

MeanSD (data(), mean, sd)

MeanSD returns the *mean* and standard deviation *sd* of a dataset *data*(). You can get the same information from Stats but this is often handier. *Call SetSD(0)* to compute the standard deviation based on denominator *n* instead of *n*–1.

GeoMeanSD (data(), mean, sd)

GeoMeanSD returns the geometric *mean* and standard deviation *sd* of a dataset *data*(). These values are only defined for datasets containing strictly positive numbers. *Call SetSD(0)* to compute the standard deviation based on denominator *n* instead of *n*–1.

HarMean (data(), mean)

HarMean returns the harmonic *mean* of a dataset *data*(). Note that harmonic means are defined only for datasets containing strictly positive numbers.

Modes (data(), n, modes(), count)

Modes returns all the modes of the dataset *data*() in *modes*(), and *n* equal to the number of modes. If there are no modes, *n* = 0.

Otherwise *n* = Size(*modes*) and each element of *modes*() is a mode of the dataset. *Count* tells the number of occurences of the modal value(s).

GroupedModes (freq(,), n, modes(), count)

GroupedModes returns all the modes of the frequency dataset *freq*(,) in *modes*(), and *n* equal to the number of modes. If there are no modes, *n* = 0.

Otherwise *n* = Size(*modes*) and each element of *modes*() is a mode of the frequency dataset, i.e, the centerpoint of an interval that has a high count. *Count* gives the frequency value of the modal centerpoint(s).

PrintStats (#n, data())

Print a table of the *Stats* information about a dataset to channel #*n*. Pass #0 to print in the current window.

The first line displays N = *nnm*, the number of non-missing elements, with the number of missing elements added afterwards in parentheses.

```
N = 45  (none missing)
Mean:          13540.62222          Sum:                609328
Variance:        648307571          Sum sq:        28525533139
SD:            25461.88468          RMS:           28587.57049
Low:                   156          Mean dev:      18341.04000
Median:               1227          Coeff var:         1.88041
High:               100958
Range:              100802
```

Output of the STATS program.

PrintLongStats (#n, data())

Print a table of the *Stats* information about a dataset, plus its modes, skew, and kurtosis, to channel #*n*. Pass #0 to print in the current window. This routine uses the moment definitions of skew and kurtosis, and is significantly slower than PrintStats since they take some time to compute.

```
N = 100  (none missing)
Mean:                .16814          Sum:              16.81420
Variance:           1.21098          Sum sq:          119.88732
SD:                 1.10045          RMS:               1.10777
Low:               -2.27722          Mean dev:           .84312
Median:              .10761          Coeff var:         6.54475
High:               3.37939          Skew:               .40716
Range:              5.65661          Kurtosis:          3.17436
Modes:       (None)
```

Output of the LONGSTAT program.

PrintGroupedStats (#n, freq(,))

Print a table containing the *GroupedStats* information about a frequency dataset, plus a list of its modes, to channel #*n*. Pass #0 to print in the current window.

LetterValues (data(), sorted(), lv())

Return the letter values of the dataset *data*() in *lv*() and the samples sorted into increasing order in *sorted*(). You can use the following functions to index this *lv*() array; see the "Subscript Functions" section.

See the BoxPlot routine below for a graphical picture of the letter values. The arrays can all have different lower bounds; the lower bounds for *sorted*() and *lv*() are kept intact, and their upper bounds adjusted.

lv_nnm	number of non-missing elements
lv_med	median
lv_lhin	left hinge (roughly 1st quartile)
lv_rhin	right hinge
lv_leig	left eighth
lv_reig	right eighth
lv_linf	left inner fence
lv_rinf	right inner fence
lv_louf	left outer fence
lv_rouf	right outer fence
lv_lout	number of left outliers (to left of left extreme)
lv_rout	number of right outliers (to right of right extreme)
lv_lext	leftmost value inside left inner fence
lv_rext	rightmost value inside right inner fence
lv_lmax	smallest value
lv_rmax	largest value
lv_tri	Tukey's trimean

The letter values are computed by first sorting the raw data. Definitions of the letter values are: **Median**, as usual. **Hinges**, values whose positions are Int(*median position* + 1) / 2. **Eighths**, values whose positions are Int(*hinge position* + 1) / 2. **Hinge spread**, difference between left and right hinges. **Inner fences**, 1.5 hinge-spreads out from the hinges. **Outer fences**, 1.5 hinge-spreads out from inner fences. **Outliers**, values outside the inner fences. **Extremes**, the most extreme values that are not outliers. **Maxima**, outermost values. **Trimean**, (left hinge + 2 * median + right hinge) / 4.

BoxPlot (data(), col$)

Plot a "box-whisker" plot using the letter values of the dataset *data*(). The color scheme *col$* has the usual meaning; for instance, "red green blue" gives a red title, green frame, and blue box plot.

A "+" marks the median. The box stretches from left hinge to right hinge. Whiskers extend to the extremes. Outliers inside the outer fences are plotted as stars "*" those outside as circles "o". The vertical scale is meaningless. *Call SetDataStyle* to add the data points.

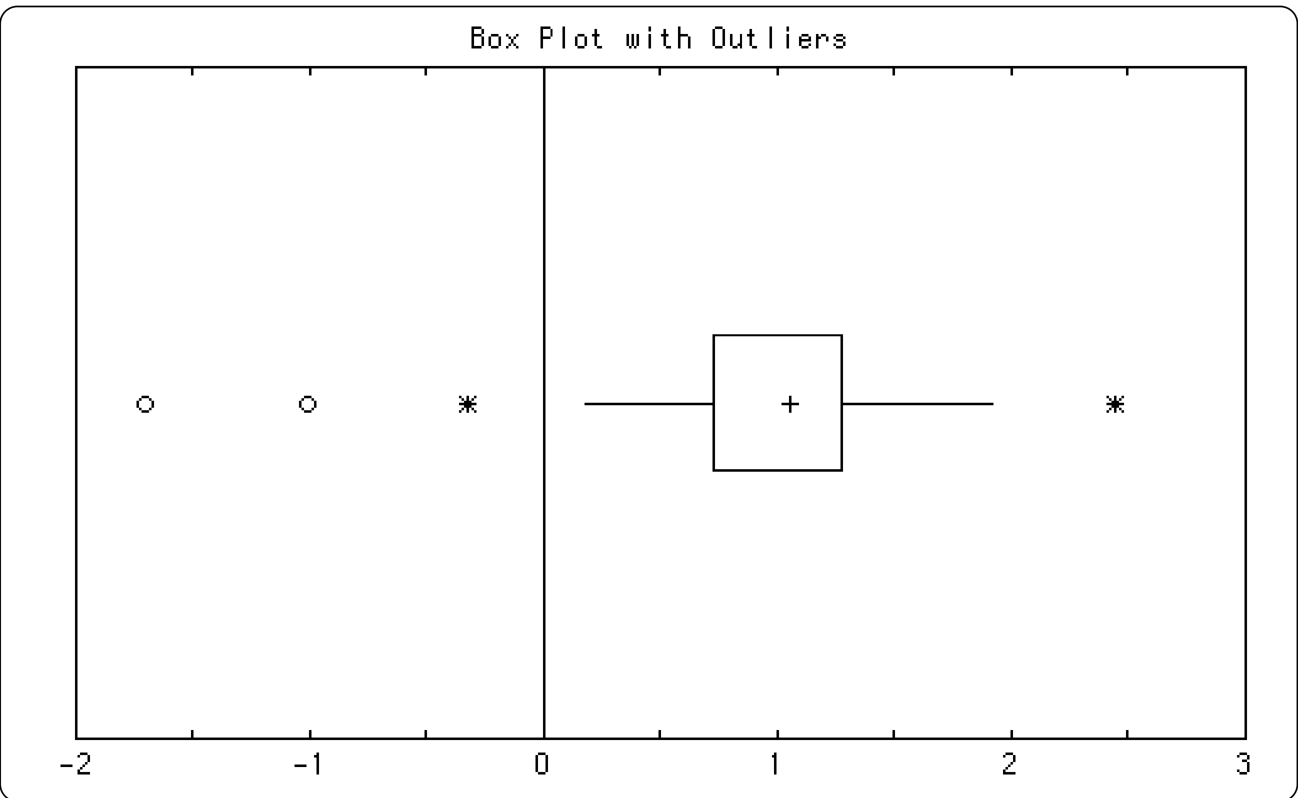

Figure 43.13: Output of the BOXPLOT program.

ManyBoxPlot (data(,), names$(), col$)

Plot stacked "box-whisker" plots using letter values of the datasets *data*(,). Each column of *data*(,) is taken as an independent dataset, and gets its own box plot. The plot is scaled so all data points are in the graph. Datasets need not have the same numbers of elements; just pad short sets with missing values. Each dataset's name is displayed beside its plot. Pass names in the *names$*() array. Size(*names$*) must equal Size(*data*,1).

The color scheme *col$* has the usual meaning; for instance, "red green blue" gives a red title, green frame, and blue box plot. If you give multiple data colors, the boxes are

drawn in that sequence of colors.

A "+" marks the median. The box stretches from left to right hinge. Whiskers extend to the extremes. Outliers inside the outer fences are plotted as stars "*"; those outside as circles "o". By default, data points are not shown; call *SetDataStyle* to add the data points.

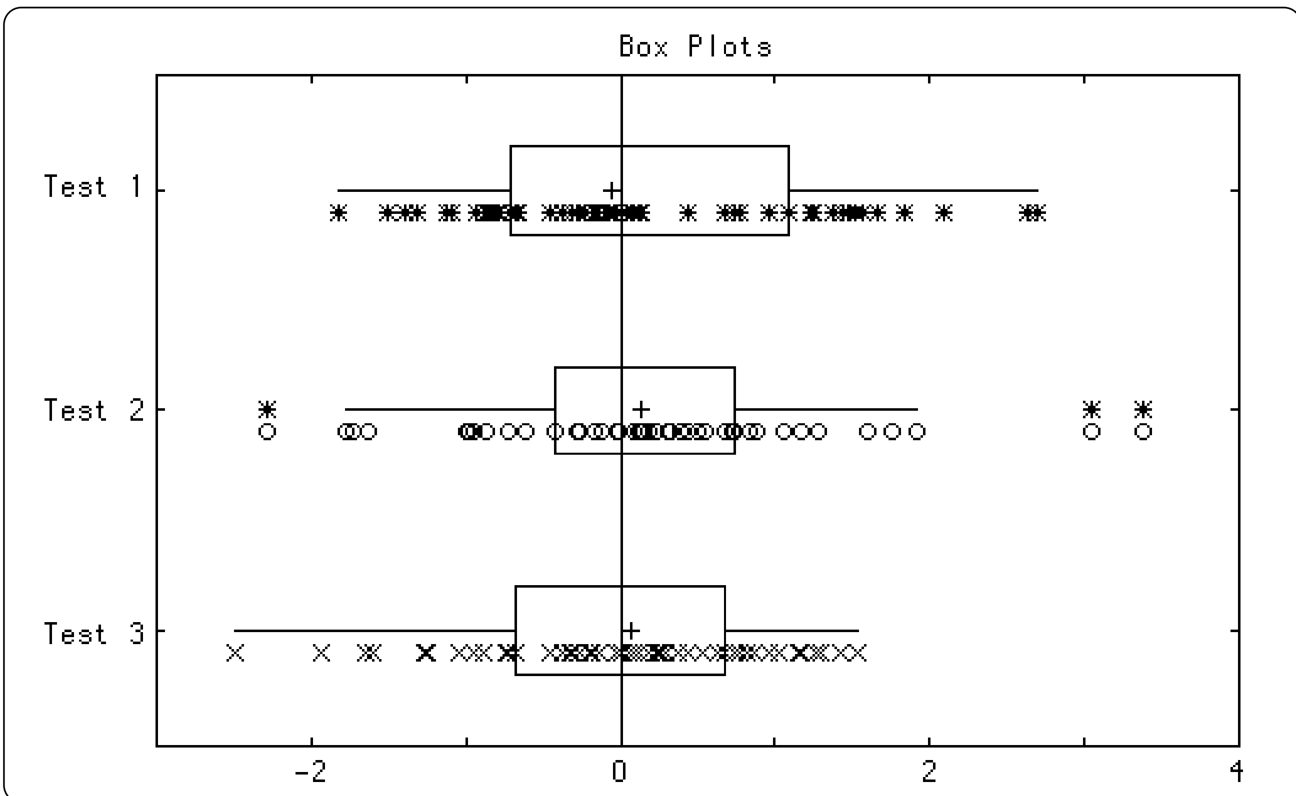

Figure 43.14: Output of the MANYBOX program.

Advanced Options on Box Plots

By default, this Toolkit draws horizontal box plots. To get vertical boxes, just call *SetBoxVert(1)* before drawing the box plot.

You can add "crossbars" to the whiskers by calling *SetErrorBeam(p)* where *p* is the number of pixels wide each side of the bar should be.

See BOXVERT, on your diskette, for an example of a vertical plot with crossbars on the whiskers.

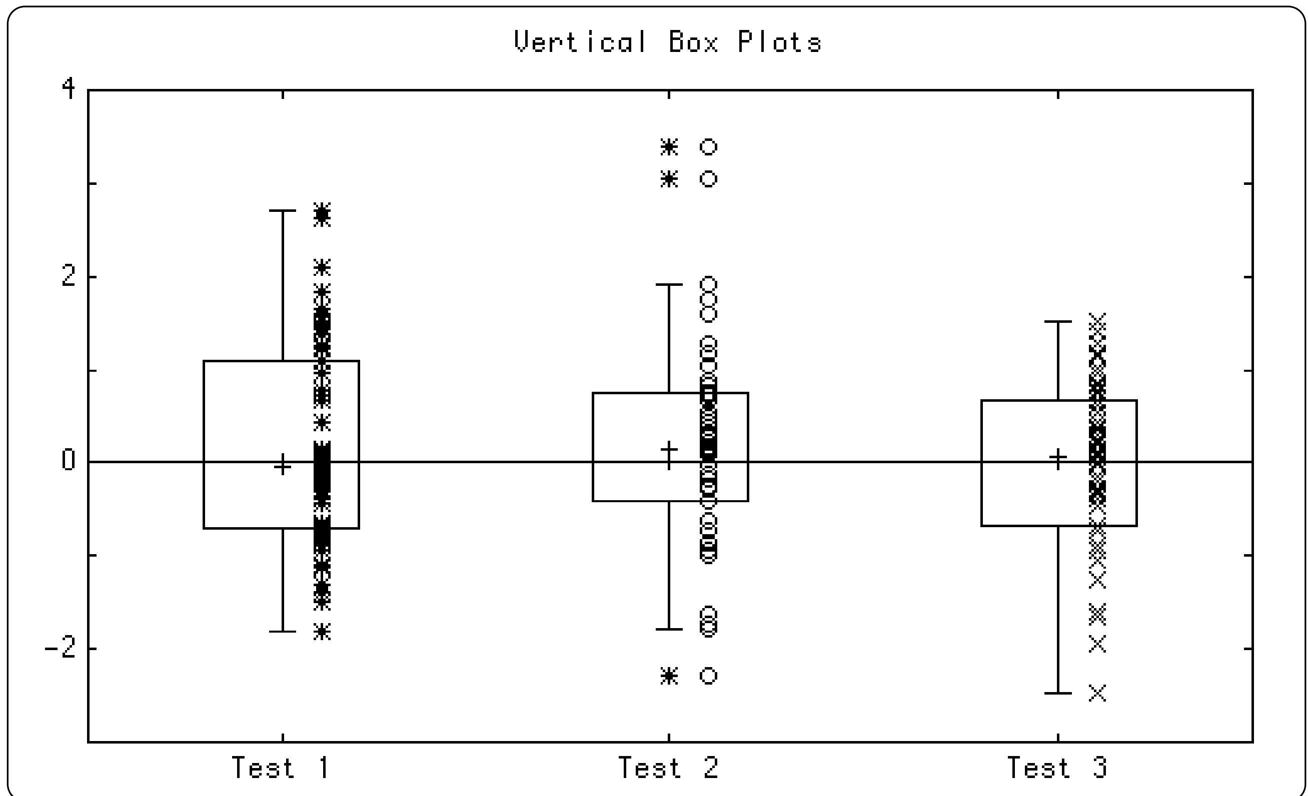

Figure 43.15: Output of the BOXVERT program.

You can also use *SetGraphType* to get a semi-logarithmic scale, and *SetBoxColor* to shade the box. See FANCYBOX, on your diskette, for an example of both options.

SetBoxColor (col$)

Change the color used for subsequent boxes. The new color *col$* can be a True BASIC color name such as "red" or a color number such as "1". It will be used for the insides of boxes; the outlines and whiskers are controlled by the graph's color scheme. Pass "" to get uncolored boxes (the default).

AskBoxColor (col$)

The opposite of *SetBoxColor*. It returns *col$* = the color used for subsequent box colors.

Skew, Kurtosis, and Moment Routines

Skew and kurtosis have various definitions: moment, Pearson's first or second, quartile, and percentile. This Toolkit includes routines for all, plus routines to find "moments" about a dataset's mean or an arbitrary origin.

SkewM (data(), skew)

Moment coefficient of skew: $m_3 / (sd\verb|^|3)$ where m_3 is the third moment about the mean. See MomentAbout below for a definition of moments about a point.

Exceptions:

713 Can't use SkewM with SD = 0.

Skew1 (data(), skew)

Pearson's first coefficient of skew: $(mean - mode) / sd$. Note that this is chancy to use; the mode often doesn't exist or isn't unique.

Exceptions:

708 Can't use Skew1 with SD = 0.
709 Skew1 needs data with exactly one mode.

Skew2 (data(), skew)

Pearson's second coefficient of skew. This is defined as $(mean - median) / sd$.

Exceptions:

710 Can't use Skew2 with SD = 0.

SkewQ (data(), skew)

Quartile coefficient of skew: $(Q_3 - 2*Q_2 + Q_1) / (Q_3 - Q_1)$.

Exceptions:

711 Quartiles 1 and 3 equal in SkewQ.

SkewP (data(), skew)

Percentile coefficient of skew: $(P_{90} - 2*P_{50} + P_{10}) / (P_{90} - P_{10})$.

Exceptions:

712 Percentiles 10 and 90 equal in SkewP.
761 Need at least 99 data items for Percentiles.

KurtosisM (data(), k)

Moment coefficient of kurtosis: $m_4 / (sd\verb|^|4)$ where m_4 is the fourth moment about the mean. See MomentAbout below for a definition of moments about a point.

Exceptions:

714 Can't use KurtosisM with SD = 0.

KurtosisP (data(), k)

Percentile coefficient of kurtosis: $.5 * (Q_3 - Q_1) / (P_{90} - P_{10})$.

Exceptions:

715 Can't use KurtosisP with percentiles 10 and 90 equal.
761 Need at least 99 data items for Percentiles.

Moment (data(), r, mr)

Given r, and dataset $data()$, this computes the rth moment mr about the dataset's mean.

Exceptions:

716 Moment 'r' must be a positive integer: r

MomentAbout (o, data(), r, mr)

Given r, an arbitrary origin o and dataset $data()$, compute the rth moment mr about o as shown by the formula where n = Size($data$).

$$\frac{\sum_{i=1}^{n} (data(i) - o)^r}{n}$$

Exceptions:

716 Moment 'r' must be a positive integer: r

Quartiles and Percentiles

These routines return the quartiles and percentiles of a dataset.

Quartiles (data(), q1, q2, q3)

Quartiles of a dataset $data()$, returning the first quartile in $q1$, second (median) in $q2$, and third in $q3$. Your dataset must contain at least 3 non-missing items. Quartile values will be interpolated when the quartile falls between two data items.

Exceptions:

762 Need at least 3 data items for Quartiles.

Percentiles (data(), p())

Percentiles of a dataset $data()$ in the array $r()$ which is automatically redimensioned to $p(1{:}99)$. Thus, for example, $p(95)$ will contain the 95% percentile value. Your dataset

must contain at least 99 non-missing items. Percentile values will be interpolated when the percentile falls between two data items.

Exceptions:

761 Need at least 99 data items for Percentiles.

Set/Ask Routines

These routines govern the workings of other routines described in this section.

SetSD (f)

Control whether variance, standard deviation, and coefficient of variance are computed by using n, the number of non-missing items, in the denominator, or the more common $n - 1$.

By default, this Toolkit uses $n - 1$. Call SetSD with f=0 to use n, or f=1 to revert to using $n - 1$.

AskSD (f)

Opposite of *SetSD*. Returns 0 if variance calculations use n in the denominator, 1 if they use $n - 1$.

SetSheppard (f)

SetSheppard controls whether or not Sheppard's correction is used when calculating grouped statistics. By default, it is not used. Call *SetSheppard(1)* to turn on Sheppard's correction in all subsequent calls to GroupedStat.

AskSheppard (f)

The opposite of *SetSheppard*. Returns 1 if Sheppard's correction will be used for later calls to GroupedStat, 0 otherwise.

SetBoxVert (f)

SetBoxVert controls whether box plots are drawn horizontally (the default) or vertically. Call *SetBoxVert* with f = 0 to get horizontal plots, or 1 for vertical plots.

AskBoxVert (f)

The opposite of *SetBoxVert*. Returns 1 if subsequent box plots will be drawn vertically, 0 if horizontally.

Frequency Distributions

This section describes frequency distributions and how to create and use them with the *Statistics Graphics Toolkit*. Think of a frequency distribution as a histogram; indeed, a histogram is simply a picture of a frequency distribution.

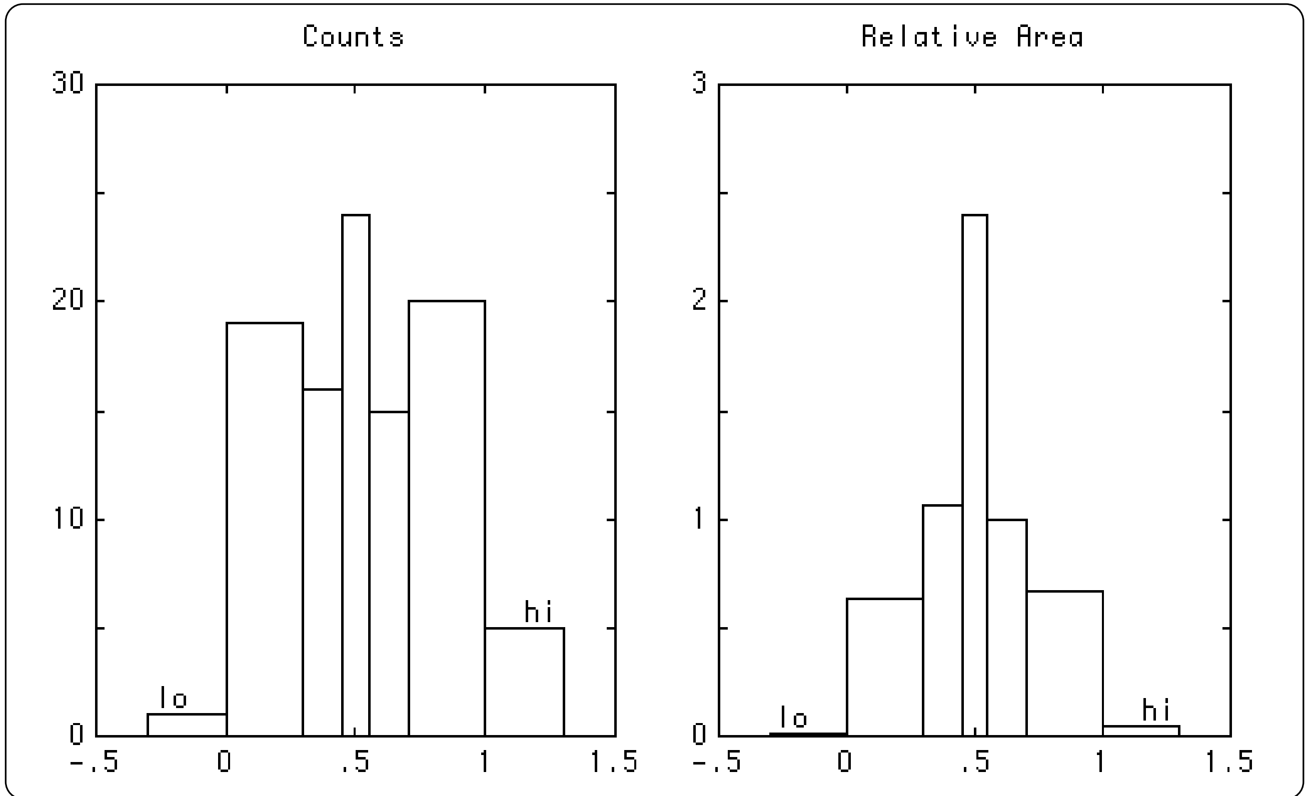

Figure 43.16: Output of the HISTAREA Program.

The “count” of each interval is the number of items in a dataset that is *less than or equal to* the top end of the interval, but not in any interval farther to the left.

Absolute vs. Relative Frequency Distributions

In the simplest terms, frequency distributions contain counts and so are integers. However, true histograms show *relative frequency distributions* in which each count is a fraction, and the total area (count x interval width) sums to 1. *Percent* frequency distributions are like relative distributions, but the area sums to 100 instead of 1.

Cumulative Frequency Distributions

Cumulative frequency distributions — also known as *ogives* — keep running totals of the counts (or relative counts) in the intervals. Thus each count is ≤ the count to its right, since its value gets included in that of its right neighbor.

Low and High Values

We have neglected the question of data values that are below the lowest interval or above the highest. Therefore, the Toolkit's frequency datasets include two more counts: one for those data values below the intervals and one for those above. These are called the *low* and *high* values.

Bar Heights vs. Areas

Remember that relative histograms and frequency polygons plot *areas* of intervals, not heights! If two intervals have the same count, therefore, the wider interval will be shorter than the narrower interval.

The figure below illustrates. When counts are plotted, each bar's *height* indicates its count; but when you plot a relative histogram, each bar's *area* (interval width x fractional count) indicates its relative size.

Frequency Dataset Formats

This Toolkit stores frequency datasets in a special kind of array $f(i,j)$ in the following format:

		j		
		-1	0	+1
	low	*-Maxnum*	count	low end - *epsilon*
		low end #1	count	high end #1
		low end #2	count	high end #2
i		...	...	...
		low end #*n*	count	high end #*n*
	high	high end #*n*	count	*Maxnum*

The *Statistics Graphics Toolkit* gives two basic ways to construct frequency datasets. One is suitable for cases in which every bar is the same width; the other for cases in which bars have different widths.

Making "Same-Width" Frequency Datasets

To create frequency datasets where each interval is the same width, you give three parameters:

from	the centerpoint of the lowest interval
step	the distance between centerpoints
to	the top end of the frequency distribution

To be precise, the topmost interval has its centerpoint at the largest number $from + k * step \leq to$. Thus if $from = 1$, $step = 2$, and $to = 6$, the topmost bar has its centerpoint at 5. For all distributions, *from* must be less than *to*, and there must be at least one interval! The interval's "height" is the count of how many raw data items fit inside that bar's interval. Intervals are defined as $pt\text{-}step/2 < x \leq pt+step/2$.

DataToFreq (data(), from, to, step, freq(,), n)

Convert the raw data items from *data*() into frequency counts in *freq*(,). The *from, to,* and *step* variables define the intervals as described above. Note that *freq*(,) will contain integer counts, and that the first element in *freq*(,) will be the "low" count, and the last element the "high" count. Missing data items in *data*() are simply ignored. The total number of non-missing items is returned in *n*. Note that *data*() and *freq*(,) need not have the same bounds; the lower bound for *freq*(,) is left untouched and the upper bound adjusted.

DataToRelFreq (data(), from, to, step, freq(,), n)

This is precisely like DataToFreq except that the resulting *freq*(,) array will contain *relative* frequencies — each count is divided by *n* to give its relative value. Missing data items in *data*() are simply ignored. The total number of non-missing items is returned in *n*. Note that the lower bound for *freq*(,) is left untouched and the upper bound adjusted as needed.

DataToCum (data(), from, to, step, freq(,), n)

Convert raw data items from *data*() into cumulative frequency counts in *freq*(,). The *from, to,* and *step* variables define the intervals as described above. Note that *freq*(,) will contain integer counts; its first element will be the "low" count, and its last the cumulative "high" count, i.e., *n*. Missing data items in *data*() are simply ignored. The number of non-missing items is returned in *n*. The lower bound for *freq*(,) is left untouched and its upper bound adjusted as needed.

DataToRelCum (data(), from, to, step, freq(,), n)

DataToRelCum is like DataToCum but produces a *relative* cumulative frequency dataset.

Making "Irregular" Frequency Datasets

To use frequency datasets where intervals have different widths, you give an array of strings that describe the intervals. For example:

```
dim int$(5)
mat read int$
data 10:20, 20:40, 50:100, 100:250, 250:1000
```

This array *int$* defines 5 intervals: $10 < x \leq 20$; $20 < x \leq 45$; $45 < x \leq 100$; $100 < x \leq 250$; and $250 < x \leq 1000$.

Intervals must be given in ascending order and cannot overlap. If two intervals don't meet, as in 40 and 50 above, the halfway point is taken as the true interval marker.

The Toolkit automatically adds low and high intervals if you don't supply them. If you want to add either one yourself (so you can supply a count for TableToFreq) give a null string as the first and/or last interval description.

DataToIrrFreq (int$(), data(), freq(,))

Given *data*() items and an interval description *int$*() as described above, create the associated frequency dataset *freq*(,). Remember: *freq*(,) will contain low and high intervals even if you don't supply them in *int$*().

See IRRDATA on your diskette for an example.

TableToFreq (int$(), counts(), freq(,))

Given an interval description *int$*() as described above, and the *counts*() for each interval, create the associated frequency table. The arrays *int$*() and *counts*() must have the same bounds but *freq*(,) can have any bounds; it will be adjusted as necessary.

See IRRTABLE on your diskette for an example.

Relative and Cumulative Frequencies

Once you have a frequency dataset, you can convert it to a relative or cumulative frequency dataset.

FreqToRelFreq (freq(,), rfreq(,))

Converts a frequency distribution to a relative frequency distribution. Note that *freq*(,) and *rfreq*(,) need not have the same bounds; the lower bound for *rfreq*(,) is left untouched and its upper bound adjusted. SetHist("%") makes the area sum to 100; otherwise it sums to 1.

FreqToCum (freq(,), cfreq(,))

Converts a frequency distribution to a cumulative frequency distribution. Note that *freq*(,) and *rfreq*(,) need not have the same bounds; the lower bound for *rfreq*(,) is left untouched and its upper bound adjusted.

Call SetReverseCum(1) before calling this routine to create a "reverse" cumulative frequency dataset with 1 at the low end of *cfreq*(,) and 0 at the high end.

SetReverseCum (f)

Call *SetReverseCum(1)* to create *reverse* cumulative frequency distributions, that is, high at the low end tapering off to zero at the high end. Use 0 to switch back to the default.

This routine also controls plotting of cumulative histograms and frequency polygons.

AskReverseCum (f)

This is the opposite of *SetReverseCum*. Returns 1 if creating reverse frequency distributions, 0 otherwise.

Histograms and Frequency Polygons

This section describes how to plot histograms or frequency polygons of one-sample data distributions. You can plot these graphs either for raw datasets or for frequency datasets (created, say, by the DataToFreq routine).

Options

Reverse cumulative graphs. To plot reverse cumulative graphs, call SetReverseCum(1) before calling any of these routines.

Normalized areas. By default, all histograms and frequency polygons are shown as counts rather than as areas. Call SetHist to switch to displaying areas.

Superimposed normal curves. Call SetNormal(1) to automatically superimpose normal curves on your histograms or frequency polygons.

Normal residuals. See the description of PlotNormFit, in the "Residuals" section, for a way to compare histograms or frequency polygons against the normal curve.

Colored bars. By default, all bars are drawn in outline only. Call SetHistoColor to set the bar colors.

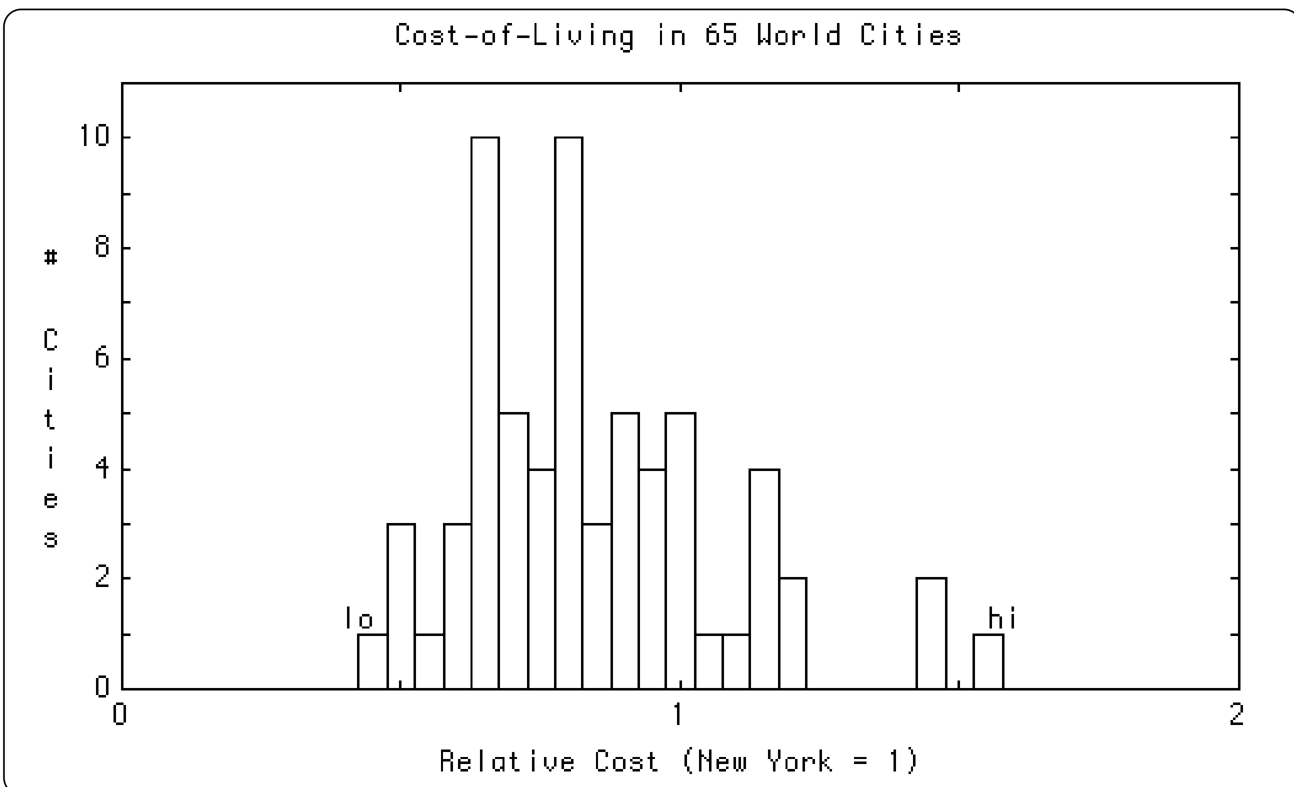

Figure 43.17: Output of the HIST program.

PlotHist (data(), from, to, step, col$)

Plot the frequency distribution of the raw dataset *data*(), grouped into bars with centerpoints at *from*, *from+step*, ..., *from+k*step* ≤ *to*. All *data*() values lower than *from* are included in a "lo" bar centered at *from-step*, and all high values are included in a "hi" bar centered one step beyond the interval's end. If there are any low or high values, these bars are flagged with labels **lo** or **hi** as appropriate.

The rules governing *from*, *to*, and *step* are described in the "Frequency Datasets" section. Any missing values in *data*() are ignored. The color scheme *col$* is treated as usual, so a scheme like "red green blue" gives a red title, green frame, and blue histogram.

Call *SetNormal(1)* to automatically superimpose normal curves on subsequent histograms. See its description at the end of this section.

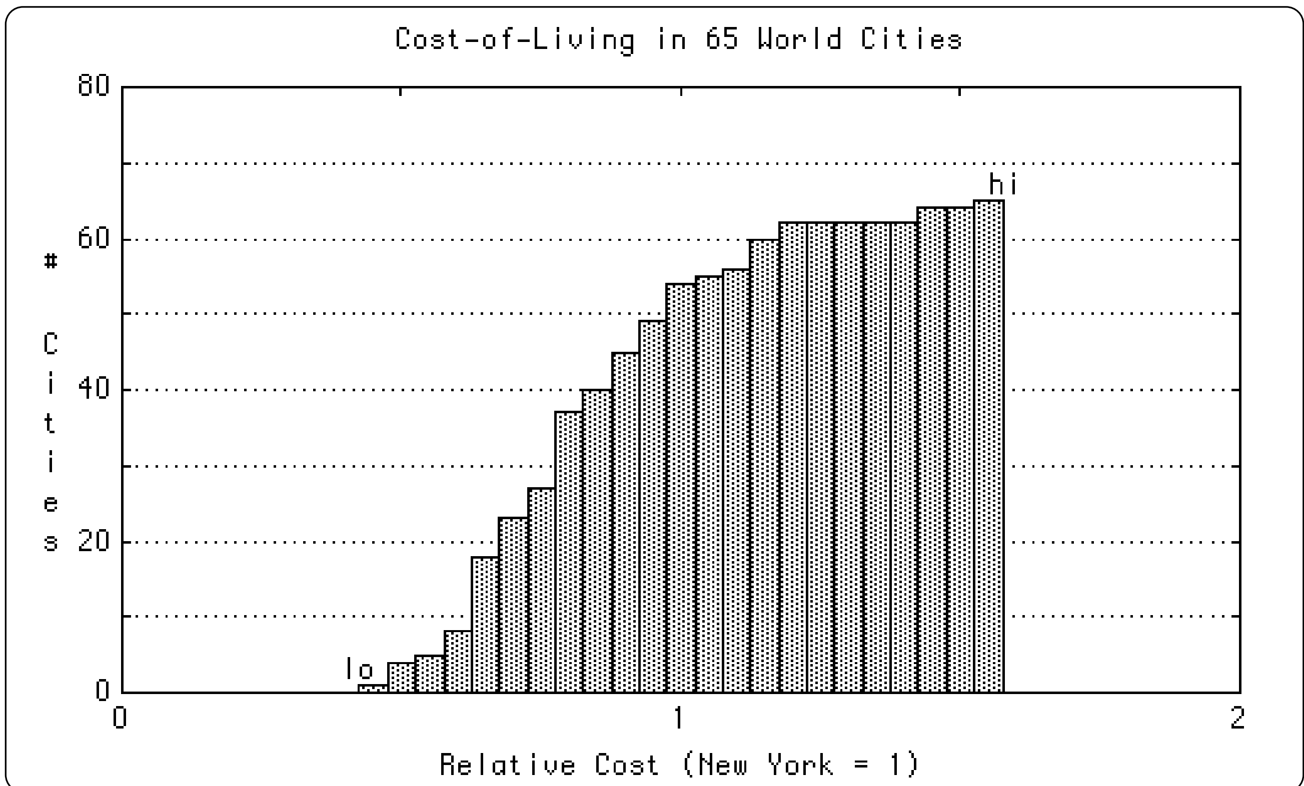

Figure 43.18: Output of the CUMHIST program.

PlotCumHist (data(), from, to, step, col$)

This routine is precisely like PlotHist except that it plots a *cumulative* histogram.

It plots a cumulative frequency distribution of the raw dataset *data*(), grouped into bars with centerpoints at *from*, *from+step*, ..., *from+k*step* ≤ *to*. All *data*() values lower

than *from* go in a “lo” bar centered at *from-step*; high values go in a “hi” bar centered one step beyond the interval’s end. These bars are flagged with labels **lo** or **hi** if non-empty.

The rules governing *from*, *to*, and *step* are described in the “Frequency Datasets” section. Any missing values in *data*() are ignored. The color scheme *col$* is treated as usual, so a scheme like “red green blue” gives a red title, green frame, and blue histogram.

Call SetNormal(1) to automatically superimpose cumulative normal curves on subsequent histograms. See its description at the end of this section.

PlotFP (data(), from, to, step, ps, ls, col$)

This routine is like *PlotHist,* except that it plots a frequency polygon instead of a histogram.

It plots the frequency distribution of the raw dataset *data*(), grouped with centerpoints at *from*, *from+step*, ..., *from+k*step* ≤ *to*. All *data*() values lower than *from* are included in a “lo” point centered at *from-step*, and all high values in a “hi” point centered one step beyond the interval’s end. If there are low or high values, these end points are flagged with labels **lo** or **hi**.

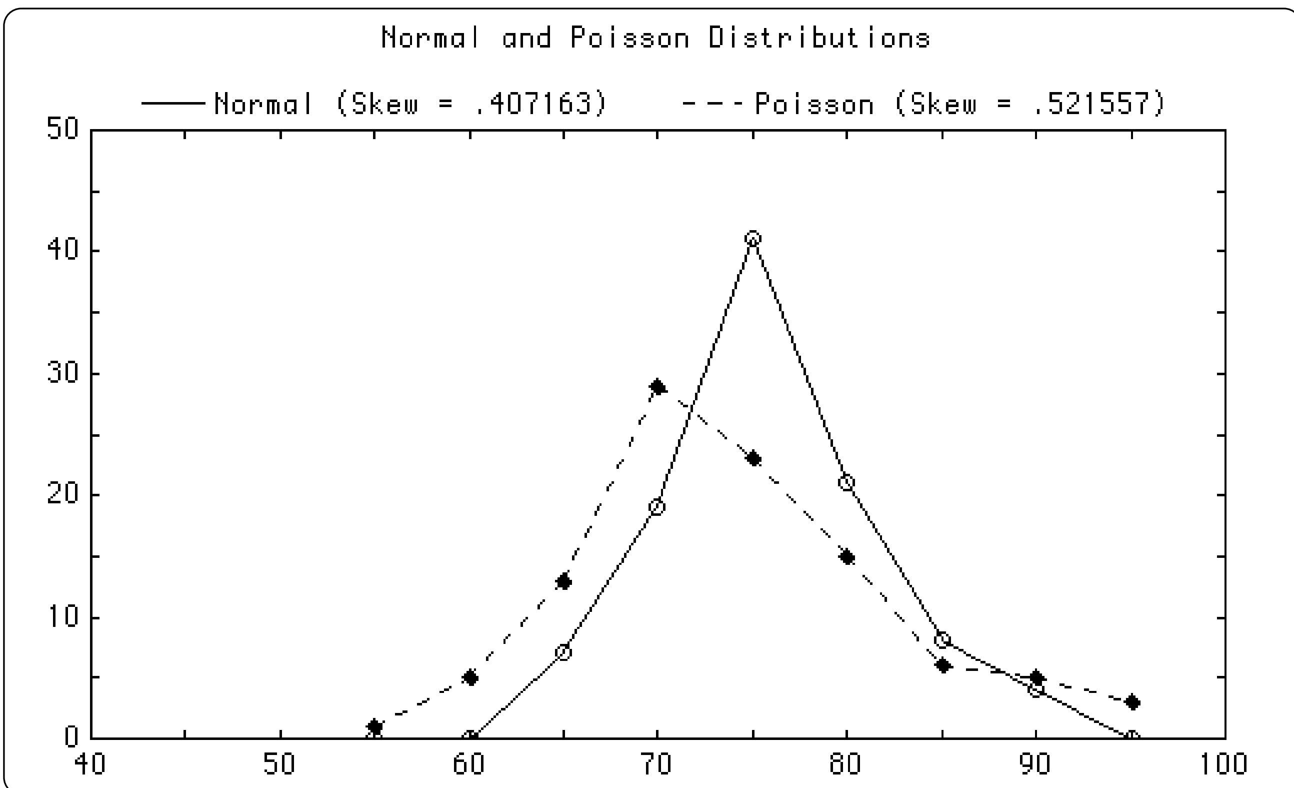

Figure 43.19: Output of the FP program.

Centerpoints are drawn in the *ps* point style connected by lines drawn in the *ls* line style. If *ps* = 0, no points are shown. If *ls* = 0, no connecting lines are shown. There are currently 13 point styles and 4 line styles to choose from; see the "Making Graphs" section for information. The rules governing *from*, *to*, and *step* are given in the "Frequency Datasets" section. Missing values in *data*() are ignored. The color scheme *col$* is treated as usual, so a scheme like "red green blue" gives a red title, green frame, and blue polygon.

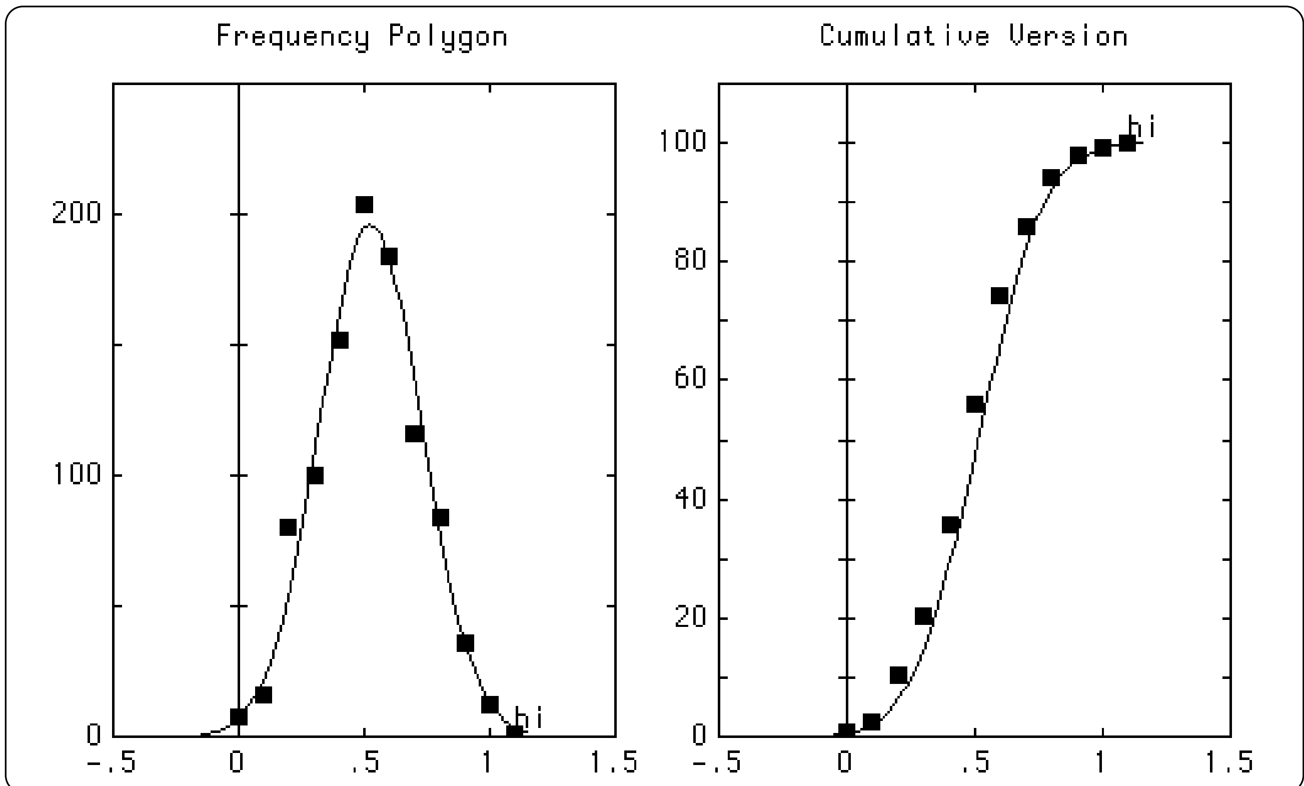

Figure 43.20: Output of the CUMFP program.

PlotCumFP (data(), from, to, step, ps, ls, col$)

This routine is precisely like PlotFP except that it plots the *cumulative* frequency polygon of the raw dataset *data*(), grouped with centerpoints at *from*, *from+step*, ..., *from+k*step* ≤ *to*. All *data*() values less than *from* go in a low point centered at *from-step*; all high values go in a high point centered one step beyond the interval's end. If there are low or high values, they are flagged **lo** or **hi** as appropriate.

Centerpoints are drawn in the *ps* point style connected by lines drawn in the *ls* line style. If *ps* = 0, points are omitted. If *ls* = 0, connecting lines are omitted. There are currently 13 point styles and 4 line styles; see the "Making Graphs" section for information.

The rules for *from*, *to*, and *step* are given in "Frequency Datasets." The color scheme *col$* is treated as usual, so "red green blue" gives a red title, green frame, and blue frequency polygon.

PlotHistFromFreq (freq(,), col$)

This routine is like *PlotHist* but takes data from *freq*(,), a frequency dataset, rather than a raw dataset.

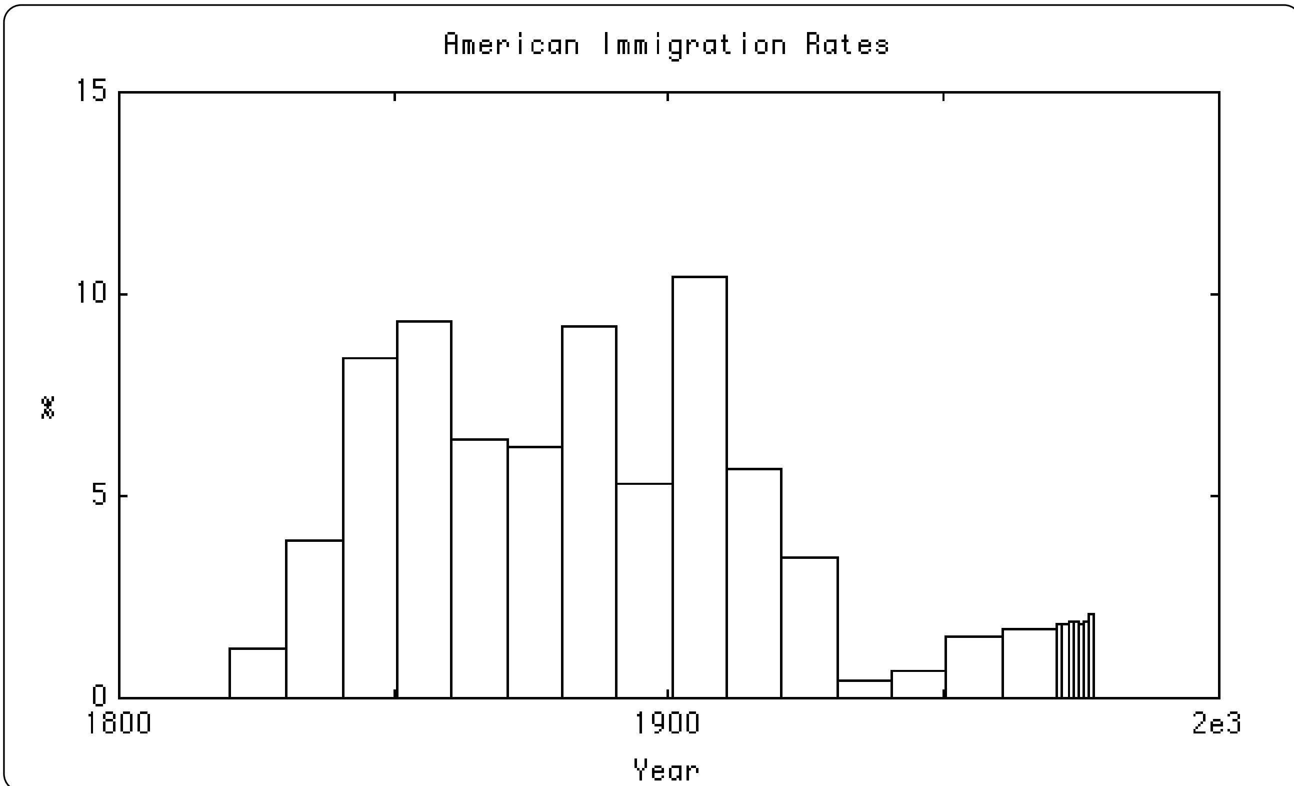

Figure 43.21: Output of the IRRTABLE program.

PlotCumHistFromFreq (freq(,), col$)

Like *PlotHistFromFreq* but plots a *cumulative* histogram. See Figure 43.22.

PlotFPfromFreq (freq(,), ps, ls, col$)

This routine is like *PlotHistFromFreq*, except that it plots a frequency polygon instead of a histogram. It plots the centerpoints of the frequency dataset *freq*(,) as the *x* coordinates, with counts as *y* coordinates. If there are low or high values, these end points are flagged with labels **lo** or **hi** as appropriate.

The centerpoints are drawn in the *ps* point style connected by lines drawn in the *ls* line style. If *ps* = 0, no points are shown. If *ls* = 0, no connecting lines are shown. There are currently 13 point styles and 4 line styles to choose from; see the “Making Graphs” section for information.

The color scheme *col$* is treated as usual, so a scheme like “red green blue” gives a red title, green frame, and blue frequency polygon.

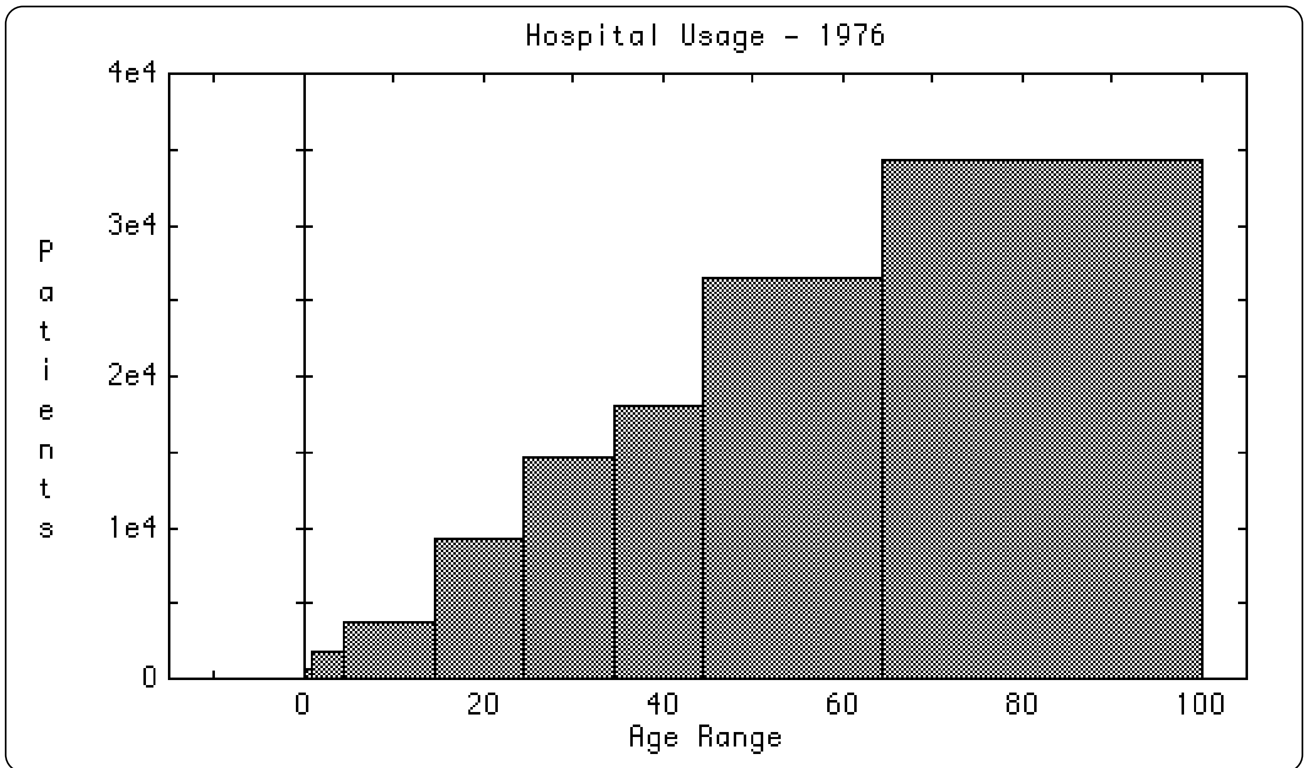

Figure 43.22: Output of the CUMIRR program.

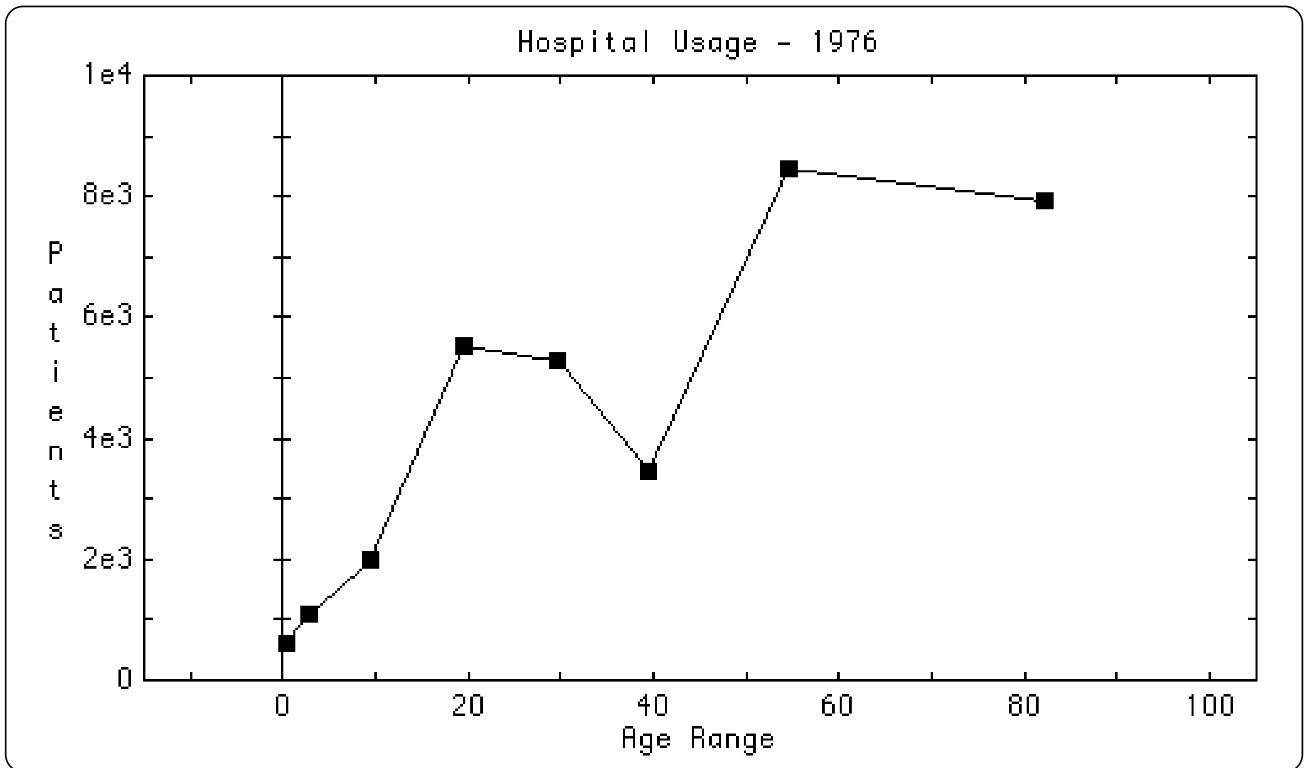

Figure 43.23: Output of the IRRFP program.

PlotCumFPfromFreq (freq(,), from, to, step, ps, ls, col$)

This routine is precisely like *PlotFPfromFreq* except that it plots the *cumulative* frequency polygon.

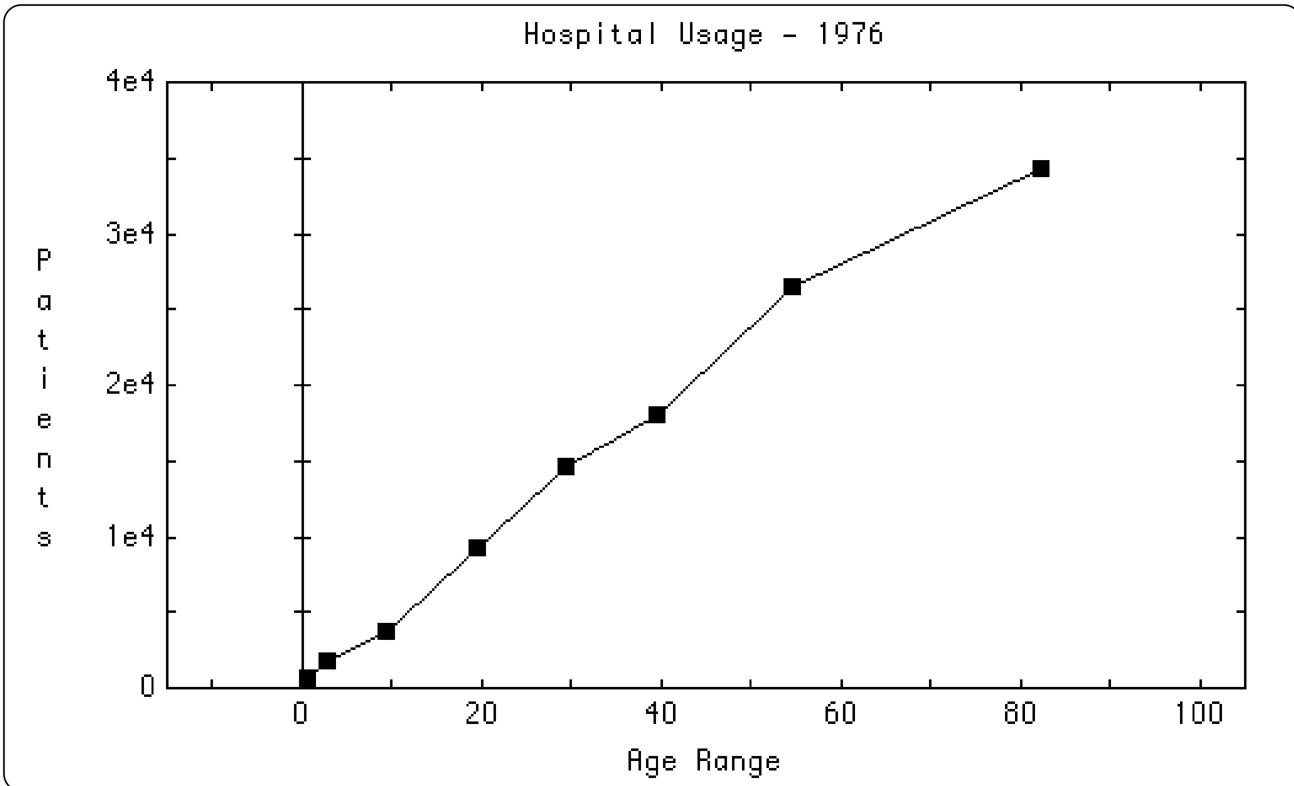

Figure 43.24: Output of the CUMIRRFP program.

AddNormalPlot (area, mean, var, ls, col$)

Draw a normal curve over the previous graph. Its shape is defined by its *area*, *mean*, and variance *var*. You can call *Stats* to find a dataset's mean and variance; the *area* is the number of non-missing elements for absolute plots, 1 for relative plots, or 100 for percentage plots.

The line style *ls* is defined in the "Making Graphs" section. Title and frame colors in the color scheme *col$* are ignored; thus, "red green blue" draws a blue line.

For most purposes you can simply call *SetNormal(1)* before drawing a histogram or frequency polygon. Then the corresponding normal curve is automatically added to the data graph. Furthermore, if the graph is cumulative, so also will be the normal curve.

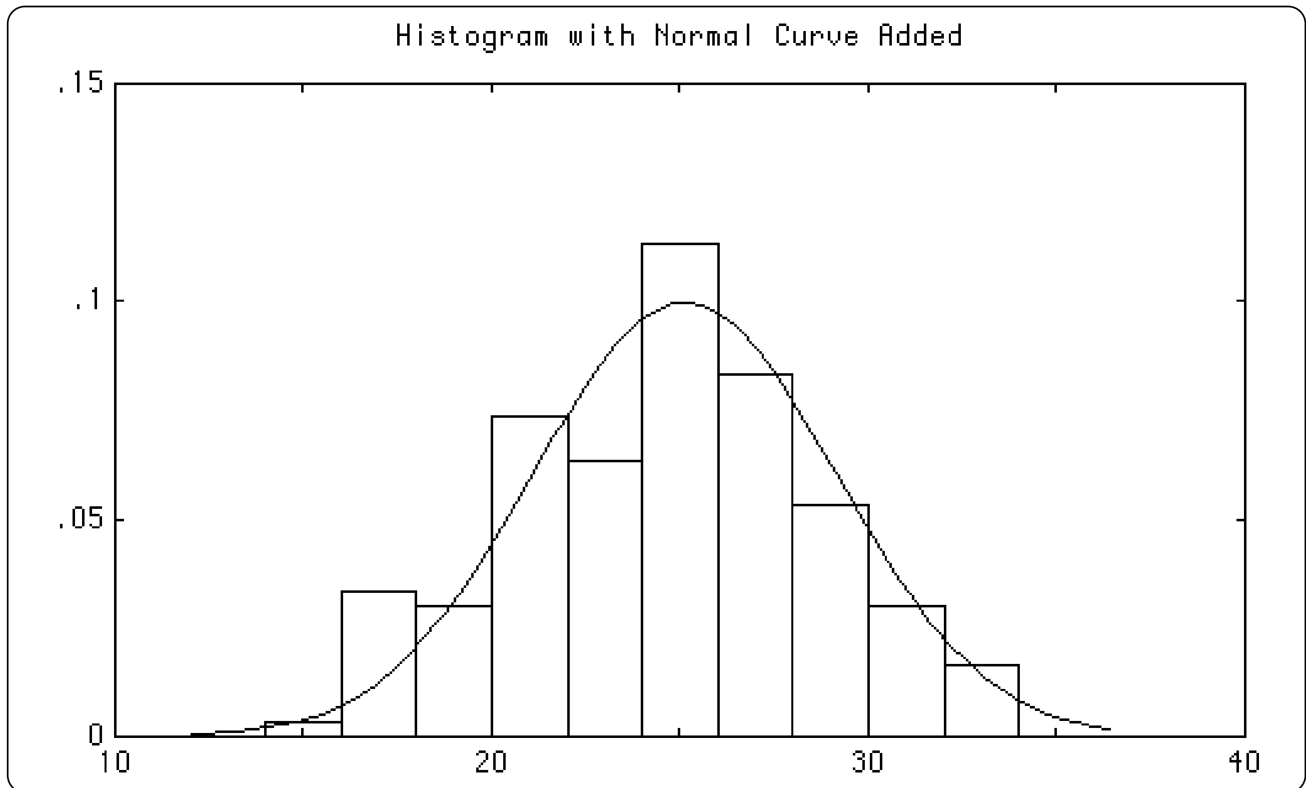

Figure 43.25: Output of the NORMAL program.

AddCumNormalPlot (area, mean, var, ls, col$)

Just like *AddNormalPlot* except that it adds the *cumulative* normal curve.

SetHist (type$)

Switch histogram or frequency polygon types. You can use any mixture of upper and lower case.

Type$	**Frequency Dataset contains...**
"COUNT"	Height = Absolute counts. **Default.**
"REL"	Area = Relative proportions (0 to 1).
"%"	Area = Percentages (0 to 100).

SetHist also controls *DataToRelFreq,* and all other routines that convert to relative frequencies. It governs whether they convert to a relative area of 1 or 100.

AskHist (type$)

The opposite of *SetHist*. Return the current histogram (or frequency polygon) type in upper case.

SetHistoColor (col$)

Set the color used for subsequent histogram bars. This color can be a True BASIC color name, such as “red” or a number such as “1”.

AskHistoColor (col$)

The opposite of *SetHistoColor*. Return the current histogram bar color.

SetNormal (f)

Call *SetNormal(1)* to add a normal curve automatically to every subsequent histogram and frequency polygon. The normal curve will be automatically scaled to fit the data’s area, mean and variance, and will be shown as a cumulative normal curve for cumulative plots. Use 0 to turn off automatic normal curves.

AskNormal (f)

The opposite of *SetNormal*. Returns 1 if subsequent histograms and frequency polygons will have normal curves added automatically. Otherwise returns 0.

T-tests and Confidence Intervals

This section describes T-tests and the confidence interval calculations that rely on them. See the “Nonparametric Tests” section for ways of estimating point locations and confidence intervals without using T-tests.

Ttest (data(), null, alt, t, p, mean, se, df)

One sample T-test. You must supply the dataset in *data*() and the null and alternate tests in *null* and *alt*. The null hypothesis is that the mean value of the population equals whatever you passed in *null*.

The *alt* parameter lets you construct one- or two-sided tests: a negative number means the alternative is that the true mean is less than *null*; 0 means that it’s not equal; and positive means that it’s greater. Thus the test is two-sided if *alt* = 0; otherwise it’s one-sided.

Example: Ttest with *null* = 3 and *alt* = 0 gives a two-sided T-test. The null hypothesis is that the true mean is 3. The alternative is that the mean is not 3.

The test returns:

t	the T-statistic
p	significance probability of *t*
mean	mean value of *data*()
se	standard error of the mean
df	degrees of freedom

Ttest2 (a(), b(), null, alt, t, p, mean, se, df)

Two sample T-test with assumption of equal variance for *a*() and *b*(). You must supply the datasets in *a*() and *b*(), and the null and alternate tests in *null* and *alt*. The null hypothesis is that the true mean difference of the populations equals whatever you passed in *null*.

The *alt* parameter lets you construct one- or two-sided tests: a negative number means the alternative is that the true mean is less than *null*; 0 means that it's not equal; and positive means that it's greater. Thus the test is two-sided if *alt* = 0; otherwise it's one-sided.

Example: Ttest2 with *null* = 17 and *alt* = –1 gives a one-sided T-test. The null hypothesis is that the difference of population means is 17. The alternative hypothesis is that the mean is less than 17.

The test returns:

t	the T-statistic
p	significance probability of *t*
mean	mean of *a*() minus mean of *b*()
se	standard error
df	degrees of freedom

Use Ftest, described at the end of this section, to check that two normally-distributed datasets come from populations with equal variances. If your datasets don't have equal variances, use Ttest2Var instead of Ttest2.

Ttest2Var (a(), b(), null, alt, t, p, mean, se, df)

Ttest2Var gives a two sample T-test without an assumption of equal variance for a() and b(). You must supply the datasets in *a*() and *b*(), and the null and alternate tests in *null* and *alt*. The null hypothesis: the true mean difference of the populations equals whatever you passed in *null*.

The *alt* parameter lets you construct one- or two-sided tests: a negative number means the alternative is that the true mean is less than *null*; 0 means that it's not equal; and positive means that it's greater. Thus the test is two-sided if *alt* = 0; otherwise it's one-

sided. The Smith-Satterthwaite approximation (below) is used to compensate for unequal variances in the two datasets.

Example: Ttest2Var with *null* = 0.5 and *alt* = 1 gives a one-sided T-test. The null hypothesis is that the difference of population means is 0.5; the alternative hypothesis is that the difference is greater than 0.5. See TTESTVAR, on your diskette, for a real example.

The test returns:

t	the T-statistic
p	significance probability of *t*
mean	mean of *a*() minus mean of *b*()
se	standard error
df	degrees of freedom

If n_a and n_b are the number of items in *a*() and *b*(), and *ssa* is $\sum(a_i - abar)^2$ and *ssb* is $\sum(b_i - bbar)^2$, Smith-Satterthwaite approximates a t-distribution with *df* computed as follows (rounded down to an integer).

$$\frac{1}{df} = \frac{K^2}{n_a - 1} + \frac{(1-K)^2}{n_b - 1}, \quad \text{where } K = \frac{\dfrac{ssa}{n_a - 1}}{\dfrac{ssa}{n_a - 1} + \dfrac{ssb}{n_b - 1}}$$

Ttest2MP (a(), b(), null, alt, t, p, mean, se, df)

Ttest2MP gives a two sample T-test for matched pairs a() and b(). You must supply the datasets in *a*() and *b*(), and the null and alternate tests in *null* and *alt*. The null hypothesis is that the true mean difference of the populations equals whatever you passed in *null*.

Arrays *a*() and *b*() must be the same size. If either element in a pair is missing, the pair is ignored.

The *alt* parameter lets you construct one- or two-sided tests: a negative number means the alternative is that the true mean is less than *null*; 0 means that it's not equal; and positive means that it's greater. Thus the test is two-sided if *alt* = 0; otherwise it's one-sided.

Example: Ttest2MP with *null* = 3 and *alt* = 0 gives a two-sided T-test. The null hypothesis is that the difference of population means is 3. The alternative hypothesis is: the difference of means is not 3.

The test returns:

t	the T-statistic
p	significance probability of *t*
mean	mean of *a*() minus mean of *b*()
se	standard error
df	degrees of freedom

See TTEST2MP, on your diskette, for an example.

ConfInt (data(), ci, left, right, se, df)

Use the one-sample T-test to compute a confidence interval for the true mean of a population. You must supply the dataset in *data*() and the desired confidence level in *ci*. For example *ci* = .95 gives a 95% confidence interval.

The test returns:

left	left edge of confidence interval
right	right edge of confidence interval
se	standard error
df	degrees of freedom

ConfInt2 (a(), b(), ci, left, right, se, df)

Use the two-sample T-test, with assumption of equal variance of *a*() and *b*(), to compute a confidence interval for the difference of the population means. You must supply the datasets in *a*() and *b*() and the desired confidence level in *ci*. For example *ci* = .95 gives a 95% confidence interval.

The test returns:

left	left edge of confidence interval
right	right edge of confidence interval
se	standard error of the difference used
df	degrees of freedom

ConfInt2Var (a(), b(), ci, left, right, se, df)

Use the two-sample T-test, with no assumption of equal variance of *a*() and *b*(), to compute a confidence interval for the difference of the population means. The Smith-Satterthwaite approximation is used to compensate for the differing variances. You must supply the datasets in *a*() and *b*() and the desired confidence level in *ci*. For example *ci* = .95 gives a 95% confidence interval.

The test returns:

left	left edge of confidence interval
right	right edge of confidence interval
se	standard error of the difference used
df	degrees of freedom

See TTESTVAR, on your diskette, for an example.

ConfInt2MP (a(), b(), ci, left, right, se, df)

Use the matched-pairs T-test to compute a confidence interval for the difference of the population means. You must supply the datasets in *a*() and *b*()and the desired confidence level in *ci*. For example *ci* = .95 gives a 95% confidence interval.

The test returns:

left	left edge of confidence interval
right	right edge of confidence interval
se	standard error of the difference
df	degrees of freedom

See TTEST2MP, on your diskette, for an example.

ConfPlot (data(), ci, col$)

Plot a “confidence interval” plot of the dataset *data*(). These confidence intervals are based on T-tests. The plot is scaled so all data points are within the graph coordinates. Your confidence interval *ci* should be a number such as .95 for a 95% confidence interval.

The color scheme *col$* has the usual meaning; for instance, “red green blue” gives a red title, green frame, and blue box plot.

By default, the actual data values are not displayed. Call *SetDataStyle(ps)* with $ps \neq 0$ to display data points. Intervals and points are drawn in the same colors, but points are lowered slightly to make them visible as shown in the figure below.

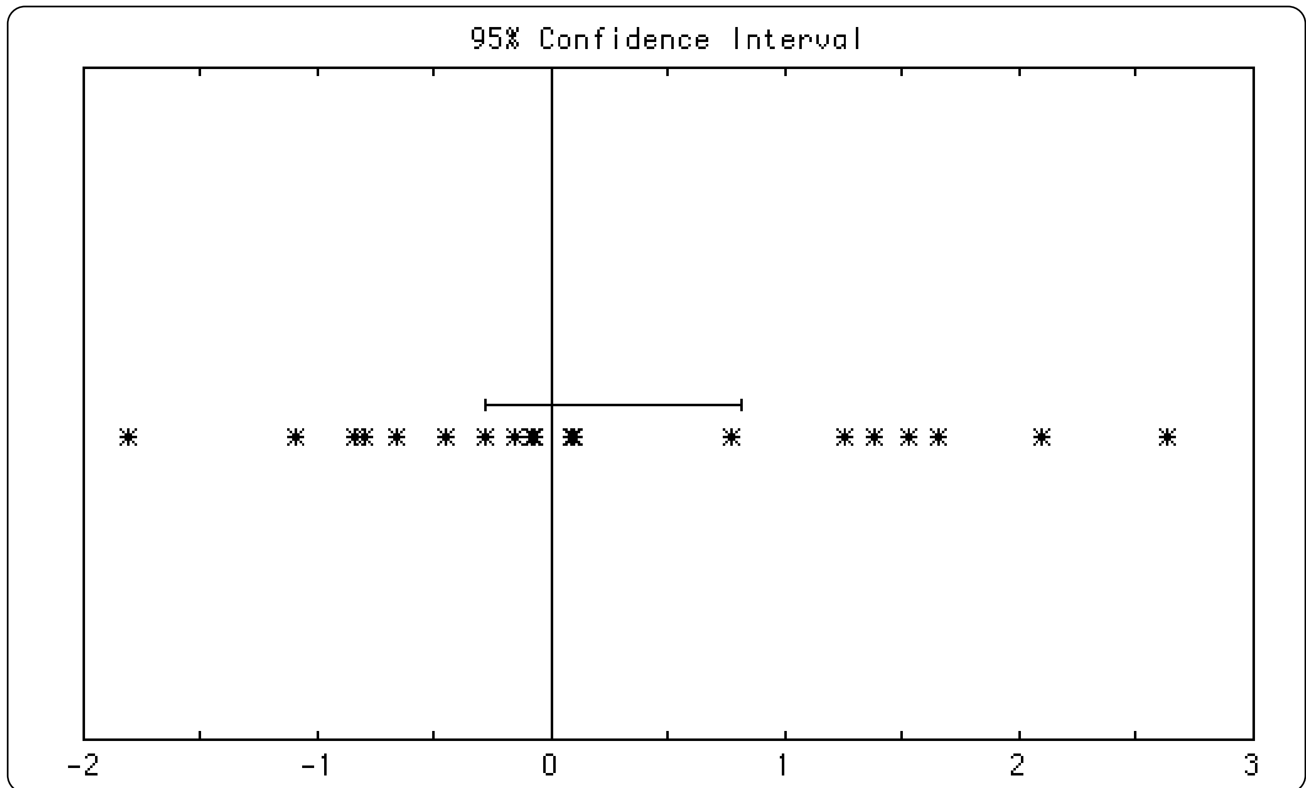

Figure 43.26: Output of the CONF program.

ManyConfPlot (data(,), names$(), ci, col$)

Plot stacked "confidence interval" plots of the datasets *data*(,). These confidence intervals are based on T-tests. Each column of the *data*(,) is taken as an independent dataset, and gets its own confidence interval plot. The plot is scaled so all data points are within the graph coordinates.

Your confidence interval *ci* should be a number such as .95 for a 95% confidence interval.

Datasets need not have the same numbers of elements; just pad the short datasets with missing values. Each dataset's name is displayed on the frame beside its box plot. Pass the names in the *names$*() array. Size(*names$*) must equal Size(*data*,1).

The color scheme *col$* has the usual meaning; thus, "red green blue" gives a red title, green frame, and blue box plot. If you give multiple data colors, the intervals are drawn with that sequence of colors.

By default, the actual data values are not displayed. Call SetDataStyle(*ps*) with $ps \neq 0$ to display data points. The first dataset is drawn with point style *ps*, the next with *ps*+1, and so forth. When the end of the point styles are reached, the styles will cycle back to style 2 (skipping style 1 because it's hard to see). Intervals and points are

drawn in the same colors, but the points are lowered slightly to make them visible as shown in the figure on the facing page.

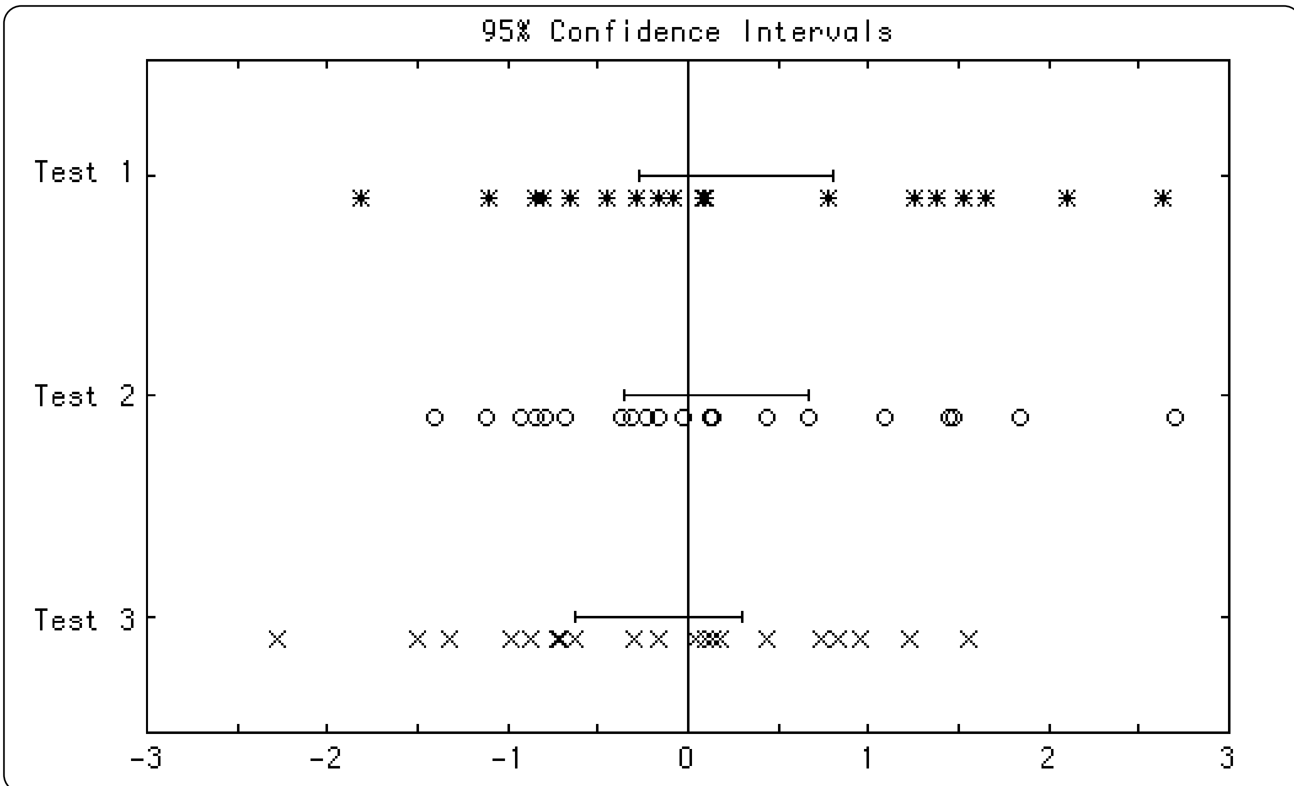

Figure 43.27: Output of the MANYCONF program.

Ftest (a(), b(), f, p)

Perform an F-test on two datasets *a*() and *b*() from normally distributed populations to see if the populations have equal variances.

The test returns:

f	the F-statistic
p	significance probability for test of equal variance

WARNING: The F-test is very sensitive to deviations from a normal distribution. If normality is not assured, use the nonparametric Siegel-Tukey test instead.

It is also wise to remember that *p* is an approximation — for small samples, use a table of critical values instead! See *Applied Statistics* by L. Sachs, pp. 260-264, or F-TEST, on your diskette, for an example.

Chi-Square and Contingency Tables

This section describes contingency tables and chi-square (c^2) tests. The *Statistics Graphics Toolkit* handles most of these cases in full generality; it has only a few special routines for 2 x 2 tables.

ChiSq (data(,), chi2, df, p)

Perform a chi-square test on tabular data. The *data*(,) array can have any numbers of rows and columns. Call SetYates(1) if you want to use Yates correction for the chi-square analysis. The test returns:

chi2	chi-square χ^2
df	degrees of freedom
p	significance probability for test of independence

Note that ChiSq is just a simple interface to the ContTable routine.

FisherExact (t(,), p)

Compute Fisher exact probability for a 2 x 2 table *t*(,). The result:

p	Fisher exact probability for this table; this is *not* the significance probability!

This requires computing quite a few factorials, so results get slow as table values get big. When table values are quite big (say >50) you may get overflows, underflows, or inaccuracies due to round-off error. See FISHER, on your diskette, for an example.

Exceptions:

759	FisherExact dataset can't contain missing values.
760	FisherExact works on 2x2 tables only.

ContTable (data(,), chi2, df, p, v, cc, ccc, ca, e(,), rs(), cs(), gs)

Compute contingency table statistics based on a *data*(,) array. The *data*(,) array need not be square; but *ca* is undefined for other shape tables and for them is returned as the missing value.

The test returns:

chi2	chi-square χ^2
df	degrees of freedom
p	significance probability
v	Cramer's V (a.k.a. *phi* for 2 x 2 tables)
cc	Pearson's contingency coefficient
ccc	Pawlik's corrected *cc*
ca	correlation of attributes (square *data*(,) only)
e(,)	expected data values
rs()	row sums
cs()	column sums
gs	grand sum (total)

The *v, cc* and *ccc* coefficients are not defined for some tables; ContTable returns the missing value for those cases and does not give an error.

By default, Yates' correction for discontinuity is not used in this analysis. To get Yates' correction, call SetYates(1) before you call ContTable.

Exceptions:

732	Can't do contingency table with 0 row.
733	Can't do contingency table with 0 column.

PrintCrossTab (#n, data(,))

Perform a chi-square test on tabular data and print a cross-tabulation table. The *data*(,) array can have any numbers of rows and columns. If #*n* is #0, the result is printed in the current window.

Call SetYates(1) if you want to use Yates' correction for the chi-square analysis.

Exceptions:

732	Can't do contingency table with 0 row.
733	Can't do contingency table with 0 column.
7004	Channel isn't open.
8501	Must be text file.

```
World War II Volunteers: Occupation & Aptitude

      122        30        20       472           644
      226        51        66       704          1047
      306       115        96      1072          1589
      130        59        38       501           728
       50        31        15       249           345

      834       286       235      2998          4353

Chi-Square: 35.7989    DF: 12    Prob: 0.00035
```

Output of the CROSSTAB program.

McNemarChi (t(,), chi2, p)

Perform McNemar chi-square test on a 2 x 2 table *t*(,). This is a specialized test for the intensity of change between two dependent samples. It is very different from the ordinary chi-square test and the two cannot be used interchangeably!

The results:

chi2	McNemar chi-square statistic
p	significance probability

WARNING: McNemarChi should not be used if $b + c < 8$. The software does not check for this condition but does use a continuity correction if $b + c < 30$. Calculations are as shown below.

$t(,)$	$8 \le b + c < 30$	$b + c \ge 30$
$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$	$\chi^2 = \dfrac{(\lvert b - c \rvert - 1)^2}{b + c + 1}$	$\chi^2 = \dfrac{(b - c)^2}{b + c + 1}$

Reference: *Applied Statistics* by L. Sachs, pp. 363-365. See MCNEMAR, on your diskette, for an example.

Exceptions:

759	McNemarChi dataset can't contain missing values.
760	McNemarChi works on 2x2 tables only.

SetYates (f)

By default, Yates' correction is not used when analyzing contingency tables and chi-square statistics. Call SetYates(1) to use Yates' correction in subsequent computations; use 0 to turn it off again.

AskYates (f)

The opposite of SetYates. Returns 1 if Yates' correction will be used on subsequent contingency table and chi-square calculations; 0 otherwise.

Nonparametric Tests

This section describes the nonparametric tests included in the *Statistics Graphics Toolkit*. These kinds of tests are sometimes called distribution-free tests because they do not require the underlying populations to be normally distributed.

See the "References" for a list of books that describe statistical tests. *Nonparametric Statistical Methods* describes various nonparametric tests in detail, but you can also find good descriptions in *Statistical Analysis* and *Applied Statistics*.

KS2 (a(), b(), d, p)

KS2 gives the Kolmogorov-Smirnov test for two independent samples. This tests whether two samples *a*() and *b*() were drawn from the same population.

The output is:

d	Kolmogorov-Smirnov test statistic
p	significance probability of *d* (see Warning)

WARNING: The probability *p* is an approximation good for medium to large samples (Size(*a*)+Size(*b*) ≥ 35). For smaller samples, use a table of critical values instead!

See Siegel's *Nonparametric Statistical Methods*, pp. 394-419, for critical value tables and KS, on your diskette, for an example.

MannWhitney2 (a(), b(), ci, n, u, z, p, med, low, high)

MannWhitney2 gives the Mann-Whitney U test for two unpaired sets of data. This test helps estimate the median difference of the populations. It is equivalent to the Wilcoxon two-sample rank-sum test. Pass datasets *a*() and *b*() with a confidence interval *ci*. The test returns:

n	the number of non-missing samples from the two datasets
u	the Mann-Whitney U statistic
z	critical value for *u* (see Warning)
p	probability of *z* (see Warning)
med	estimated median difference
low	low end of confidence interval for difference
high	high end of confidence interval for difference

By default, this test does not use fractional counts when computing the confidence intervals; that is, the (*low,high*) pair are values existing in the pairwise differences between the two datasets. Call SetFracConf(1) if you wish to use fractional counts and hence interpolate values for the confidence intervals.

NOTE: The returned values *z* and *p* are obtained from a normal approximation and are reasonable so long as both datasets have at least 8 non-missing values and *n* > 60.

See *Applied Statistics*, pp. 293-303, for a critical value table. and MANNWHIT, on your diskette, for an example.

Wilcoxon1 (data(), ci, n, rhat, z, p, med, low, high)

Wilcoxon1 gives the Wilcoxon Signed-Rank Sum procedure for one dataset. This test helps estimate the median of a population. Pass dataset *data*() with a confidence interval *ci*. The test returns:

n	the number of non-missing samples in *data*()
rhat	the Wilcoxon statistic
z	critical value for *rhat*
p	probability of *z* (see Warning)
med	estimated median
low	low end of confidence interval for median
high	high end of confidence interval for median

By default, this test does not use fractional counts when computing the confidence intervals; that is, the (low,high) pair are values existing in the dataset. Call SetFracConf(1) if you wish to use fractional counts and hence interpolate values for the confidence intervals. See *Basic Statistics*, p. 310, for a critical value table.

WARNING: The returned value *p* is a good approximation so long as *n* > 25.

Wilcoxon2 (a(), b(), ci, n, w, z, p, med, low, high)

Wilcoxon2 gives the Wilcoxon Rank Sum test for two unpaired sets of data.

This test helps estimate the median difference of two populations; it's equivalent to the Mann-Whitney U test. Pass datasets *a*() and *b*() with a confidence interval *ci*. The test returns:

n	the number of non-missing samples from the two datasets
w	the Wilcoxon statistic
z	critical value for *w*
p	probability of *z* (see Warning)
med	estimated median difference
low	low end of confidence interval for difference
high	high end of confidence interval for difference

By default, this test does not use fractional counts when computing the confidence intervals; that is, the (*low,high*) pair are values existing in the pairwise differences between the two datasets. Call SetFracConf(1) if you wish to use fractional counts and hence interpolate values for the confidence intervals.

WARNING: The returned value *p* is a good approximation so long as *n* > 25.

See *Basic Statistics*, p. 310, for a critical value table, and SIGNRANK, on your diskette, for an example.

Wilcoxon2MP (a(), b(), ci, n, rhat, z, p, med, low, high)

Wilcoxon2MP gives the Wilcoxon Rank Sum test for matched pairs. This test helps

estimate the median difference. Pass datasets *a*() and *b*() with a confidence interval *ci*. The two datasets must have the same number of elements. If either element of a pair is missing, the entire pair is discarded. The test returns:

n	the number of non-missing pairs from the two datasets
rhat	the Wilcoxon statistic
z	critical value for *rhat*
p	probability of *z* (see Warning)
med	estimated median difference
low	low end of confidence interval for difference
high	high end of confidence interval for difference

By default, this test does not use fractional counts when computing the confidence intervals; that is, the (*low,high*) pair are values existing in the pairwise differences between the two datasets. Call SetFracConf(1) if you wish to use fractional counts and hence interpolate values for the confidence intervals.

WARNING: The probability *p* is a good approximation so long as *n* > 25.

See *Applied Statistics*, pp. 312-315, for a critical value table.

Spearman (a(), b(), n, rho, t, p)

Spearman gives Spearman's rank correlation coefficient ρ (rho) for matched pairs. This tests for correlation between two datasets. Pass independent paired datasets *a*() and *b*(). The two datasets must have the same number of elements; if either element in a pair is missing, the pair is ignored. The test returns:

n	the number of non-missing pairs
rho	Spearman rank correlation coefficient
z	test statistic for *rho*
p	probability of *z* (see Warning)

There are several common techniques for computing Spearman's rho when the data has ties, and they give different results. By default, this routine uses a complicated method to compute rho for ties; see the description of SetSpearmanTie as the end of this section, and *Nonparametric Statistical Inference*, p. 234. Many statistics books give only the simple form, so you may have to call SetSpearmanTie(0) to get answers that match your statistics book.

☑ **WARNING: The *z* and *p* results are good approximations so long as *n* > 9.**

See *Applied Statistics*, pp. 396-403, for a critical value table.

KendallTau (a(), b(), n, tau, z, p)

KendallTau gives Kendall's rank correlation coefficient τ (tau) for matched pairs. This tests for correlation between two datasets and is an alternative to Spearman's rho. Pass independent paired datasets *a*() and *b*(). They must have the same number of elements; if either element of a pair is missing, the pair is ignored. The test returns:

n	the number of non-missing pairs
tau	Kendall rank correlation coefficient
z	test statistic for *tau*
p	probability of *z* (see Warning)

☑ **WARNING: The *z* and *p* results are good approximations so long as *n* > 10.**

See *Nonparametric Statistical Methods*, pp. 384-393, for a critical value table and TAU, on your diskette, for an example.

WallisMoore (data(), h, z, p)

WallisMoore gives the Wallis-Moore phase frequency test. This tests the randomness of a sequence by checking the number of "phases" – runs of increasing/decreasing data values – in *data*(). The initial and final phases are not counted. The test returns:

h	count of phases
z	test statistic for *h*
p	probability of *z* (see Warning)

☑ **WARNING: *z* and *p* are good approximations only if *n* > 10. A continuity correction is applied if Size(*data*) ≤ 30. "Zero" changes from one item to the next are treated as no change in phase. If *data*() is a single run, both *z* and *p* will be zero.**

See *Applied Statistics*, pp. 378-379, and WALLIS, on your diskette, for an example.

RunTest1 (data(), n, r, z, p)

RunTest1 gives the Wald-Wolfowitz runs test on a single sample. This test helps determine if a population's median is zero by counting runs of positive and negative numbers. Zero is treated as positive. Pass dataset *data*(). The test returns:

n	the number of non-missing samples from the dataset
r	the number of runs
z	standardized *r*
p	significance probability of *z* (see Warning)

WARNING: The *z* and *p* results are good approximations so long as *n* > 20. If the entire dataset consists of a single run, both *z* and *p* will be zero.

See *Applied Statistics*, pp. 375-378, for a critical value table.

RunTest2 (a(), b(), n, r, z, p)

RunTest2 gives the Wald-Wolfowitz runs test for unpaired samples. This test helps determine whether both samples were drawn from the same population. Datasets *a*() and *b*() need not have the same number of elements. The test returns:

n	the number of non-missing samples from the two datasets
r	the number of runs
z	standardized *r*
p	significance probability of *z* (see Warning)

WARNING: The *z* and *p* results are good approximations so long as *n* > 20.

See *Applied Statistics*, pp. 375-378, for a critical value table.

MedianTest (a(), b(), t(,), chi2, p)

MedianTest performs a median test on two independent datasets to check if the two datasets are drawn from populations with equal medians. Pass the datasets in *a*() and *b*().

The median test first pools all dataset values and finds their common median. Then it creates a 2 x 2 table from the number of values in each dataset greater or less than this common median, as shown below. And finally it uses a chi-square test to analyze this table. Call SetYates to control whether Yates' correction is used when computing the chi-square statistic.

The test returns:

t(,)	2 x 2 median contingency table
chi2	chi-square statistic for *t*(2,2)
p	significance probability of *chi2* (see Warning)

On output, *t*(,) contains the 2 x 2 contingency table, as follows, that was used for computing *chi2* and *p*:

	Number of occurrences	
Dataset	≤ common median	> common median
a()	a	b
b()	c	d

WARNING: The *chi2* and *p* results are only chi-square approximations.

For more information, see *Applied Statistics* by L. Sachs, pp. 301-302; or *Basic Statistics* by T. Kurtz, pp. 235-241, and MEDIAN, on your diskette, for an example.

SiegelTukey (a(), b(), ia, z, p)

SiegelTukey gives the Siegel-Tukey rank dispersion test, a nonparametric replacement for the F-test for equal variances. Pass the two datasets to compare in *a*() and *b*().

The test returns:

ia	Siegel-Tukey index sum for *a*()
z	approximate standard normal for *ia*
p	significance probability of *z* (see Warning) small *p* indicates unequal variances

WARNING: Both *z* and *p* are approximations that can be safely used if Size(*a*)>9 and Size(*b*)>9, or Size(*a*)>2 and Size(*b*)>20. For small sample sizes, use a critical value table for Siegel-Tukey index sums on *ia*.

See *Applied Statistics* by L. Sachs, pp. 286-289, for the algorithm and a small critical value table. See also *Nonparametric Statistical Inference* by J. Gibbons, pp. 183-184.

This algorithm makes corrections to *z* and *p* when more than one fifth of the items are involved in cross-sample ties, and corrects for "very different" sample sizes when sizes differ by a factor of 2 or more.

See SIEGEL, on your diskette, for an example.

KruskalWallisH (data(,), pairsig, h, p, diff(,))

KruskalWallisH gives the Kruskal-Wallis H-test for independent samples. Pass a *data*(,) array that contains "treatments" in its columns; that is, each column is a dataset. Short columns can be padded out with missing values. Also pass *pairsig*, the significance level at which columns are considered to be different. The test returns:

h	the H-hat statistic from the Kruskal-Wallis test; note that if more than 25% of all values are involved in ties, *h* is the corrected H-hat statistic
p	significance probability of *h* (see Warning)
diff(,)	pairs (*i,j*) of rank indices that test significantly different

For example, you could pass *pairsig* = .05 to find all columns that are different at the 5% level. If there are no such columns, Size(*diff*,1) is zero. But if there are, say, three such pairs of columns *i-j*, *i-k*, and *k-l*, then Size(*diff*,1) is 3; and *diff*(1,1) is *i*, *diff*(1,2) is *j*; *diff*(2,1) is *i*; *diff*(2,2) is *k*, etc.

By default, pairwise significances are computed by via chi-square distributions, as given in *Applied Statistics*, p. 305. To switch to Dunn's procedure for pairwise comparisons via normal distributions, call SetHtest(1) before calling this routine.

WARNING: For small samples, Size(*data*,1) < 5 or Size(*data*,2) < 4, *p* may be incorrect. Use a critical value table instead.

See *Applied Statistics*, pp. 303-306, for a critical value table.

Exception:

755 Too many identical values for Kruskal-Wallis H test.

See KRUSKAL, on your diskette, for an example.

Friedman (d(,), cr(), fs, p)

Friedman gives the Friedman rank-ANOVA test, a distribution-free two-way ANOVA for correlated samples.

Input the data in *d*(,) where each column is a treatment. The output is:

cr()	column rank sums
fs	Friedman statistic
p	significance probability of *fs* (see Warning)

WARNING: The probability *p* is an approximation good only for large samples. If you have only 2 treatments, use the Wilcoxon2MP test instead. If you have 3 treatments and <10 observations, or 4 treatments and <5 observations, do not use the *p* value; use a critical value table instead.

See *Applied Statistics*, pp. 549-553, for a critical value table.

The FRIEDMAN program, on your diskette, uses a Friedman test to evaluate whether the timings for three ways to round first base (in a baseball game) are significantly different. The data gives a series of 22 timings for the three techniques: *wide-out, narrow-angle, and round-out*.

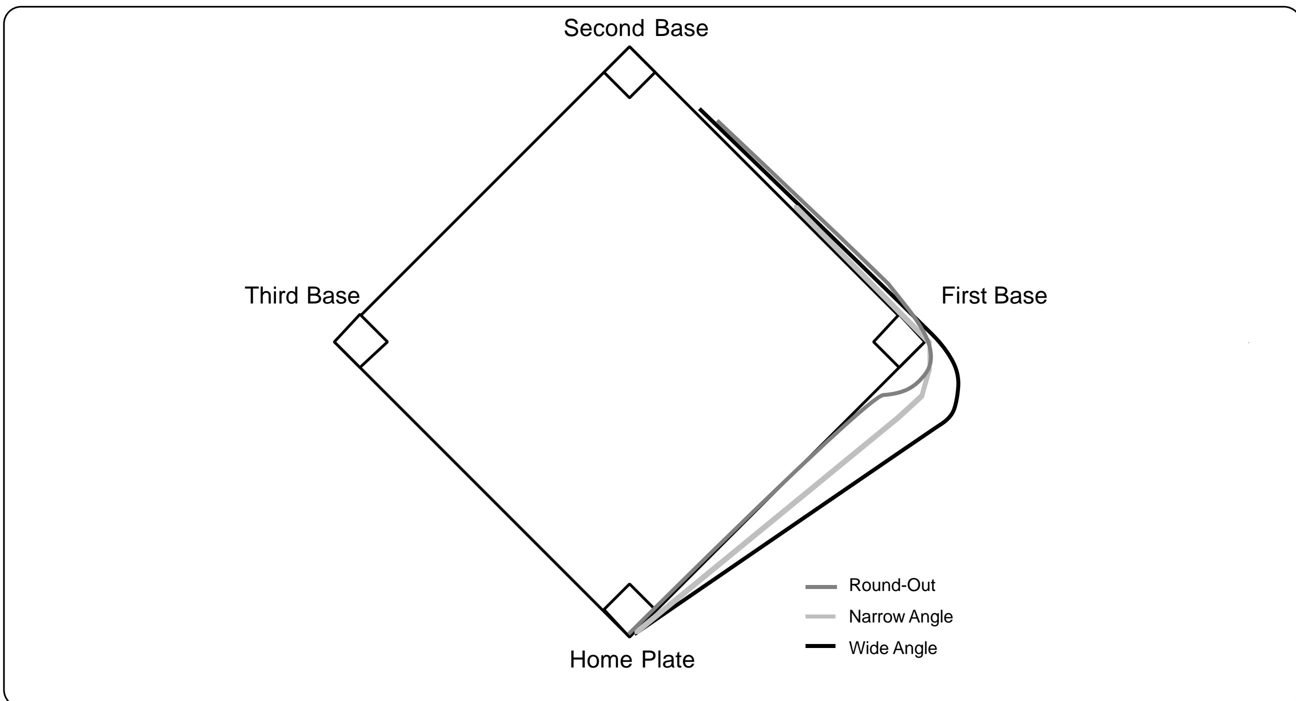

Figure 43.29: Three Methods of Rounding First Base.

The "best" way to round first base is that which takes the least time. The resulting statistic shows that there is indeed a statistically significant difference in the various ways to round first base.

The data were collected in 1970 by W. F. Woodward, shortstop of the Cincinnati Reds baseball team. The analysis and figure are taken from *Nonparametric Statistical Methods* by Hollander, p. 141.

KendallW (a(,), w, f, p)

KendallW gives Kendall's W coefficient of concordance. Pass an array $a(,)$ that contains "judges' rankings" in its columns; that is, each column is a dataset.

Each row must contain integers in the range 1 to n, where n is the number of columns; duplicates are allowed for tie values. Thus to get W for four judges' rankings of three items, you might pass:

1	2	3
2	3	1
2	1	1
2	1	3

KendallW returns:

w	W coefficient of concordance, ranging from 0 to 1 inclusive, where 1 means perfect concordance
f	F-value of *w*
p	significance probability of *f* (see Warning)

NOTE: Calculations for *f* and *p* have been taken from *Analysing Qualitative Data* by A. E. Maxwell, pp. 119-121, but Maxwell does not supply information on the accuracy of these computations. Use a critical-value table for W for anything but quick work.

See KENDALLW, on your diskette, for an example.

SetFracConf (f)

SetFracConf determines whether or not fractional values should be used to interpolate nonparametric confidence intervals. Pass $f = 0$ to forbid using fractional intervals, or any nonzero value to use them (and hence interpolate). By default, fractional intervals are not used.

AskFracConf (f)

AskFracConf is the inverse of SetHtest. It returns $f = 1$ if fractional confidence intervals will be interpolated, and zero otherwise.

SetHtest (f)

SetHtest determines which procedure will be used to compute pairwise significances in the Kruskal-Wallis H test. Pass f=0 for the usual procedure via the chi-square distribution, or any nonzero value to use Dunn's procedure instead.

AskHtest (f)

AskHtest is the inverse of SetHtest. It returns $f = 0$ if the usual procedure will be followed for comparing pairwise significances in the Kruskal-Wallis H test, or 1 if Dunn's procedure will be used instead.

SetSpearmanTie (f)

SetSpearmanTie determines which procedure will be used to compute Spearman's rho in cases with tied data. Pass $f = 0$ for the simple procedure, or any nonzero value to use the more complicated (but better) procedure instead. By default, the complicated calculation is used.

Simple calculation ($f = 0$), where D is the dataset of rank differences between datasets x and y:

$$rho = 1 - \frac{6 \sum D^2}{n(n^2 - 1)}$$

Complex calculation ($f = 1$), where $R(x_i)$ is the rank of the ith element of dataset x:

$$rho = \frac{\sum R(x_i)R(y_i) - \frac{n(n+1)^2}{4}}{\sqrt{\left[\sum R^2(x_i) - \frac{n(n+1)^2}{4}\right]\left[\sum R^2(y_i) - \frac{n(n+1)^2}{4}\right]}}$$

See *Nonparametric Statistical Inference*, p. 228. Many statistics books give only the simple form, so you may have to call SetSpearmanTie(0) to get answers that match your statistics book.

AskSpearmanTie (f)

AskSpearmanTie is the inverse of SetSpearmanTie. It returns $f = 0$ if the simple procedure will be followed for calculating Spearman's rho in the presence of ties, or 1 if the more complicated procedure will be used instead.

Line-Fits, Regressions, and ANOVA

This section describes least-squares linear, least absolute deviation linear, and polynomial curve-fitting routines, multiple linear regressions, and ANOVA calculations.

LSFit (x(), y(), resid(), ls())

LSFit computes a least-squares linear fit of the dependent variable *y*() to the independent variable *x*(). The *x*() and *y*() arrays must have the same sizes but needn't have the same upper and lower bounds; they're taken as paired data points (x_i, y_i) starting with the first element of each array. If either number in a pair is missing, the pair is not used.

Residuals are returned in *resid*(), which is resized as needed. If an *x* or *y* value is missing, the corresponding residual value will be missing.

The resulting statistics are returned in *ls*(), which can be indexed via the function names below. For instance, *ls*(*ls_r2*) is the r^2 value. See the "Subscript Functions" section for more information.

Least-Squares Statistics Subscript Functions

ls_n	number of non-missing *x* / *y* points
ls_slo	slope of fitted line
ls_int	intercept of fitted line
ls_xbar	mean of *x*()
ls_ybar	mean of *y*()
ls_ssx	sum of squares x
ls_sxy	sum of products xy
ls_ssy	sum of squares y
ls_sse	sum of squares error
ls_se	standard error
ls_ts	t-statistic for slope
ls_dfs	degrees of freedom for slope
ls_p	probability for slope's t-statistic
ls_r	Pearson's product-moment correlation coefficient
ls_r2	r^2 (coefficient of determination)
ls_z	Fisher's *z*-transform of *r*
ls_f	F-statistic (same as ts^2)

See the "Scatter and Residual Plots" section for an example of how to plot least-squares fits to data points.

If you give only one data point, no statistics can be computed except *xbar* and *ybar*. The other statistics will be set to the missing value. If *ssx, ssy,* or *se* = 0, then certain statistics can't be computed and will be set to the missing value instead. If you have disallowed missing values, you will get an error in these cases.

Use PolyFit if you need to assign weightings to the data points.

Exception:

110	Data arrays have different bounds in LsFit.
717	Can't use LsFit with SSX, SSY, or SE = zero.

PolyFit (x(), y(), w(), n, coeff(), resid(), var, cv(,))

PolyFit computes a least-squares polynomial fit of the dependent variable *y*() to the independent variable *x*(). The *x*() and *y*() arrays must have the same sizes but needn't have the same upper and lower bounds; they're taken as paired data points (x_i, y_i) starting with the first element of each array. If either number in a pair is missing, the pair is not used.

The *w*() array gives weights for each of the *x* / *y* data points. Pass a zero-size array if you want all points to have the same weighting. If *w*() is not zero-sized, its size must match those of *x*() and *y*().

You must also pass *n*, the degree of the polynomial. For instance, if you give $n = 2$, PolyFit will fit a second-degree polynomial through the data points.

The polynomial coefficients are returned in *coeff*(), which is redimensioned to *coeff*(0:*n*), so *coeff*(0) is the constant term, *coeff*(1) is the *x* term, *coeff*(2) is the x^2 term, and so forth. The residuals are returned in *resid*(); if either x_i or y_i is missing, the corresponding residual is also missing. The variance is returned in *var*, and the covariance matrix in *cv*(,).

See the "Scatter and Residual Plots" section for an example of how to plot polynomial fits to data points.

Exception:

110	Data arrays have different bounds in PolyFit.
741	Polynomial degree must be positive integer in Polyfit: n

MultiLSFit (d(,), y, corr(,), int, beta(), se(), t(), p(), ey(), res(), sres(), stres(), press(), hat(,), ra())

MultiLSFit computes a multiple linear regression based on least-squares linear fitting. It finds the intercept *int* and slopes *beta*() for the equation:

$$y = int + \beta_1 x_1 + \ldots + \beta_n x_n$$

You must supply a nonsingular array of data points *d*(,) in which each column is a dataset, and an index *y* that indicates which column is the dependent variable. This *y* must be an integer in the range Lbound(*d*,2) to Ubound(*d*,2). Naturally *d*(,) must have ≥ 2 columns, but there is no upper bound. It cannot have any missing values. MultiLsFit returns quite a few results:

corr(,)	correlation matrix
int	intercept
beta()	β values
se()	standard errors of *beta*()
t()	t-statistics for *beta*()
p()	probability of *t*()
ey()	estimated *y*() values ($int + \sum b_i x_i$)
res()	residuals
sres()	standardized residuals
stres()	Studentized residuals. Let *s* be *ra*(*ra_se*). Then *stres*(*i*) = *res*(*i*) / (*s* * Sqr(1–*hat*(*i*,*i*)).
press()	PRESS residuals: *res*(*i*) / (1 – *hat*(*i*,*i*))
hat()	HAT matrix = *d**Inv(Trn(*d*)**d*)*Trn(*d*) with *d*’s *y* column replaced by 1’s
ra()	miscellaneous Regression ANOVA statistics (see next page)

The *ra*() array contains a number of regression ANOVA statistics. You can access these array elements by using the function names below. For instance, *ra*(*ra_ar2*) is the adjusted R^2 value. See the “Subscript Functions” section for more information.

Regression ANOVA Statistics Subscript Functions

ra_ssm	sum squares mean
ra_ssr	sum squares regression
ra_sse	sum squares error (residual)
ra_sst	sum squares total
ra_dfm	degrees of freedom mean
ra_dfr	degrees of freedom regression
ra_dfe	degrees of freedom error (residual)
ra_dft	degrees of freedom total
ra_msr	mean square regression
ra_mse	mean square error (residual)
ra_se	standard error

ra_f	F-statistic
ra_p	Prob(*f*)
ra_r	multiple R
ra_r2	R-square
ra_ar2	adjusted R-square
ra_d	Durbin-Watson d statistic
ra_press	PRESS statistic: $\sum press(i)^2$

Exceptions:

756	Too few observations in MultiLSFit.
757	Dependent variable must be integer between i and j: y
758	Singular matrix in MultiLSFit.
759	MultiLSfit dataset can't contain missing values.

PrintMultiRegress (#n, d(,), y)

PrintMultiRegress computes a multiple linear regression based on least-squares linear fitting, and then prints a concise table of regression statistics.

Channel #*n* must refer to an open window or a text file. Pass #0 to print in the current window.

You must supply an array of data points *d*(,) in which each column is a dataset, and an index *y* that indicates which column of *d*(,) should be taken as the dependent variable. This *y* must be an integer in the range Lbound(*d*,2) to Ubound(*d*,2). Naturally *d*(,) must have at least two columns but there is no upper bound on the number of columns.

Exceptions:

756	Too few observations in MultiLSFit.
757	Dependent variable must be integer between i and j: y
758	Singular matrix in MultiLSFit.
759	MultiLSfit dataset can't contain missing values.
7004	Channel isn't open.
8501	Must be text file.

```
ANOVA       DF      Sum of Squares      Mean Square
Mean        1        158117.54450
Regression  2         10221.91570         5110.95785
Residual    17         1263.76980           74.33940

F ratio         68.75167         Std. Error           8.62203
Prob(F)       7.12598e-9         Multiple R            .94338
                                 R square              .88997
                                 Adj R square          .87703

Variable    DF    Estimate        Std. Error          T ratio          Prob(T)

Intercept    1        1.67713
X2           1         .07856          .01880         4.17861          .00063
X3           1        1.79840          .61413         2.92838          .00938

Durbin-Watson D Statistic:        .78355
```

Figure 43.30: Output of the MULTIREG program.

```
   PRESS Residual Analysis of Shingle Sales

          y       Est.y       Resid      PRESS
  1     79.30    77.726       1.574      2.019
  2    200.10   207.935      -7.835     -8.552
  3    163.20   161.667       1.533      2.185
  4    200.10   212.363     -12.263    -13.869
  5    146.00   147.259      -1.259     -1.485
  6    177.70   175.944       1.756      2.762
  7     30.90    29.608       1.292      1.839
  8    291.90   278.327      13.573     18.105
  9    160.00   161.535      -1.535     -1.736
 10    339.40   339.003        .397       .655
 11    159.60   158.957        .643       .760
 12     86.30    79.576       6.724      8.093
 13    237.50   234.638       2.862      3.465
 14    107.20   106.279        .921      1.052
 15    155.00   163.386      -8.386     -9.177

 PRESS Statistic:    782.1896
```

Figure 43.31: Output of the REGPRESS program.

Advanced Examples in Multiple Regression

Your diskette contains three sample programs that illustrate advanced techniques in multiple regression. You should skip this section unless you are truly zealous and well-versed in regressions.

The REGPRESS program analyzes a model of asphalt shingle sales. It is taken from Myers' *Classical and Modern Regression with Analysis*, pp. 108-111, and illustrates how PRESS residuals and the PRESS statistic can be used to evaluate a 2-variable model. Columns display actual vs. predicted *y* values, simple residuals, and corresponding PRESS residuals. See Figure 43.31.

The REGHAT program analyzes a model of U. S. Navy Bachelor Officers' Quarters manpower and workload. It too is taken from Myers, pp. 144-147. As he notes, careful analysis of the Studentized residuals leads one to believe that the model is strained for $y > 2000$, perhaps because of nonhomogeneous variance. Data point 23 is particularly troublesome as its Studentized residual is large and its HAT diagonal value indicates that the point is remote from the data center.

Columns display actual vs. predicted *y* values, simple and Studentized residuals, and the corresponding HAT matrix diagonal values.

U.S. Navy BOQ Manpower/Workload Analysis

	y	est.y	e	Stu.e	hat
1	180.23	209.985	-29.755	-.076	.2573
2	182.61	213.796	-31.186	-.075	.1609
3	164.38	360.486	-196.106	-.470	.1614
4	284.55	360.106	-75.556	-.181	.1631
5	199.92	380.703	-180.783	-.430	.1475
6	267.38	510.373	-242.993	-.582	.1589
7	999.09	685.167	313.923	.763	.1829
8	1103.24	1279.299	-176.059	-.483	.3591
9	944.21	815.466	128.744	.334	.2808
10	931.84	891.846	39.994	.094	.1295
11	2268.06	1632.137	635.923	1.493	.1241
12	1489.50	1305.177	184.323	.453	.2024
13	1891.70	1973.416	-81.716	-.187	.0802
14	1387.82	1397.786	-9.966	-.023	.0969
15	3559.92	4225.131	-665.211	-2.197	.5576
16	3115.29	3134.895	-19.605	-.056	.4024
17	2227.76	2698.738	-470.978	-1.302	.3682
18	4804.24	4385.778	418.462	1.236	.4465
19	2628.32	2190.326	437.994	1.007	.0868
20	1880.84	2750.910	-870.070	-2.401	.3663
21	3036.63	2210.134	826.496	1.883	.0704
22	5539.98	5863.874	-323.894	-1.536	.7854
23	3534.49	3694.766	-160.276	-3.278	.9885
24	8266.77	7853.505	413.265	2.580	.8762
25	1845.89	1710.861	135.029	.441	.5467

Figure 43.32: Output of the REGHAT program.

The REGHAT2 program shows the same results as a scatter plot of Studentized residuals vs. predicted man-hours. This figure is directly drawn from Myers, p. 147, but also draws data points "distant" from the data center as solid blocks for emphasis. Distance is measured by a point's HAT value — *not* by the coordinate system that this graph uses — and that's why one of the "distant" points happens to be near the "center" of this graph.

The lowest point is data point 23 as shown in the table on the facing page. Note that it has the most extreme Studentized residual *and* is distant from the data center in terms of the HAT measure. This strongly suggests an unusual departure from zero.

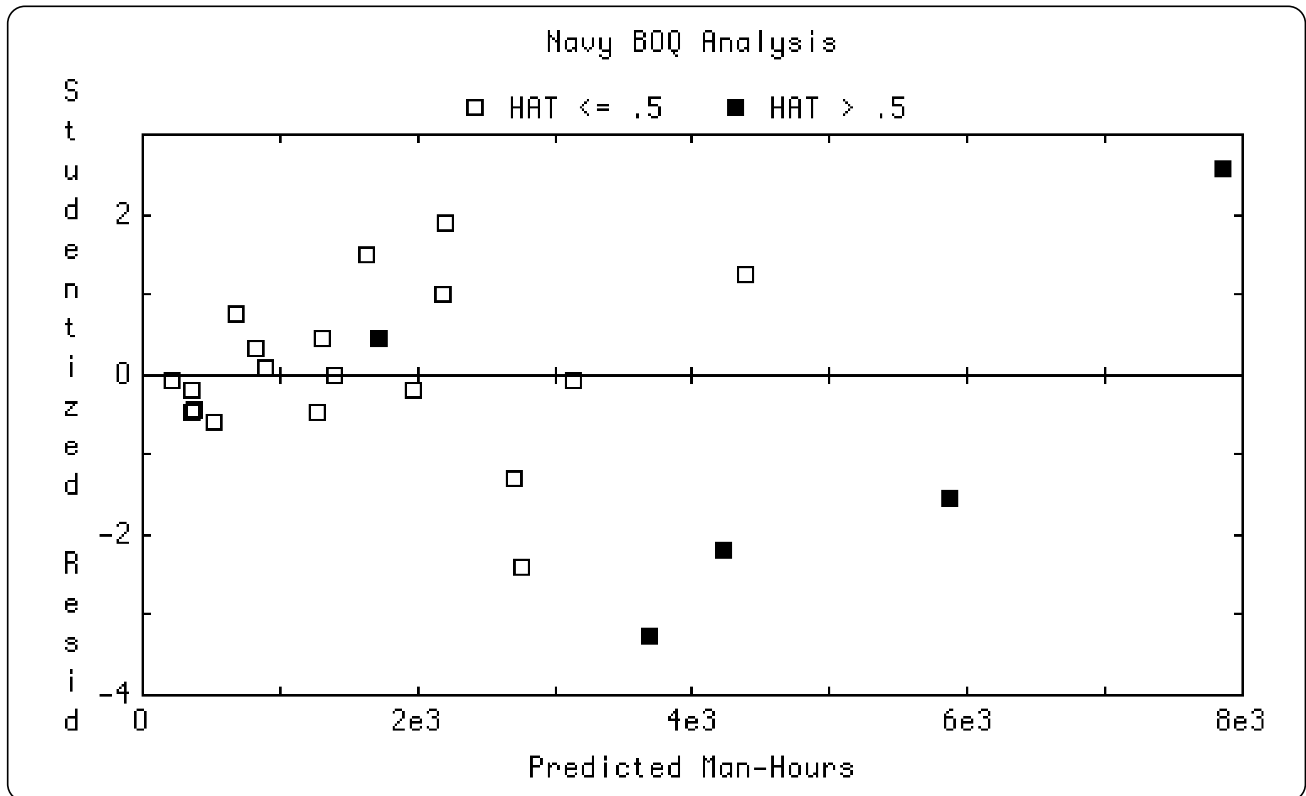

Figure 43.33: Output of the REGHAT2 program.

Anova (d(,), an())

Anova computes a one-factor analysis of variance from the data points *d*(,) and returns statistics in the *an*() vector. Each column of *d*(,) is taken as one dataset, so "within" refers to columns and "between" refers to rows. You can access the *an*() array elements by using the function names below. For instance, *an*(*an_f*) is the F-statistic. See the "Subscript Functions" section for more information.

If you want to get fancier, try the MultiLSFit routine. See also the Friedman routine for a nonparametric two-way ANOVA.

ANOVA Statistics Subscript Functions

an_msw	mean square within
an_msb	mean square between
an_ssw	sum of squares within
an_ssb	sum of squares between
an_sst	sum of squares total
an_dfw	degrees of freedom within
an_dfb	degrees of freedom between
an_dft	degrees of freedom total
an_f	F-statistic
an_p	Prob(F)

PrintAnova (#n, d(,))

PrintAnova computes a one-factor analysis of variance from the data points *d*(,) and prints an ANOVA table from the results. Each column of *d*(,) is taken as one dataset, so “within” refers to columns and “between” refers to rows. If you want to get fancier, try the PrintMultiRegress routine.

Channel #*n* must already be open. It can refer to a window or a text file. Pass #0 to print in the current window.

Exceptions:

7004	Channel isn’t open.
8501	Must be text file.

MedFit (x(), y(), slope, inter, mad,resid())

MedFit computes a “median” (least absolute deviation) linear fit of the dependent variable *y*() to the independent variable *x*(). The *x*() and *y*() arrays must have the same sizes but needn’t have the same upper and lower bounds; they’re taken as paired data points (x_i, y_i) starting with the first element of each array. If either number in a pair is missing, the pair is not used.

MedFit is like LSFit except that it uses a more “robust” technique so that outliers have less effect on the fitted line. It is, however, significantly slower than LSFit.

MedFit returns the slope and intercept of the fitted line in *slope* and *inter*, the mean absolute deviation *mad*, and the residuals in *resid*().

Use SetLineFit to switch to using least-absolute-deviation fits instead of least-squares fits in scatter and regression plots.

DataToNormResid (data(), from, to, step, resid())

Compute the "residuals" derived by subtracting a normal curve from the dataset's histogram. This gives a quick check to see if the data appears normally distributed. This routine is used by PlotNormFit to calculate its residuals; see its description for a picture of the results.

You must pass a *data*() array, and *from, to,* and *step* to define the histogram grouping. For instance, values of 1, 10, and .5 create a histogram with centerpoints at 1, 1.5, ..., 9.5, 10. As usual, low and high values are collected one step beyond either end of the interval; in this example, all low values are collected in .5 and all high values into 10.5. See the "Frequency Distributions" section for more information about *from, to,* and *step*.

Once the histogram has been created, it is subtracted from the normal curve defined by the data's mean and variance. If histograms are being plotted by count, these absolute differences are directly plotted as the residuals. But if not, the curve and histogram are first normalized so the area equals 1. Then the residuals are computed. Finally, the residuals are divided by the normal curve's highest value to give a relative residual. This makes it easier to spot variations between curves with different variances.

Data() and *resid*() may have any bounds. The lower bound of *resid*() is left unchanged, but the upper bound is adjusted to hold the residuals.

FreqToNormResid (freq(,), resid(,))

This is just like DataToNormResid except that the input and output arrays *freq*(,) and *resid*(,) are frequency distributions instead of raw data.

Such frequency distributions can be created by DataToFreq, etc. See the "Frequency Distributions" section for more information.

Freq(,) and *resid*(,) may have any bounds. The lower bounds of *resid*(,) are left unchanged, but the upper bounds are adjusted to hold the residuals.

Scatter and Residual Plots

This section describes how to get scatter plots. In general, you simply pass your *x*/*y* data to the appropriate routine, along with information about point and line styles and the color scheme.

If you wish, you can have least-squares or "median" (least absolute deviation) fitted lines, confidence bands, and/or polynomial fits added automatically to all scatter plots. See the end of this section for details.

Logarithmic scaling for either the X or Y axis, or both, is easy. Just call SetGraphType before drawing your graph:

```
call SetGraphType("logx")
call SetGraphType("logy")
call SetGraphType("logxy")
```

See the "Advanced Graph Controls" section for more information.

PlotScat (x(), y(), ps, ls, col$)

Draw a scatter plot of the *x*/*y* pairs. These two arrays must have the same bounds; if either coordinate in a point is missing, that point is ignored. By default, PlotScat automatically picks coordinates that show all the data. You can, however, override it by calling SetXscale and SetYscale.

If *x()* is the "null" vector, (having *no* elements), then the x-coordinates will be 1,2, ….

Points are drawn in the *ps* point style connected by lines drawn in the *ls* line style. If *ps* = 0, no points are shown. If *ls* = 0, no connecting lines are shown. There are currently 13 point styles and 4 line styles to choose from; see the "Making Graphs" section for more help.

PlotScat draws the first point given in *x* and *y*, then the next point, and so forth. This might not be in left-to-right order, so if you connect unordered points, the result will look scrambled. You can use SortPoints to order the points before graphing them.

If you call SetLS, SetConfBand, or SetPolyFit before calling PlotScat, the scatter plot will be drawn with an automatic linear fit, confidence band, or polynomial fit.

The color scheme *col$* is treated as usual. A string such as "red green yellow" gives a red title, green frame, and yellow data. The data color is used for data points, fitted line, confidence bands, and polynomial fitting (if requested). See the end of this section for more about least-squares, confidence bands, and polynomial fitting, and the "Making Graphs" section for help with color schemes.

AddScatPlot (x(), y(), ps, ls, col$)

Add a scatter plot of x/y pairs to the existing graph. These arrays must have the same bounds; if either coordinate in a point is missing, that point is ignored. If $ps \neq 0$, each point is drawn in that style. If $ls \neq 0$, the points are connected in order by lines in that style. AddPlotScat uses the existing coordinates, so some or all of your added data may not be visible on the graph.

Title and frame colors are ignored in the color scheme *col$*. A string such as “red green yellow” gives yellow data, as does a string such as “yellow”. The data color is used for data points, fitted line, confidence bands, and polynomial fitting (if requested). See the end of this section for fitted lines, confidence bands, and polynomial fitting, or “Making Graphs” for help with color schemes and point and line styles.

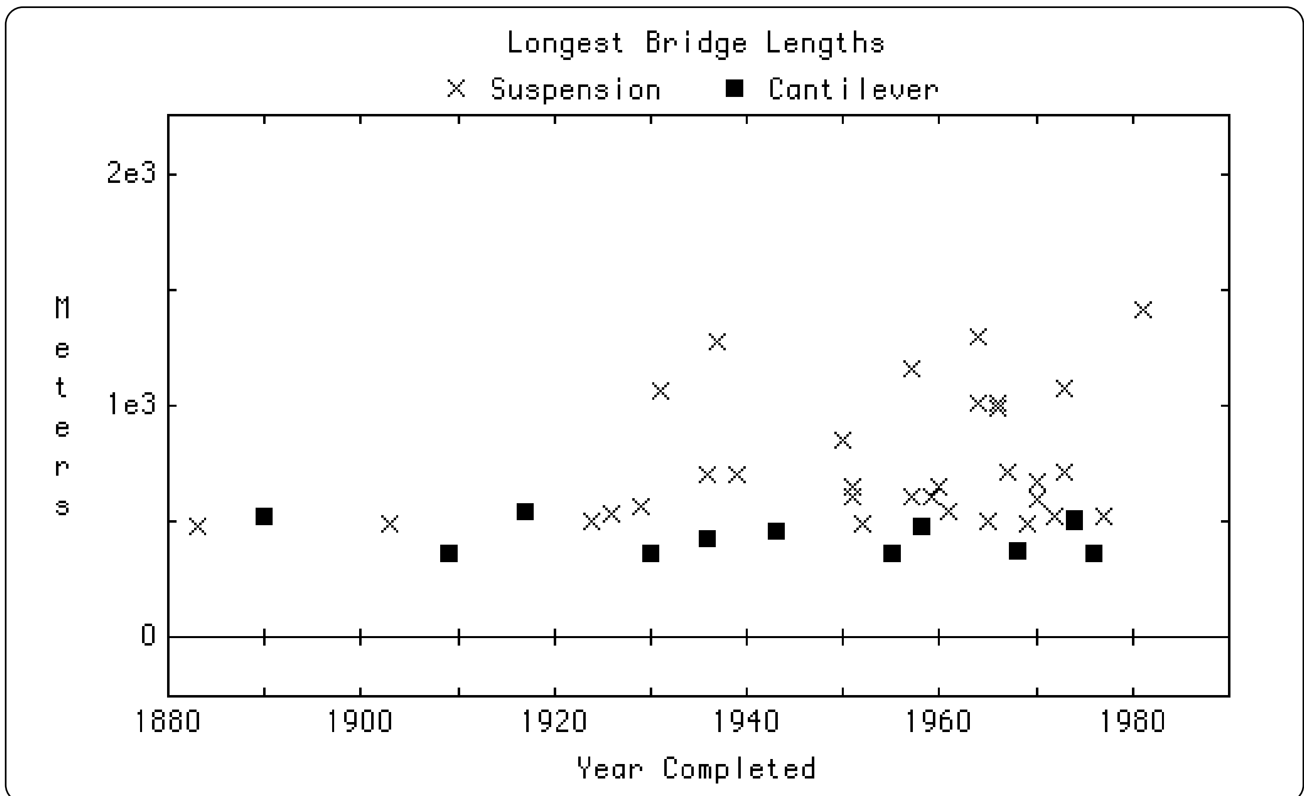

Figure 43.34: Output of the SCAT2 program.

PlotManyScat (x(,), y(,), connect, legend$, col$)

PlotManyScat plots multiple sets of data points in one graph. Each row of the x and y arrays contains the coordinates for one set of data points, so x and y must have the same sizes. As illustration, **dim x(3,15)** creates an x array with 3 datasets of 15 points each. You can **mat read** these coordinates from 3 data statements, each of which holds 15 numbers.

The datasets' point styles are taken in order from the GraphPoint set – plus, asterisk, circle, and so on – and cycle through the set if you plot more than 12 datasets in one graph. (Style #1 isn't used since it's hard to see.)

If *connect* is nonzero, each data point is connected to its neighbors by a straight line. You can use SortPoints2 to make sure your data points are in order. If you've used SetLS to turn on fitted lines, each dataset gets its own line. Confidence bands and connecting or fitted lines have the same colors and line styles. To distinguish them, use PlotScat and AddPlotScat to control colors and line styles.

Size(*x*,1) gives the number of datasets. Each is identified by a label from *legends$*() just below the graph's title, so Size(*x*,1) must equal Size(*legends$*).

The *col$* string gives the color scheme; thus "red yellow green blue" gives a red title, yellow frame, and green and blue datasets. If you give more datasets than data colors, PlotManyScat uses different line styles to distinguish the connecting or fitted lines. First it draws solid lines in the colors you've chosen. Then it switches to dashed, dotted, and then dash-dotted lines. So if you graph five datasets with connecting lines in the color scheme above, you get (in order): solid green, solid blue, dashed green, dashed blue, dotted green. See the "Making Graphs" section for more help.

Compare the output of Figures 43.04 and 43.05 to see the correlation differences.

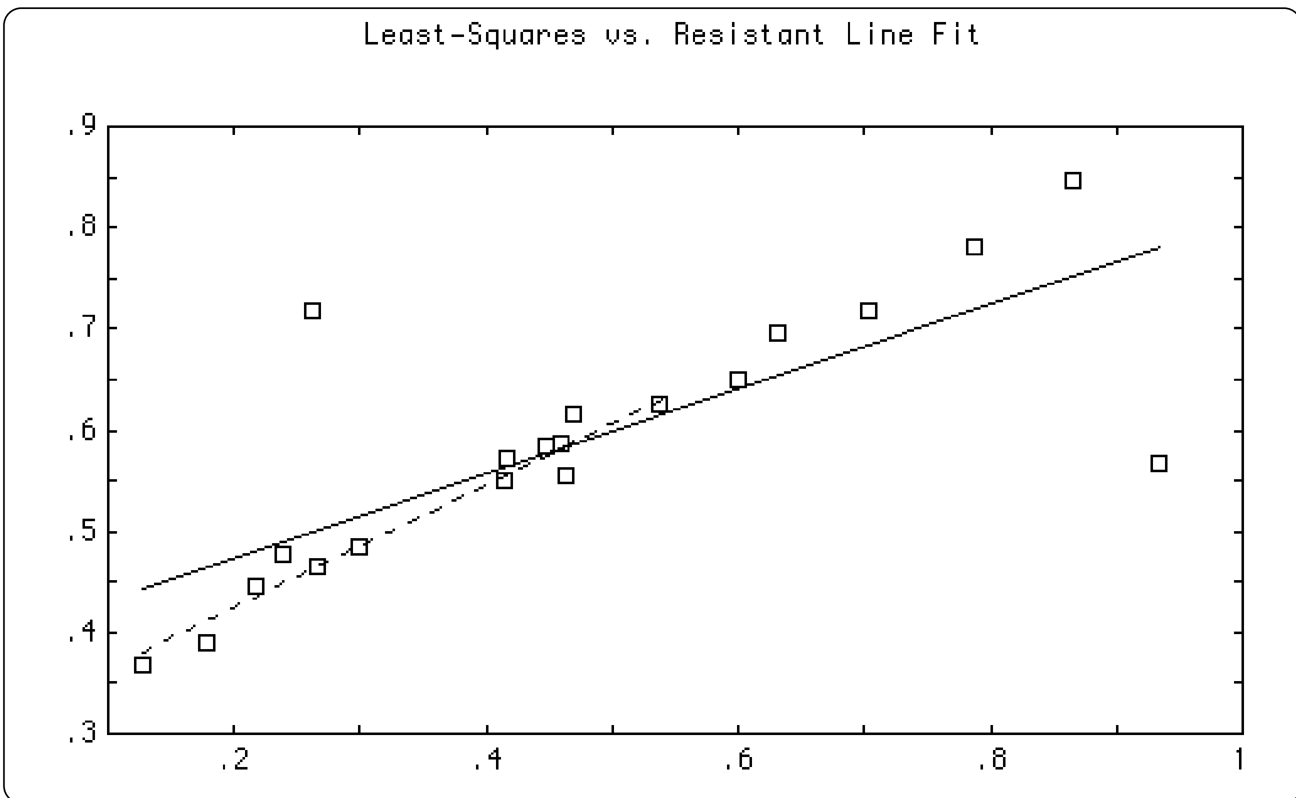

Figure 43.35: Output of the LINEFIT program.

AddLinFitPlot (x(), y(), ls, col$)

Draw a least-squares or least-absolute-deviation line for the x/y data on top of the existing graph. These two arrays must have the same bounds; if either coordinate in a point is missing, that point is ignored. The least-squares line for non-missing data points is drawn in line style *ls*. The points themselves are not displayed.

AddLinFitPlot uses the existing coordinates, so some or all of your added line may not be visible on the graph. Use SetLineType to switch between least-squares and least-absolute-deviation line fits.

Title and frame colors are ignored in the color scheme *col$*. A string such as “red green yellow” gives a yellow line. See the “Making Graphs” section for information about color schemes and a list of line styles.

AddPolyPlot (coeff(), ls, col$)

Draw a least-squares polynomial with coefficients *coeff*() on top of the existing graph in line style *ls*. AddPolyPlot uses the existing coordinates, so some or all of your added line may not be visible on the graph.

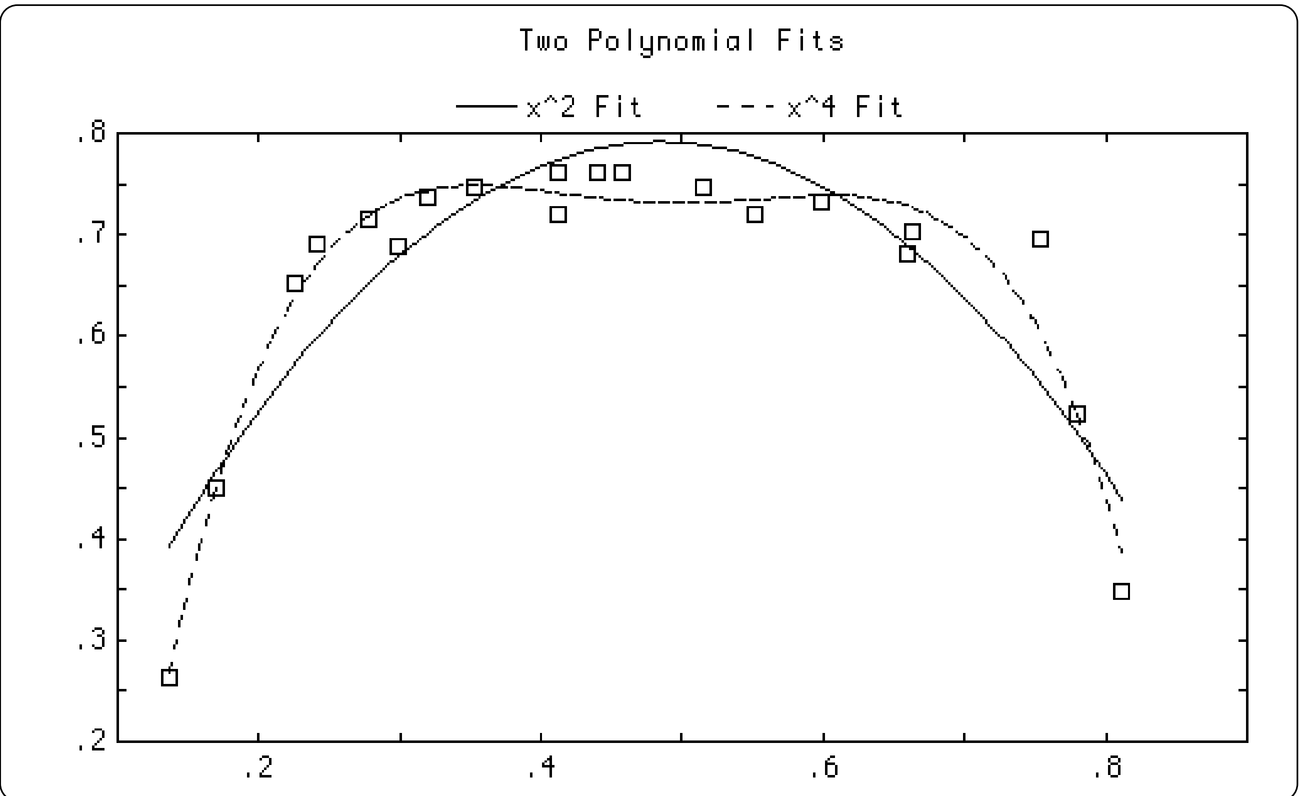

Figure 43.36: Output of the POLYFIT program.

Coeff(0) is the constant term, *coeff*(1) is the x term, *coeff*(2) is the x^2 term, etc. The array size determines the polynomial degree: all *coeff*() elements are used as terms. You can get *coeff*() by PolyFit.

Title and frame colors are ignored in the color scheme *col$*. A string such as “red green yellow” gives a yellow line. See the “Making Graphs” section for information about color schemes, and a list of line styles.

PlotResid (x(), y(), ps, ls, col$)

Plot the residuals of *y*() against *x*(), where the residuals are computed against the least-squares or median-fit line for the x/y data points. These arrays must have the same bounds; if either coordinate in a point is missing, that point is ignored. If $ps \neq 0$, each point is drawn in that point style. If $ls \neq 0$, the points are connected in order by a line in that style.

By default, PlotResid picks coordinates that show all the data. To override it, use SetXrange and SetYrange.

The color scheme *col$* is treated as usual. A string such as “red green yellow” gives a red title, green frame, and yellow data.

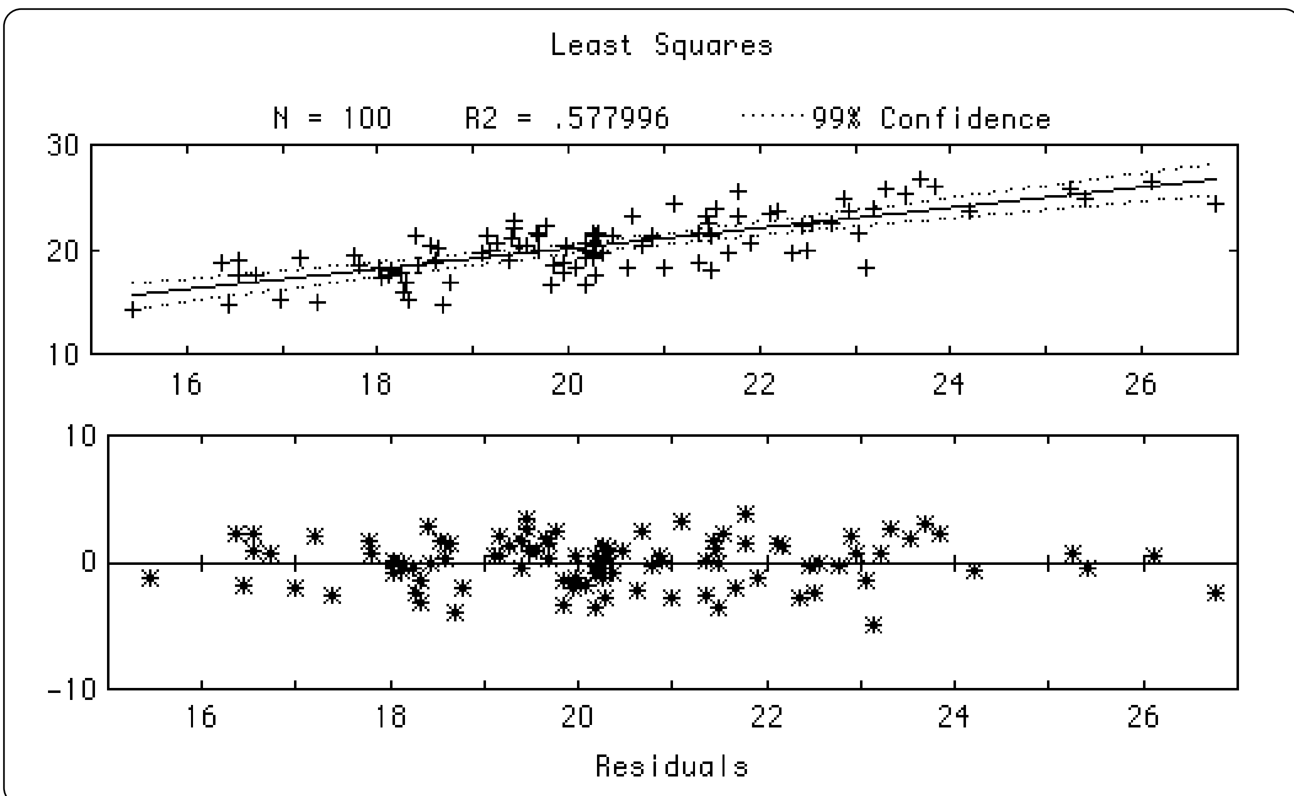

Figure 43.37: Output of the RESID program.

See the end of this section for more about linear fits, confidence bands, and polynomial fitting, and the “Making Graphs” section for information about color schemes and a list of point and line styles.

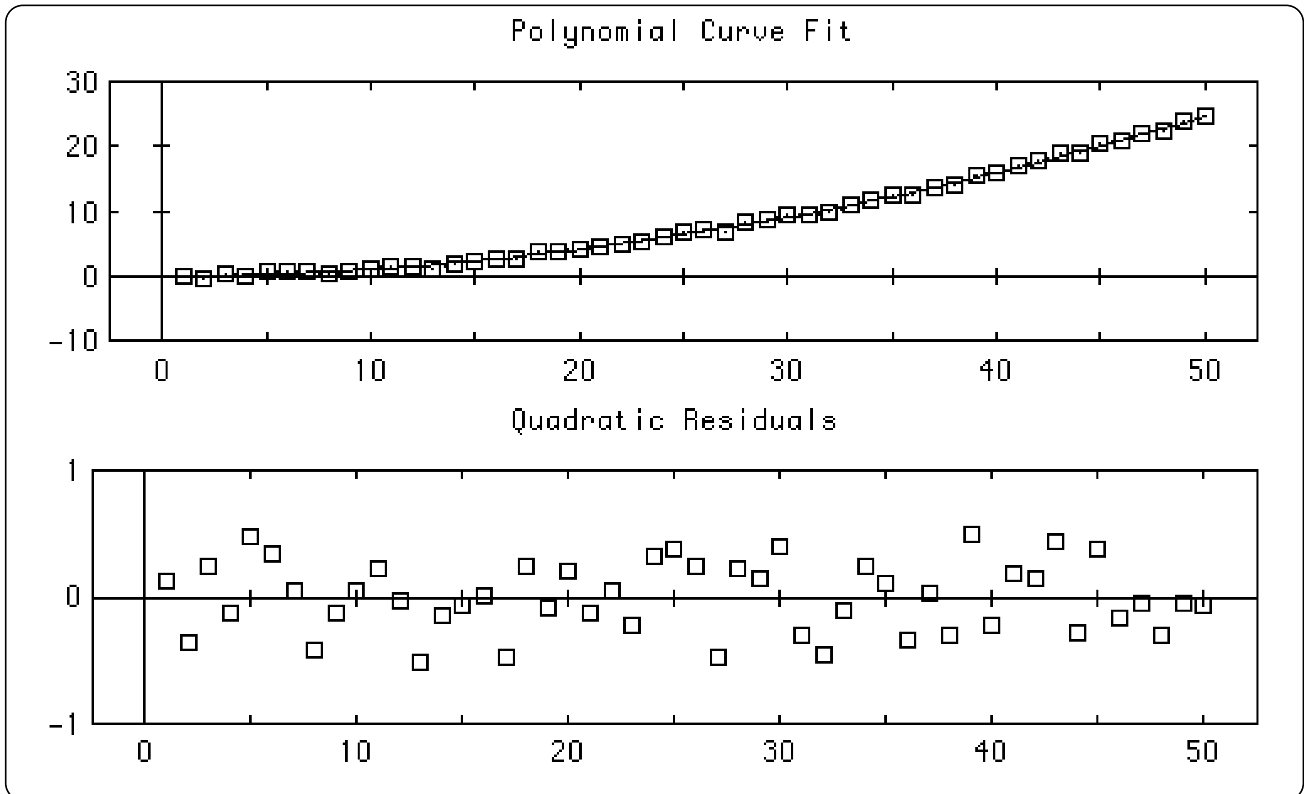

Figure 43.38: Output of the RESID2 program.

PlotPolyResid (x(), y(), n, ps, ls, col$)

Plot the residuals of $y()$ against $x()$, where the residuals are computed against the least-squares polynomial of degree n for the x/y data points. These arrays must have the same bounds; if either coordinate in a point is missing, that point is ignored. If $ps \neq 0$, each point is drawn in that point style. If $ls \neq 0$, the points are connected in order by a line in that style.

By default, PlotResid automatically picks coordinates that show all the data. You can, however, override its choices by using SetXrange and SetYrange.

The color scheme *col$* is treated as usual. A string such as “red green yellow” gives a red title, green frame, and yellow data.

Exception:

741 Polynomial degree must be positive integer in PlotPolyResid: n

PlotNormFit (data(), from, to, step, ps, ls, col$)

Plot the "residuals" derived by subtracting a normal curve from the dataset's histogram. This gives a quick check to see if the data appears normally distributed.

You must pass a *data*() array and *from, to,* and *step* to define the histogram grouping. For instance, values of 1, 10, and .5 create a histogram with centerpoints at 1, 1.5, ..., 9.5, 10. As usual, low and high values are collected one step beyond either end of the interval; in this example, all low values are collected in .5 and all high values into 10.5. See the "Frequency Distributions" section for more information about *from, to,* and *step*.

Once the histogram has been created, it is subtracted from the normal curve defined by the data's mean and variance. If histograms are being graphed by count, these absolute differences are directly plotted as the residuals.

But if not, the curve and histogram are first normalized so the area equals 1. Then the residuals are computed. Finally, the residuals are divided by the normal curve's highest value to give a relative residual. This makes it easier to spot variations between curves with different variances.

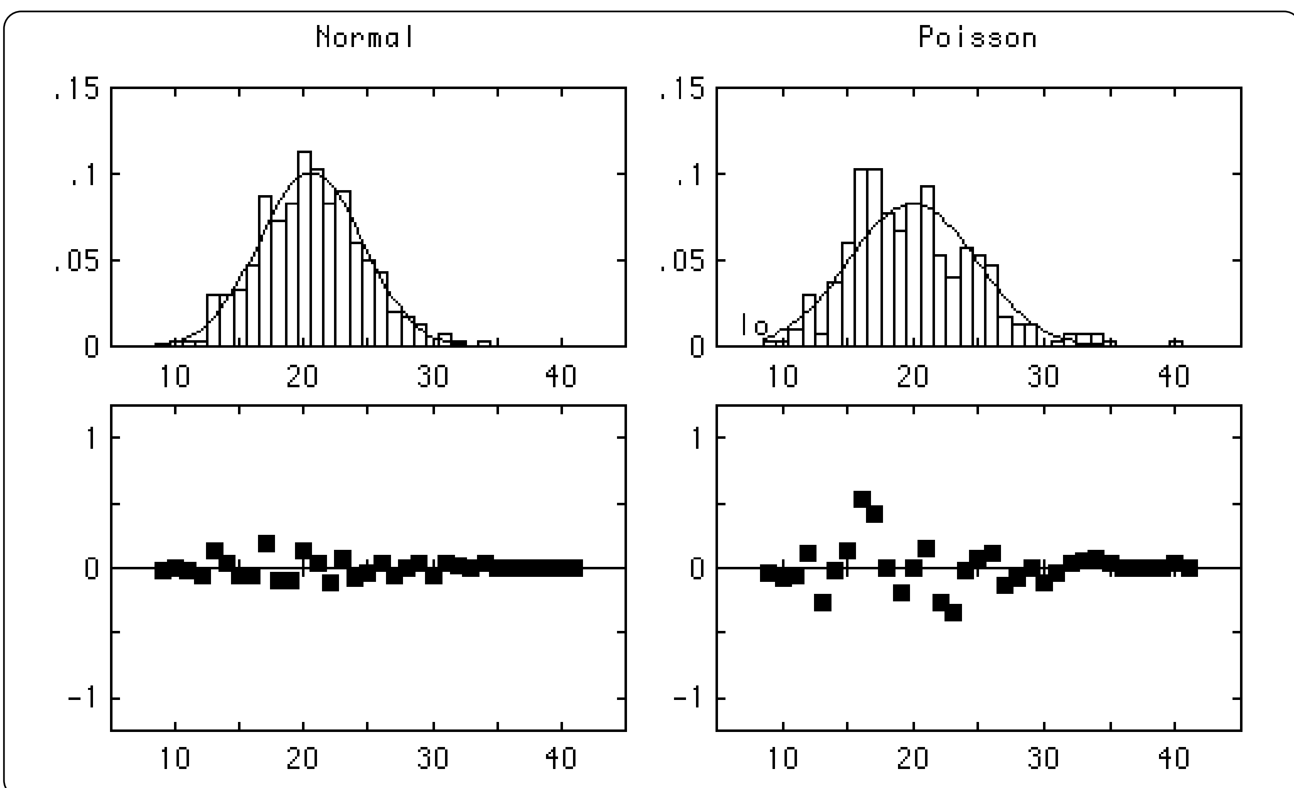

Figure 43.39: Output of the NORMFIT program.

The residual points are drawn in the *ps* point style connected by lines drawn in the *ls* line style. If *ps* = 0, no points are shown. If *ls* = 0, no connecting lines are shown. There

are currently 13 point styles and 4 line styles to choose from; see the “Making Graphs” section for information.

As usual, *col$* gives the color scheme; for instance “red green blue” gives a red title, green frame, and blue data.

Observation Plots

This section describes how to plot data with multiple observations of each item. For example, you may have run an experiment three times and hence have three observations of each data point.

X Coordinates

The X coordinates for observation plots are given by a series of *strings* rather than numbers. Thus you can label your horizontal axis with marks such as “1:1” and “2:1”, or “Jan” and “Feb”, and so forth. Of course, you can also get standard numbers along the X axis by passing “1”, “2”, or whatever.

If your X axis labels are all numbers, the Toolkit will use those numbers to define the coordinates of the observation plot. But if they aren’t, it will assign the first mark an X coordinate of 1, the second one 2, and so forth.

WARNING: If you supply non-numeric labels for the X axis, legends that include line slope, intercepts, and so forth, will be based on the implied X coordinates of 1, 2, 3, ... This may not be what you want! In addition, log-X plots will look bizarre.

Remember, if you supply numeric X axis labels, the legend’s slope, intercept, etc., will in fact match the graph, and log-X plots will work correctly.

PlotObs (data(,), xlabel$(), meanps, ls, col$)

Plot a dataset *data*(*item,obs*) where each data item has multiple observations. Each row contains observations of the same data point.

By default, PlotObs doesn’t plot individual observation points; it plots the *mean* of each set of observations, and an error bar that extends 1 standard deviation above and below the mean. Call SetErrorBeam to change the shape of the error bar, SetObsSD to control its length or remove it altogether, or SetObsSE to draw bars of standard error rather than standard deviation.

Each mean observation is drawn in point style *meanps*; pass *meanps* = 0 to omit plotting the mean points. If $ls \neq 0$, the mean points are connected by a line drawn with line style *ls*. See the “Making Graphs” section for information on these styles.

You can also ask for least-squares lines, polynomial fits, and confidence bands to be drawn through the means of each observation. See SetLS, SetPolyFit, and SetConfBand for more information.

Each set of observations is labelled, along the X axis, by a corresponding label from the *xlabel$*() array. Therefore, Size(*data*,1) must equal Size(*xlabel$*). You may use whatever labels you like in the *xlabel$()* array so long as they all fit on the screen. The color scheme *col$* works as usual; for instance “red green blue” draws a red title, green frame, and blue data.

Call SetDataStyle(*ps*) to plot the individual data points in the indicated point style.

AddObsPlot (data(,), meanps, ls, col$)

AddObsPlot is just like PlotObs except that it simply draws the data on top of an existing graph. Because it does not adjust the graph’s scale, the new data may lie outside the graph frame and hence be invisible. To plot multiple sets of data and make sure they all show, use PlotManyObs.

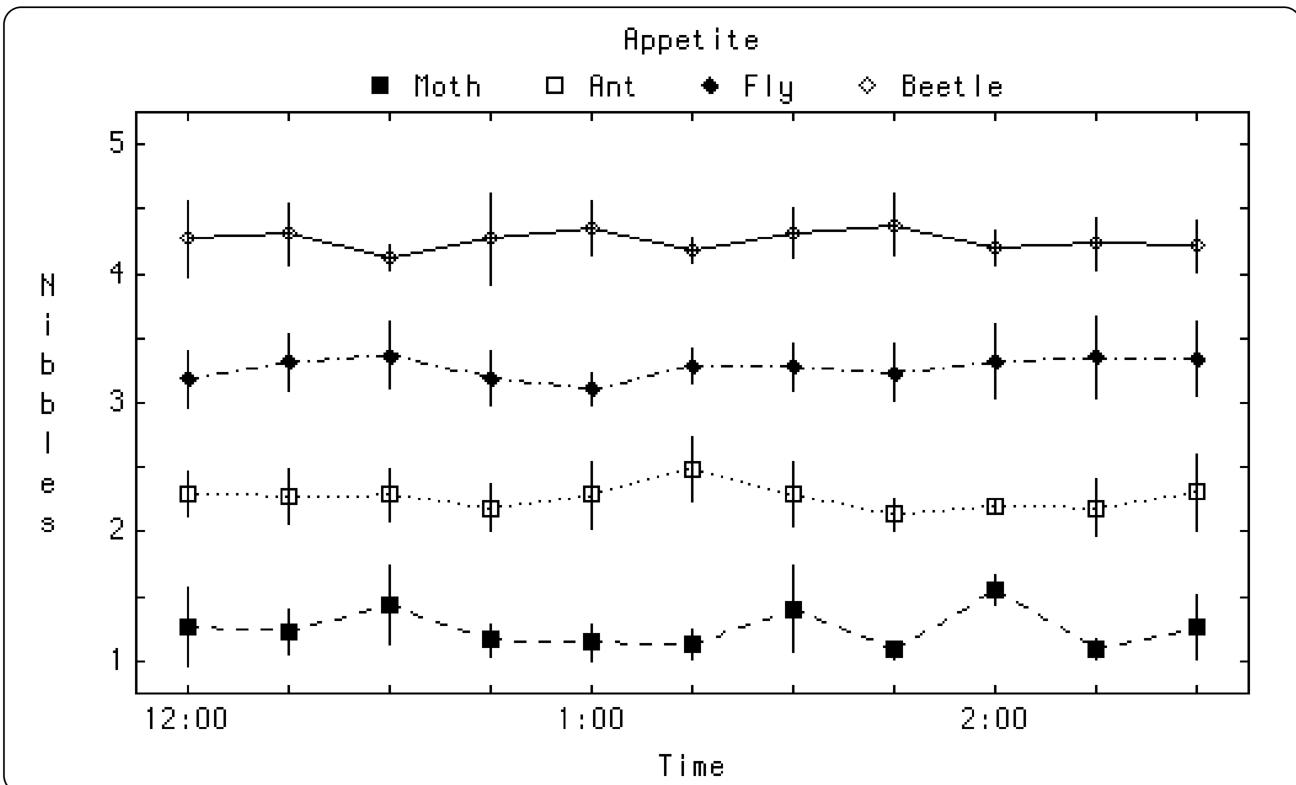

Figure 43.40: Output of the MANYOBS program.

PlotManyObs (data(,,), legend$(), xlabel$(), col$)

Plot multiple observation datasets on one graph. The *data*(,,) array contains all datasets: *data*(*set,item,obs*). These datasets needn't have the same sizes; simply pad short datasets with the missing value.

First the graph is scaled so all data points are visible. Then the datasets are plotted in turn, as if by PlotObs. Use SetObsSD or SetObsSE to control the error bars. SetDataStyle works as follows: if you've set a nonzero value, data points are plotted using this sequence of point styles: circle, X, up triangle, down triangle, plus. If you have more than 5 datasets, styles are reused.

Mean points are always plotted. Their symbols are: solid box, box, solid diamond, diamond, solid up triangle. Symbols are reused if need be. Lines that connect a dataset's mean observations area are also plotted; their styles cycle through all available styles. You can also ask for linear fits, polynomial fits, and confidence bands to be drawn through the means of each observation. See SetLS, SetPolyFit, and SetConfBand for more information.

If *col$* = "red green blue brown" then dataset 1 is plotted in blue, 2 in brown, 3 in blue, and so forth.

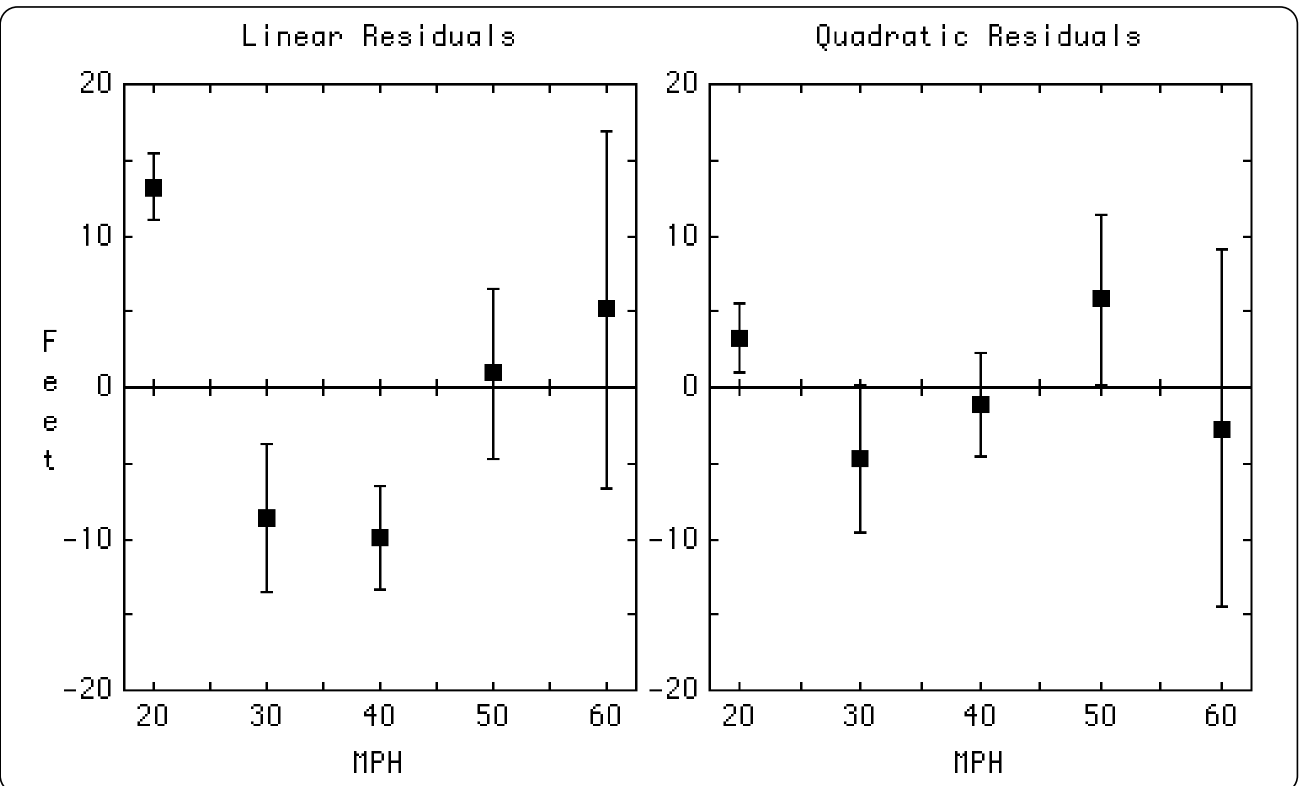

Figure 43.41: Output of the OBSRESID program.

The X axis is labelled by the *xlabel$()* items, so Size(*xlabel$*) must equal Size(*data*,2). Each dataset is identified by a legend; pass legend items in *legend$()*. Hence Size(*legend$*) must equal Size(*data*,1).

PlotObsResid (data(,), xlabel$(), meanps, ls, col$)

Plot residuals of the *data*(,) set from a linear fit through the data points. Parameters are as in PlotObs.

PlotPolyObsResid (data(,), xlabel$(), n, meanps, ls, col$)

Plot residuals of *data*(,) from a least-square polynomial of degree *n* that fits the data points. Aside from the polynomial degree *n*, parameters are as in PlotObs.

Exception:

741 Polynomial degree must be positive integer in PlotPolyObsResid: n

ObsResid (data(,), xlabel$(), n, resid(,))

Compute residuals *resid*(,) of the *data*(,) set from a least-squares polynomial of degree *n* that fits the data points. If *xlabel$*() contains numbers, these will be used as the X coordinates for the polynomial fit; otherwise the first data points will have X coordinate 1, the second points will have 2, and so forth.

Exception:

741 Polynomial degree must be positive integer in ObsResid: n

Box Plots of Observations

You can also see box-and-whisker plots of your observed data, instead of graphs with error bars. These box plots are appropriate when you are not sure that your data is normally distributed. Just call SetObsBox before drawing the observation plot.

Program OBSBOX, on your diskette, shows a box-observation plot.

Fitted lines are based on the *medians* of box plots, rather than means. (Remember that the median is marked with a “+” in box plots.)

If you ask to see the raw data points (with SetDataStyle), they are displayed slightly to the right of the box plot as shown above. The box plot itself occupies the correct horizontal position.

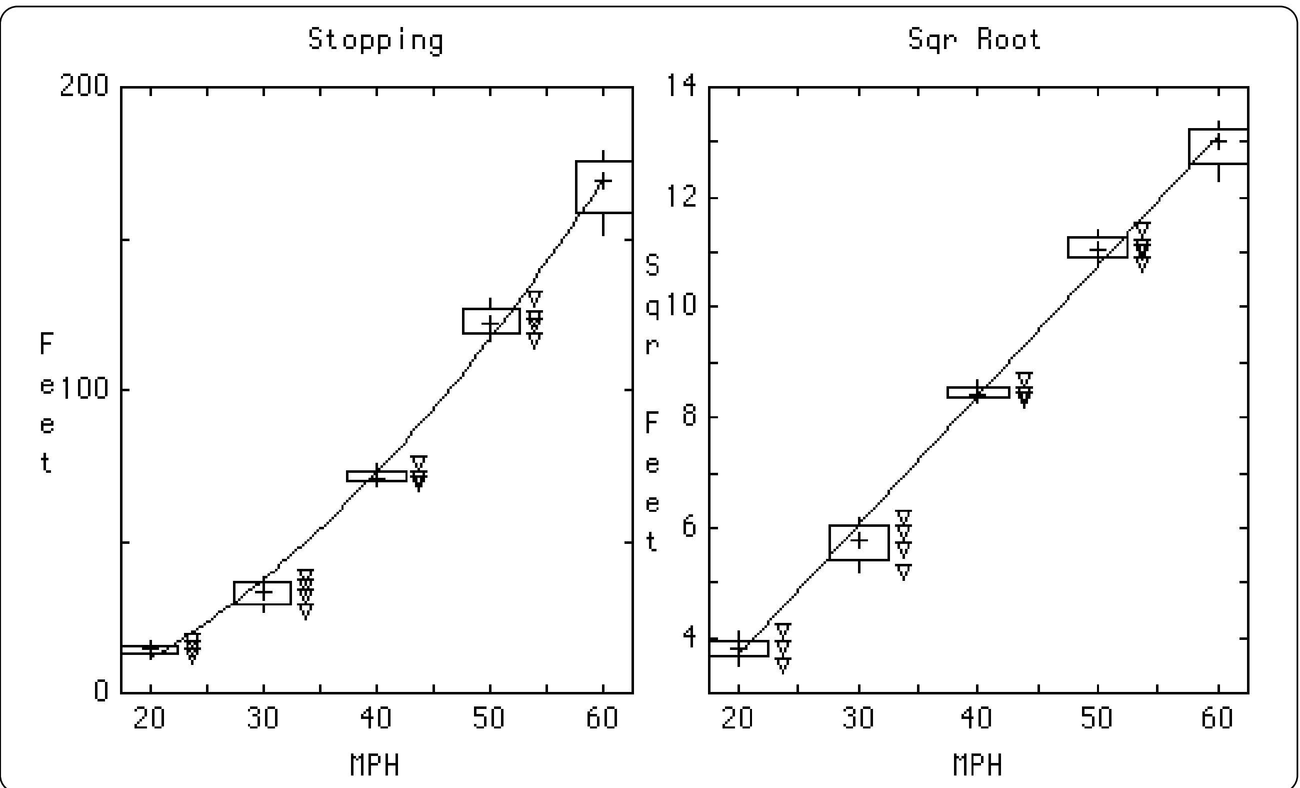

Figure 43.42: Output of the OBSBOX program.

SetLS (f)

Add least-squares or median lines, automatically, to every subsequent scatter or observation plot. Use non-zero *f* to turn on the least-squares lines, 0 to turn them off. The line will be drawn in line style 1 (solid line).

Note that this routine is also used by the *Scientific Graphics Toolkit*, so if you use both toolkits together, you only need to call this one routine. By default, least-squares lines are turned off.

AskLS (f)

The opposite of SetLS. Returns 1 for automatic least-squares lines, 0 otherwise.

SetConfBand (ci, ls)

Add confidence bands to every subsequent scatter or observation plot. These bands show the confidence interval for the least-squares line, not for the data!

The bands are added at the *ci* confident interval in the *ls* line style. For example *ci* = .95 and *ls* = 3 gives dotted lines at the 95% confidence interval. Use *ls* = 0 to turn off the confidence bands. By default, confidence bands are off.

AskConfBand (ci, ls)

The opposite of SetConfBand. Returns the current confidence interval for automatic confidence bands, along with its line style.

SetLineFit (type$)

Set the type of linear fits used in scatter and regression plots. Pass *type$* = "LS" for least-squares fits, or "MEDIAN" for median (least-absolute-deviation) fits.

Note that confidence bands, r^2 values, etc., cannot be computed for median fits.

AskLineFit (type$)

The opposite of SetLineFit. Returns the current linear fitting type as "LS" or "MEDIAN". The default is "LS".

SetPolyFit (n)

Add a polynomial curve fit of degree n, automatically, to every subsequent scatter or observation plot. For instance, $n = 2$ fits a parabola to subsequent scatter plots. Note that n must be a non-negative integer. The polynomial is drawn in line style 1 (solid line). Call SetPolyFit(0) to turn off the polynomial. By default, polynomial fitting is turned off.

Exception:

741	Polynomial degree must be positive integer in SetPolyFit: n

AskPolyFit (n)

The opposite of SetPolyFit. Returns $n > 0$ for automatic polynomial fitting of degree n, 0 otherwise.

AskLineStats (ls())

Return the statistics computed for the last least-squares line fitted through data points, either by calling LSFit or by adding a least-squares line to a graph. This array has the same format as that returned by LSFit.

AskPolyCoeff (coeff())

Return the coefficients for the last polynomial fitted through data points, either by calling PolyFit or by adding a least-squares polynomial fit to a graph. This array has the same format as that returned by PolyFit.

SetDataStyle (ps)

Set the point style *ps* used to plot individual data points in observation plots and confidence interval plots. For instance, SetDataStyle(2) will draw data points in point style 2 in subsequent observation or confidence interval plots. Your *ps* must be a point style as defined in the "Making Graphs" section or 0 to omit plotting the raw data points. By default, raw data is not plotted in these graphs.

AskDataStyle (ps)

The opposite of SetDataStyle. Return the current point style used for raw data points in observation plots and confidence interval plots.

SetErrorBeam (pix)

By default, error bars are shown as simple vertical lines. You can switch to an I-beam shape if you wish.

To get I-beams, call SetErrorBeam before you graph a data range. Pass in *pix* the pixel length for each side of the cross-bar. The full cross-bar is 2**pix*+1 pixels wide.

All subsequent data ranges will be drawn as I-beams with cross-bars of this size. For most computers, 2 or 3 pixels are enough for a cross-bar. But you can make them as big as you want. To go back to the simple vertical lines, just pass *pix* = 0.

AskErrorBeam (pix)

AskErrorBeam is the opposite of SetErrorBeam. It returns the current size of the cross-bar for error bars (in pixels).

SetObsSD (n)

Call SetObsSD(*n*) to create error bars *n* * *sd* units long, where *sd* is the standard deviation of a set of observations. For instance SetObsSD(2) will tell PlotObs to produce error bars 2 sd's long about the mean of each observation set. To turn off error bars, call SetObsSD(0). By default, *n* = 1.

Notes: use SetSD to control whether the standard deviation is computed with a denominator of *n* data items or *n* – 1. If you call SetObsSD, then SetObsSE will automatically be turned off.

Exception:

704	SetObsSD can't be negative: n

AskObsSD (n)

The opposite of SetObsSD. Returns the current sd multiplier for error bars in observation plots.

SetObsSE (n)

Call SetObsSE(n) to create error bars n * *se* units long, where *se* is the standard error of the mean of a set of observations. For instance, SetObsSE(2) will tell PlotObs to produce error bars 2 sem's long about the mean of each observation set. To turn off error bars, call SetObsSE(0). By default, $n = 0$.

Notes: use SetSD to control whether the standard deviation, and hence the standard error of the mean, is computed with a denominator of n data items, or $n - 1$. If you call SetObsSE, SetObsSD will automatically be turned off.

Exception:

704	SetObsSE can't be negative: n

AskObsSE (n)

The opposite of SetObsSE. Returns the current *se* multiplier for error bars in observation plots.

SetObsBox (f)

Pass $f = 1$ to use box-and-whisker plots in observation plots, or 0 to return to using the default (error bars).

AskObsBox (f)

The opposite of SetObsBox. Returns $f = 0$ if observation plots use error bars, 1 if they use box plots.

Mixing Statistics, 3-D, Scientific, and Business Graphics

This section shows how you can mix and match tools from True BASIC's graphics packages. By using tools from different packages together, you can get clear and useful graphs tailored to specific problems.

The *3-D Graphics Toolkit* lets you draw data plots or functions in three dimensions with just one or two subroutine calls. It's ideal for displaying multi-dimensional data.

The *Business Graphics Toolkit* provides very good ways to draw overlapped histograms, scatter plots that use words or letters as point marks, pie charts and area charts.

The *Scientific Graphics Toolkit* can draw splines and Bezier curves, and plots arbitrary functions with ease.

Just remember to call SetOverlay(1) as you do when overlaying scientific graphs. You may also want to call SetScale from the Business Graphics Toolkit to alter the overlaid chart's scale.

NOTE: If you use the *Business, Scientific,* and *Statistics Graphics Toolkits* together, you need only one copy of FRAMELIB. If your copies of FRAMELIB have different version numbers, use the latest version. Your graphs will all have scientific-style tick marks inside all four edges of the frame. Call SetInTicks("") to get the usual *Business Graphics* ticks.

3-D Graphics

The STAT3D program, on your disk, shows a 3-D bar chart of a 5 x 5 contingency table so you can easily see the data distribution. It adds bar charts for the row and column sums along the sides, and prints various statistics in a separate window along the bottom of the screen. The bar-chart routines come from the *3-D Graphics Toolkit*, so you need that Toolkit to run STAT3D.

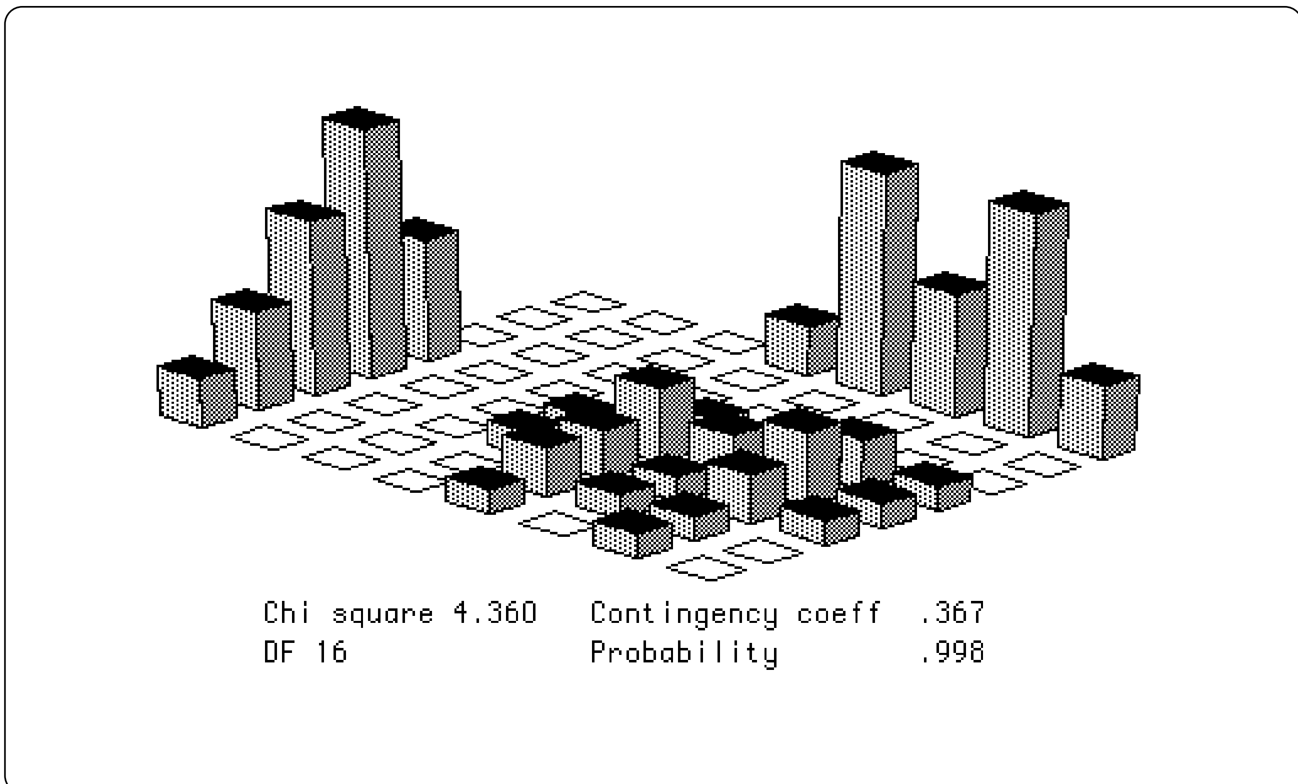

Figure 43.43: Output of the STAT3D program.

Data Transforms

This section describes the Data Transforms available in the *Statistics Graphics Toolkit*. Transformation is a very powerful technique in data analysis that can be used to clarify raw data with any of the following characteristics:

- Strong asymmetry
- Many outliers in one tail
- Batches at different levels with different spreads
- Large and systematic residuals

The illustration below, taken from *Understanding Robust and Exploratory Data Analysis*, shows output of your diskette's CITIES1 and CITIES2 programs side by side. Notice how applying a Log10 transform to the raw data corrects for increasing spread as a function of level.

Notice how many "spurious" outliers have been removed, and how smaller countries are now much easier to read.

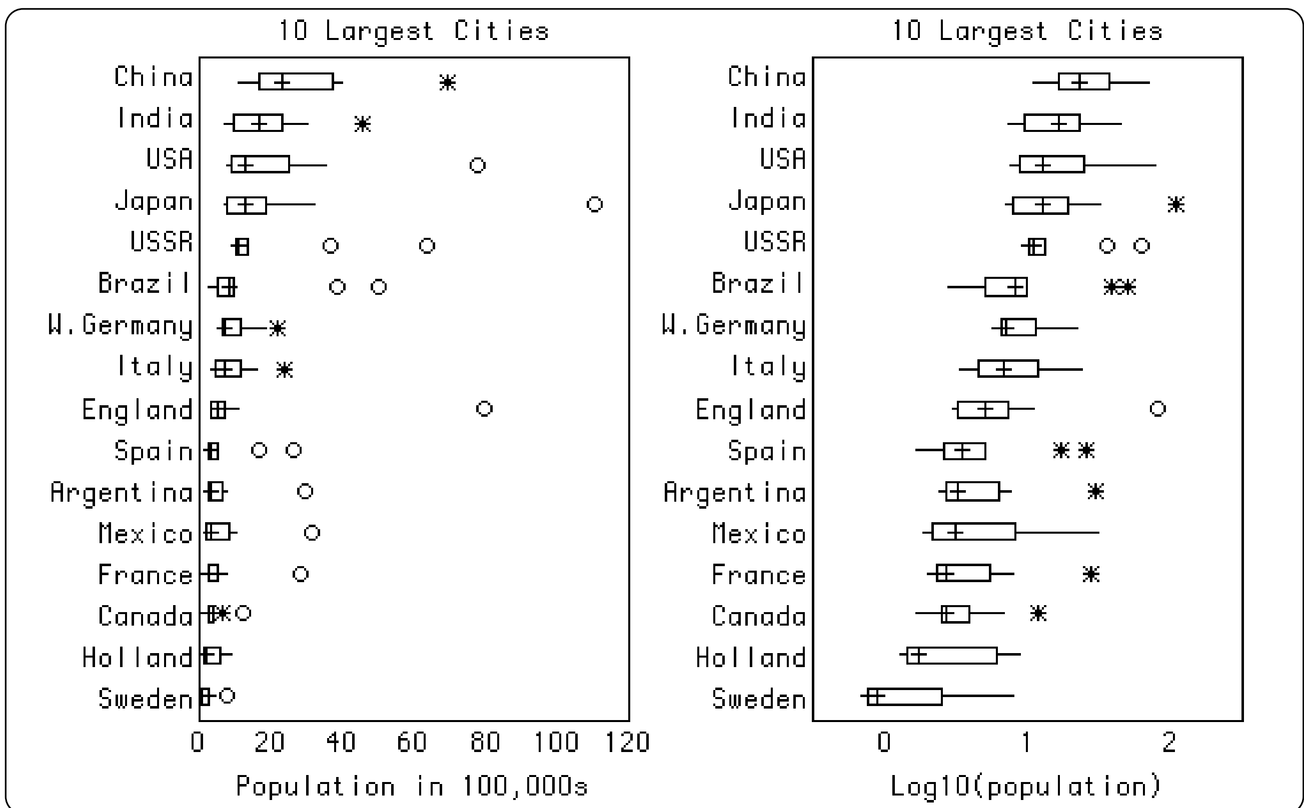

Figure 43.44: Using Log10 Transform to Correct Spread

True BASIC's MAT Statements

Remember that you can use the **mat** statements for many kinds of transforms. For instance, to "aggregate" two variables *x1*() and *x2*() into a new variable *x*():

```
mat x = x1 + x2
```

You can build even complicated weighted aggregations by sequences of **mat** scalar multiplications and additions.

One-Dimensional Transforms

These transforms handle advanced matrix manipulation that can't be done directly with **mat** statements.

If these routines cannot transform a data value — for example, if they must take Sqr of a negative number — they supply the missing value instead or give an error if you have disallowed missing values.

PowerTran (x(), n)

Box-Cox power transform. If $n = 0$, then $x' = \text{Log}(x)$; else $x' = (x^n-1)/n$.

LogitTran (x())

Logit transform: $x' = \text{Log}(x/(1-x))$.

AddTran (x(), k)

Add a constant: $x' = x + k$. Note that multiplication and division can be accomplished with True BASIC **mat** statements: **mat x = k * x** or **mat x = (1/k) * x**.

ToNthTran (x(), n)

Nth power: $x' = x^n$. Note that n can be negative; for example, $n = -1$ gives $1/x$.

SinTran (x())

Sine transform: $x' = \text{Sin}(x)$.

CosTran (x())

Cosine transform: $x' = \text{Cos}(x)$.

AsinTran (x())
Arcsine transform: $x' = \text{Asin}(x)$.

SqrTran (x())
Square root transform: $x' = \text{Sqr}(x)$.

LogTran (x())
Natural logarithm transform: $x' = \text{Log}(x)$.

Log2Tran (x())
Logarithm base-2 transform: $x' = \text{Log2}(x)$.

Log10Tran (x())
Common logarithm transform: $x' = \text{Log10}(x)$.

ExpTran (x())
Natural exponential transform: $x' = e^x$.

SgnTran (x())
Sign transform: $x' = \text{Sgn}(x)$.

IntTran (x())
Greatest integer (“floor”) transform: $x' = \text{Int}(x)$.

Two-Dimensional Transforms

The same transforms are also defined for two-dimensional data arrays (*not* frequency arrays!):

PowerTran2 (x(,), n)
Box-Cox power transform. If $n = 0$, then $x' = \text{Log}(x)$;
else $x' = (x^n - 1)/n$.

LogitTran2 (x(,))

Logit transform: $x' = \text{Log}(x/(1-x))$.

AddTran2 (x(,), k)

Add a constant: $x' = x + k$. Note that multiplication and division can be accomplished with True BASIC **mat** statements: **mat x = k * x** or **mat x = (1/k) * x**.

ToNthTran2 (x(,), n)

Nth power: $x' = x^n$. Note that n can be negative; for example, $n = -1$ gives $1/x$.

SinTran2 (x(,))

Sine transform: $x' = \text{Sin}(x)$.

CosTran2 (x(,))

Cosine transform: $x' = \text{Cos}(x)$.

AsinTran2 (x(,))

Arcsine transform: $x' = \text{Asin}(x)$.

SqrTran2 (x(,))

Square root transform: $x' = \text{Sqr}(x)$.

LogTran2 (x(,))

Natural logarithm transform: $x' = \text{Log}(x)$.

Log2Tran2 (x(,))

Logarithm base-2 transform: $x' = \text{Log2}(x)$.

Log10Tran2 (x(,))

Common logarithm transform: $x' = \text{Log10}(x)$.

ExpTran2 (x(,))

Natural exponential transform: $x' = e^x$.

SgnTran2 (x(,))

Sign transform: x' = Sgn(x).

IntTran2 (x(,))

Greatest integer ("floor") transform: x' = Int(x).

Probabilities and Critical Values

This section describes the functions that the *Statistics Graphics Toolkit* uses to approximate significance probability and critical value functions.

Probabilities and critical values are computed for normal, Student-T, chi-square, and F distributions. The results are accurate to about 8 significant digits in most cases.

For small samples, you should always check critical values against published critical value tables; but these functions are very good approximations for larger sample sizes, and also handy for "rough and ready" calculations.

def NorProb (x)

Returns the Gaussian (normal) integral to the left of x. For right tail probabilities, call with argument $-x$ instead of x.

def NorCrit (p)

Returns the critical value for the normal distribution of right-tail probability p.

def Tprob (df, x)

Returns the t-probability integral to the left of x with df degrees of freedom. For right tail probabilities, call with argument $-x$ instead of x.

def Tcrit (df, p)

Returns the critical value for the t-distribution with df degrees of freedom of right-tail probability p.

def ChiProb (df, x)

Returns the chi-square density integral to the left of x with df degrees of freedom.

def ChiCrit (df, p)

Returns the critical value for the chi-square distribution with *df* degrees of freedom of right-tail probability *p*.

def Fprob (df1, df2, x)

Returns the F-distribution integral to the left of *x* with *df1* and *df2* degrees of freedom.

def Fcrit (df1, df2, p)

Returns the critical value for the F-distribution with *df1* and *df2* degrees of freedom of the right-tail probability *p*.

Simulated Distributions

This section describes simulated distributions — sets of random numbers created from True BASIC's Rnd function that approximate various statistical distributions. For instance, SimNormal creates a normally-distributed set of random numbers, and SimPoisson creates a set with Poisson distribution.

Most of these routines are rather slow since it takes a great deal of computation to simulate the distributions.

In all these routines, *y*() may have any lower bound. Its upper bound is adjusted as needed to hold the random sample. The sample size requested must not be negative.

SimNormal (n, mean, sd, y())

Put *n* normally distributed random numbers, with given *mean* and *sd*, into *y*.

Exceptions:

718	Number of samples must be > 0.
719	SimNormal SD must be >= 0: sd

SimBinomial (k, n, p, y())

Put *k* random numbers in *y*, with a binomial distribution of *n* trials with *p* probability of success.

Exceptions:

718	Number of samples must be > 0.
720	SimBinomial sample size must be > 0: n
738	Probability must be in range 0 to 1 inclusive for SimBinomial.

SimNegBinomial (k, nsucc, p, y())

Put *k* random numbers in *y*, with a negative binomial distribution of *nsucc* successful trials with *p* probability of success in each trial.

Exceptions:

718	Number of samples must be > 0.
720	SimNegBinomial sample size must be > 0: nsucc

SimPoisson (n, lambda, y())

Put *n* random numbers in *y,* with a Poisson distribution about the mean *lambda*.

Exceptions:

718	Number of samples must be > 0.
720	SimPoisson mean must be > 0: lambda

SimGeom (n, p, y())

Put *n* random numbers in *y,* geometrically distributed about the mean $(1-p)/p$.

Exceptions:

718	Number of samples must be > 0.

SimExp (n, mean, y())

Put *n* random numbers in *y,* exponentially distributed about the mean *mean*.

Exceptions:

718	Number of samples must be > 0.
720	SimExp mean must be > 0: mean

SimErlang (n, k, mean, y())

Put *n* random numbers in *y,* with an Erlang distribution of *k* events about a *mean.*

Exceptions:

718	Number of samples must be > 0.
719	SimErlang events must be >= 0: k

SimBeta (n, e1, e2, y())

Put *n* random numbers in *y*, with beta distribution of *e1* events in ratio to *e2* events.

Exceptions:

718	Number of samples must be > 0.
719	SimBeta events #i must be >= 0: ei

SimHyperGeom (n, ntri, p, pop, y())

Put *n* random numbers in *y,* with a hypergeometric distribution of *ntri* number of trials with *p* probability of success in each trial, drawn from a population of size *pop.*

Exceptions:

718	Number of samples must be > 0.
719	SimHyperGeom number of trials must be >= 0: ntri
719	SimHyperGeom population size must be >= 0: pop

SimLogNormal (n, mean, sd, y())

Put *n* log-normally distributed random numbers in *y,* with given *mean* and *sd*.

Exceptions:

718	Number of samples must be > 0.
719	SimLogNormal SD must be >= 0: sd

SimUniform (n, a, b, y())

Put *n* uniform-continuously distributed random numbers in *y,* drawn from the interval [*a,b*].

Exceptions:

718	Number of samples must be > 0.
724	Left endpoint must be < than right in SimUniform.

SimDiscUniform (n, a, b, y())

Put *n* discrete, uniformly distributed random numbers in *y*. That is, all the numbers are integers in the range [*a,b*].

Exceptions:

718	Number of samples must be > 0.
724	Left endpoint must be < than right in SimDiscUniform.

SimSequence (n, from, step, y())

Generate in *y*() an ordered sequence of *n* numbers evenly spaced, starting at *from* with a step of *step*.

Exceptions:

718	Number of samples must be > 0.

SimLinear (x(), slope, intercept, sd, y())

Given a dataset *x*() with a *slope* and *intercept*, create *y*() values that are normally distributed along the regression line with a standard deviation of *sd*.

Exceptions:

719	Simlinear SD must be >= 0: sd

Simulated Frequency Datasets

The following routines create simulated *frequency* datasets. These have the same format as those created from raw data by DataToFreq, etc. See the "Frequency Distributions" section for more information.

SimFreqNormal (mean, sd, from, to, step, y(,))

Create a normally-distributed frequency distribution in *y*(,).

Exceptions:

719	SD in SimFreqNormal must be >= 0: sd
722	Can't have 0 step size in sequence.
723	Can't have zero values in sequence.

SimFreqBinomial (n, p, from, to, step, y(,))

Create a binomially-distributed frequency distribution in *y*(,).

Exceptions:

721	p in SimFreqBinomial must be inside (0,1): p
722	Can't have 0 step size in sequence.
723	Can't have zero values in sequence.

SimFreqExp (emean, from, to, step, y(,))

Create an exponentially-distributed frequency distribution in *y*(,).

Exceptions:

719	Mean in SimFreqExp must be >= 0: emean
722	Can't have 0 step size in sequence.
723	Can't have zero values in sequence.

SimFreqPoisson (pmean, from, to, step, y(,))

Create a Poisson frequency distribution in *y*(,).

Exceptions:

719	Mean in SimFreqPoisson must be >= 0: pmean
722	Can't have 0 step size in sequence.
723	Can't have zero values in sequence.

SimFreqUniform (left, right, from, to, step, y(,))

Create a uniformly-distributed frequency distribution in *y*(,).

Exceptions:

722	Can't have 0 step size in sequence.
723	Can't have zero values in sequence.
724	Left endpoint must be < than right in SimFreqUniform.

Sampling

This section describes how to sample existing datasets (either real or simulated) to create new datasets. You can sample with or without replacement.

SampleRep (n, x(), y())

Create a new dataset *y*() consisting of *n* elements randomly drawn from the *x*() dataset. This routine samples with replacement; that is, the same *x* item may be placed into the *y*() dataset repeatedly.

The *x*() and *y*() arrays may have any bounds. The lower bound of *y*() will remain unchanged, and the upper bound will be adjusted to hold *n* elements.

SampleNoRep (n, x(), y())

Create a new dataset *y*() consisting of *n* elements randomly drawn from the *x*() dataset. This routine samples without replacement; that is, a given *x* item may be placed into the *y*() dataset *only once*.

The *x*() and *y*() arrays may have any bounds. The lower bound of *y*() will remain unchanged, and the upper bound will be adjusted to hold *n* elements.

Exception:

725	Sample size > population size in SampleNoRep: n

Subscript Functions

This section describes how to use the functions that work as subscripts for statistics arrays like those returned by Stats.

Using Subscript Functions

If you've used the **load** command to bring STAT1LIB through STAT4LIB into memory, you don't have to do anything special to use the subscript functions. Just use them like as if they were built into True BASIC:

```
dim d(0), s(0)
mat input d(?)
call Stats (d, s)
print "Mean: "; s(ls_mean)
end
```

Binding Programs — A Warning!

Before you bind your programs with the Runtime Package, however, you must add **declare def** statements that declare every subscript function that you use.

Otherwise True BASIC will treat these functions as if they are variables. Since they will have 0 as values, you will get "Subscript out of bounds" errors in your bound programs.

For example, the program above will run correctly in the True BASIC environment *but will fail with a subscript error if you bind it and run it!*

If you pre-load the libraries STAT1LIB through STAT4LIB, you can bind your program using the `BIND` command without having to use *declare def*.

Complete List of Subscript Functions

Below is a complete list of subscript functions. Consult the appropriate section for further information about how to use any of these functions.

Simple Statistics Subscript Functions

st_n	number of elements
st_nm	number of missing elements
st_nnm	number of non-missing elements
st_sum	sum of items
st_mean	mean (average)
st_ssq	sum of squares
st_var	variance
st_sd	standard deviation
st_sem	standard error of the mean
st_med	median
st_low	lowest value
st_hi	highest value
st_range	range
st_rms	root mean square (quadratic mean)
st_md	mean absolute deviation
st_cvar	coefficient of variance
st_wmean	Winsorized mean

Letter Values Subscript Functions

lv_nnm	number of non-missing elements
lv_med	median
lv_lhin	left hinge (roughly 1st quartile)
lv_rhin	right hinge
lv_leig	left eighth
lv_reig	right eighth
lv_linf	left inner fence
lv_rinf	right inner fence
lv_louf	left outer fence
lv_rouf	right outer fence
lv_lout	number of left outliers (to left of left extreme)
lv_rout	number of right outliers (to right of right extreme)
lv_lext	leftmost value inside left inner fence
lv_rext	rightmost value inside right inner fence
lv_lmax	smallest value
lv_rmax	largest value
lv_tri	Tukey's trimean

Least-Squares Statistics Subscript Functions

ls_n	number of non-missing x/y points
ls_slo	slope of fitted line
ls_int	intercept of fitted line
ls_xbar	mean of $x()$
ls_ybar	mean of $y()$
ls_ssx	sum of squares x
ls_sxy	sum of products xy
ls_ssy	sum of squares y
ls_sse	sum of squares error
ls_se	standard error
ls_ts	t-statistic for slope
ls_dfs	degrees of freedom for slope
ls_p	probability for slope's t-statistic
ls_r	Pearson's product-moment correlation coefficient
ls_r2	r^2 (coefficient of determination)
ls_z	Fisher's z-transform of r
ls_f	F-statistic (same as ts^2)

Regression ANOVA Statistics Subscript Functions

ra_ssm	sum squares mean
ra_ssr	sum squares regression
ra_sse	sum squares error (residual)
ra_sst	sum squares total
ra_dfm	degrees of freedom mean
ra_dfr	degrees of freedom regression
ra_dfe	degrees of freedom error (residual)
ra_dft	degrees of freedom total
ra_msr	mean square regression
ra_mse	mean square error (residual)
ra_se	standard error
ra_f	F-statistic
ra_p	Prob(*f*)
ra_r	multiple R
ra_r2	R-square
ra_ar2	adjusted R-square
ra_d	Durbin-Watson d statistic
ra_press	PRESS statistic

ANOVA Statistics Subscript Functions

an_msw	mean square within
an_msb	mean square between
an_ssw	sum of squares within
an_ssb	sum of squares between
an_sst	sum of squares total
an_dfw	degrees of freedom within
an_dfb	degrees of freedom between
an_dft	degrees of freedom total
an_f	F-statistic
an_p	Prob(F)

Advanced Graph Control

This section describes advanced techniques for creating graphs. By using these techniques, you can customize your graphs to fit your needs exactly.

Customizing the Toolkit

The *Statistics Graphics Toolkit* has a number of preset controls. For instance, grid lines are preset off, frame ticks are preset to "LRBT", and so forth. These preset values are listed at the end of this section. Also, by default the Toolkit does not use Yates' correction when analyzing contingency tables, and so forth.

It's easy to customize your own copy of the Toolkit, though. Just create your own little module that sets the controls in its initialization code. A sample follows:

```
module MyKit
     call  SetYates(1)
     call SetInTicks("BL")
     call SetCanvas("red")
     call SetAxes(0)
end module
```

Save this module on your disk with the name, say, of MYSTAT. Then change the LOADSTAT file so it loads MYSTAT as well as the compiled versions of FRAMELIB and STATLIB:

```
load FRAMELIB, ..., MYSTAT
```

Now you're all set. Just start your sessions by giving the **script LOADSTAT** command. It will load the necessary modules into memory and customize the Toolkit for you.

Using Windows to Draw Multiple Graphs

The sample program CUMFP shows how to use windows to draw multiple graphs on the screen. Call up the program and read it. First it opens window #1 and draws a graph in this window. Then it opens window #2 and draws another graph.

You can draw graphs in windows of any size or shape. However, some graphs won't fit inside windows that are too narrow or too short. If the title, horizontal label, or vertical label won't fit, you'll have to shorten these labels or use a bigger window. See the "Trouble-Shooting" section for more help.

Implicit X-Coordinates

Many datasets have straightforward *x* coordinates. The most common coordinates are 1, 2, 3, ... Graphing datasets with these coordinates requires a very simple *x*() array: *x*(1) = 1, *x*(2) = 2, and so forth.

You can skip the chore of setting up such simple *x* arrays when graphing datasets. Just pass an *x* array with no elements, created by a statement such as **dim** *x*(0). Then the *x*() array will be ignored when plotting data points.

If the graph has a fixed *x* scale – you've called SetXscale or are overlaying an existing graph – the graphing routines will use that scale. They will evenly space data points along the graph's *x* scale with the first point at the minimum *x* value and the last point at the maximum.

If, on the other hand, the graph is auto-scaled, then the graphing routines will base the data points' *x* coordinates on the lower and upper bounds of the *y* array. When *y*'s lower bound is 1, this gives 1, 2, 3, ... as the *x* coordinates. But if you used **dim** *y*(-10 to 10) the *x* coordinates will reflect these bounds: -10, -9, ..., 9, 10.

You can use null *x* arrays with any kind of scatter plot.

Horizontal and Vertical Grid Lines

It's easy to get horizontal and/or vertical grid lines on your graph. Just add this statement somewhere before you actually draw a graph:

```
call SetGrid("hv")
```

The string "hv" gives both horizontal and vertical grid lines. You can also use "h" to get only horizontal grid lines or "v" to get only vertical grid lines.

To get dashed, dotted, or dash-dotted grid lines, give the appropriate symbols after the *h* or *v*. For instance, call SetGrid("h.v-.") to get dotted horizontal lines and dot-dashed vertical lines. See the description of SetGrid in the "Low-Level Control" section.

The grid-line instructions stay in effect until your program stops; if you draw several graphs in a row, each will have the same kind of grid lines. You can turn off the grid lines entirely by calling SetGrid("").

The Graph's Canvas Color

You can give the canvas its own color to make it stand out from the rest of the screen. The canvas color is like a background color for the canvas. (In fact, it's not really a background color; it's one of the foreground colors.) You can use any foreground color as the graph's canvas color. Thus for a red canvas:

```
call SetCanvas("red")
```

You can use any color instead of “red”. You can also use color numbers. Remember to draw the data in a different color!

If you draw several graphs in a row, each will have the same canvas color. To use the real background color for the canvas, pass ““ or “background”.

Overlaying Graphs

The *Statistics Graphics Toolkit* generally “uses a new sheet of paper” for each graph you draw. The Add functions – AddScatPlot, AddNormalPlot, etc. – let you overlay one graph with another. These routines will take care of most of your overlaying needs. But there’s also a more general way to overlay graphs. It lets you overlay any kind of graph with any other kind.

To do so, call SetOverlay(1) after you’ve drawn a graph. Then draw another graph. It will be drawn on top of the existing graph. You can overlay as many graphs as you like – just call SetOverlay(1) before each overlay. The opposite is AskOverlay(n), which returns n = 1 if the next graph will be overlaid, or 0 if not.

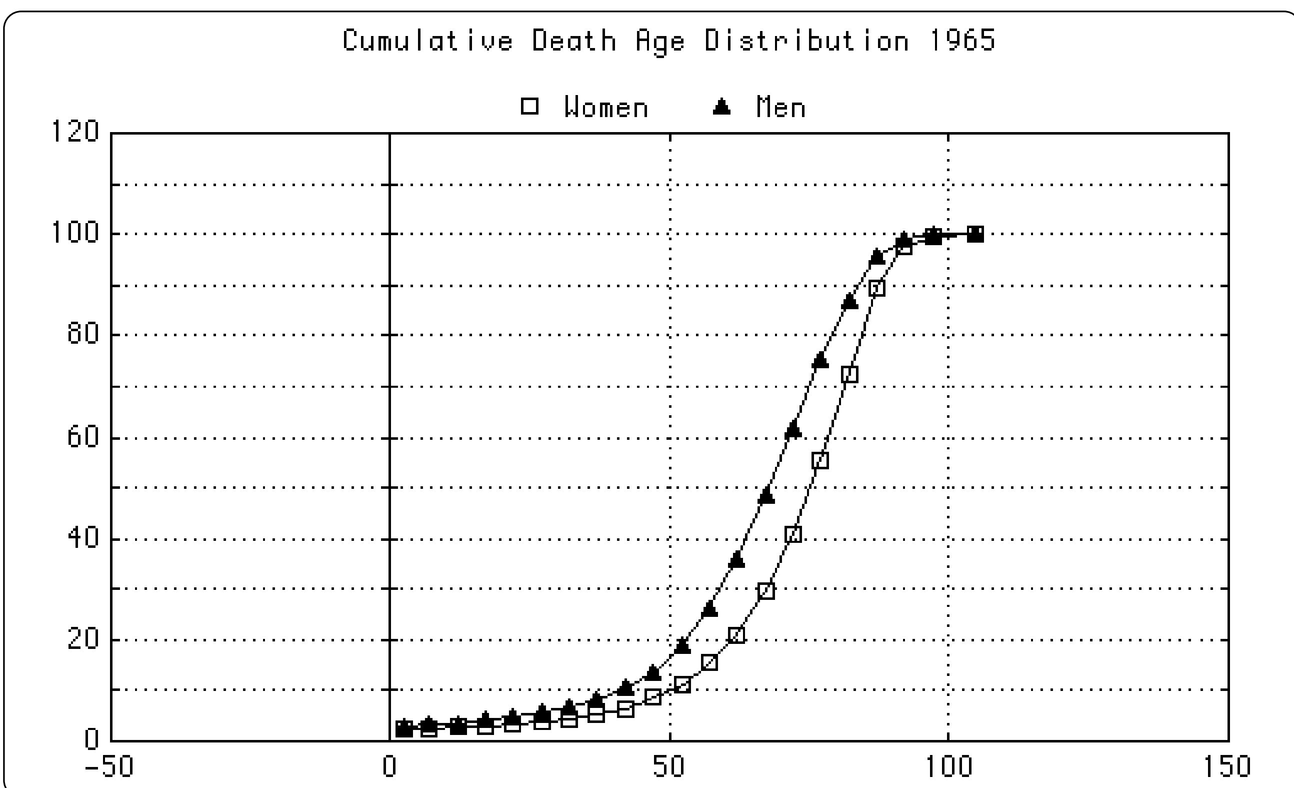

Figure 43.45: Output of the OVERSTAT program.

The new graph inherits the same scale as the existing graph. But you can overlay graphs with different scales. First, draw the original graph. Then call SetXscale

and/or SetYscale to change the graph scale. Finally, call SetOverlay(1) and draw the second graph. The final graph still has its original frame, so there will be no indication if overlaid data is drawn to a different scale. You may want to add text to the graph (by GraphText) to note that different scales are being used.

Drawing in the Canvas

It's not hard to add your own graphics inside the canvas. This lets you customize your graphs by adding more labels or special effects which the *Statistics Graphics Toolkit* does not provide.

When you call the graph-drawing subroutines, they open the canvas as a True BASIC window. The frame is not part of this window – only the canvas itself is inside the window. This window's coordinates are created to mimic the numbers shown along the side of the frame. The coordinate system is set up so that your drawings will align precisely with the tick marks on the frame.

Logarithmic axes are slightly modified. The window coordinate system is based on the Log10 of the numbers shown along the edge.

To switch to the canvas window, call:

```
call GotoCanvas
```

Drawing Points, Lines, and Text

The *Statistics Graphics Toolkit* includes several routines that make it easy to draw nice-looking points, lines, and labels on the canvas. The MARKSTAT program, on your disk, shows how you can use these routines.

GraphText(x, y, text$)

GraphText is like True BASIC's **plot text** statement. It places the *text$* label on your graph at (*x,y*). But unlike **plot text**, it works for any graph type – normal, log, or semi-log.

GraphPoint(x, y, style)

GraphPoint draws a point at location (*x,y*) in the indicated *style*. It converts from your graph's coordinates – normal, log, or semi-log – to canvas coordinates. See the "Making Graphs" section for a list of supported point styles.

Exceptions:

129 Unknown point style: n

GraphLine(x1, y1, x2, y2, style)

GraphLine draws a straight line from (*x1,y1*) to (*x2,y2*) in the indicated line *style*. It converts from your graph's coordinates – normal, normal, log, or semi-log – to canvas coordinates. See the "Making Graphs" section for a list of supported line styles.

Exceptions:

130 Unknown line style: n

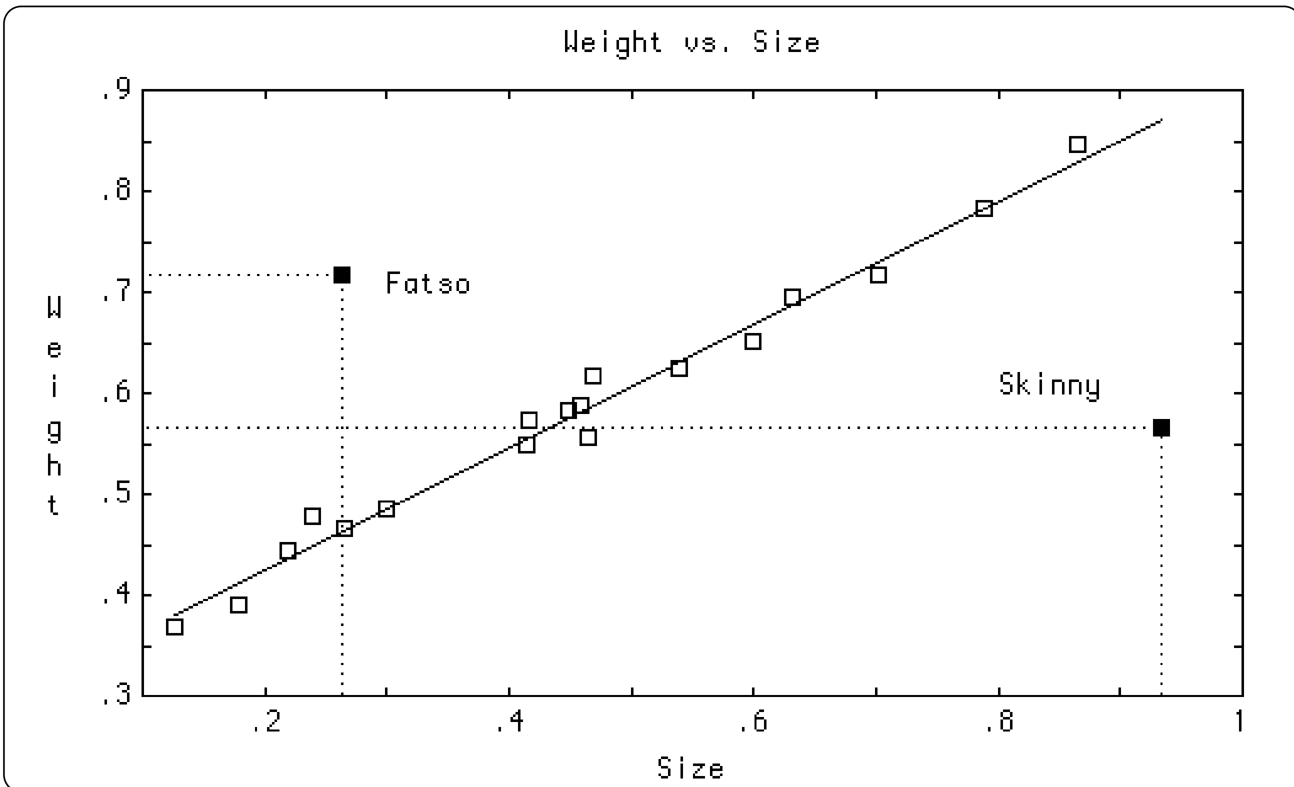

Figure 43.46: Output of the MARKSTAT program.

Routines Shared with Scientific Graphics

The routines listed in this section are shared with the *Scientific Graphics Toolkit*. You can use the same routines, therefore, with either toolkit.

NOTE: To use both Toolkits, follow these instructions. The file named MSGLIB, is a stripped-down form of the *Scientific Graphics* file SGLIB. Change the LOADSTAT file and the library **statement in STAT2LIB to refer to your copy of SGLIB rather than MSGLIB. Then recompile STAT2LIB and replace its old compiled version.**

SetGraphType (type$)

SetGraphType changes the current graph type to *type$*. This type stays in effect until you call SetGraphType again, or your program halts. The supported graph types are:

"XY" *or* ""	normal
"LOGX"	semi-log (X axis is logarithmic)
"LOGY"	semi-log (Y axis is logarithmic)
"LOGXY"	logarithmic (both axes logarithmic)

After you call SetGraphType, you must call a graphing routine (described in the previous sections) to draw the graph.

AskGraphType (type$)

AskGraphType returns the current graph type, in uppercase, in *type$*. This will be one of the following:

"XY"	normal
"LOGX"	semi-log (X axis is logarithmic)
"LOGY"	semi-log (Y axis is logarithmic)
"LOGXY"	logarithmic (both axes logarithmic)

SetAxes (f)

SetAxes turns the axes on or off. Pass $f = 0$ to remove the axes or any nonzero f to show the axes. The new setting persists until you call SetAxes again, or until your program stops. By default, axes are shown.

AskAxes (f)

AskAxes returns $f = 1$ if the axes are shown or 0 if not.

SortPoints (x(), y())

SortPoints sorts arrays of x and y coordinates by the x values. It's useful for arranging data in order before you draw lines connecting data points.

Call up the program SORTSTAT from your disk and run it. It draws a set of data points, connecting each to its neighbors. The left window shows the result for data in no particular order – the connecting lines go all over everywhere. The right window shows the same data points sorted by x coordinates before graphing.

SortPoints2 (x(,), y(,))

SortPoints2 is like SortPoints except that it sorts 2-dimensional arrays of *x* and *y* values. Each row is treated as a separate dataset. The rows are sorted independently – no row influences any other.

SetInTicks (where$)

SetInTicks controls where ticks appear on the graph's frame. By default, ticks are drawn inside all four edges of the frame. This is easy to change, however. Just call SetInTicks giving the new edges to tick:

L	left edge
R	right edge
T	top edge
B	bottom edge

For example, SetInTicks("LB") asks that subsequent graphs be drawn with ticks inside the left and bottom edges only. You can give the edge letters in upper or lower case, in any order.

If you pass the null string, ticks will be drawn outside the frame on the left and bottom edges. (The *Business Graphics Toolkit* uses this style.)

Use the SetTickSizes routine to control the sizes of ticks. You can eliminate ticks entirely by calling SetTickSizes(0,0).

AskInTicks (where$)

AskInTicks is the opposite of SetInTicks. It returns an uppercase string that tells where the tick marks will be drawn for subsequent graphs. For example, it returns "LB" if ticks will be drawn at the left and bottom.

SetGrain (n)

SetGrain sets the grain with which polynomials and confidence bands are plotted. By default, the *Statistics Graphics Toolkit* plots a curved line as a series of 64 short line segments. Thus the default grain is 64.

You can make the grain larger or smaller. If you make it larger, curves will be slower but more accurate. If you make it smaller, they will be faster but less accurate.

AskGrain (n)

AskGrain returns the current grain size for polynomials and confidence bands. By default, this is 64. You can change this by calling SetGrain.

SetAxesTick (xp, yp)

SetAxesTicks controls the sizes of ticks drawn on the axes. The *x* and *y* axes' ticks will be *xp* and *yp* pixels long for the next graph you draw. See also SetTickSizes in the "Low-Level Control" section. It controls the sizes of ticks on the frame.

AskAxesTick (xp, yp)

AskAxesTick returns the current axes' tick sizes, in pixels.

SetAutoScale (x, y)

SetAutoScale turns the X and Y coordinate auto-scaling on or off. By default, the *Statistics Graphics Toolkit* auto-scales both the X and Y coordinates so that your graph looks good.

Pass $x = 0$ to turn off X auto-scaling; pass $y = 0$ to turn off Y auto-scaling. Any nonzero value will turn auto-scaling back on. These new controls for auto-scaling stay in effect until you change them again or until your program stops.

To get a series of graphs to the same scale, draw the first one with auto-scaling turned on, then turn off auto-scaling and draw the remaining graphs. The first scale will be used for all subsequent graphs. To supply your own scales, use SetXscale and SetYscale.

AskAutoScale (x, y)

AskAutoScale returns the current values for the X and Y auto-scale controls. These values are set to 1 if auto-scaling is turned on for that axis, or 0 if turned off.

SetXscale (x1, x2)

SetXscale turns off auto-scaling for the X axis, and forces the *Statistics Graphics Toolkit* to use the interval *x1* to *x2* for subsequent graphs. The Toolkit will continue to use this interval until you change it again, or use SetAutoScale to restore auto-scaling for the X axis, or your program stops.

This Toolkit rounds scales to good-looking numbers, so the final *x* scale may not be exactly what you asked for.

AskXscale (x1, x2)

AskXscale returns the current *x* range, running from *x1* to *x2,* where $x1 \le x2$. If you've previously called SetXscale, these values are the ones you supplied. Otherwise, they'll be automatically computed from the data you gave.

SetYscale (y1, y2)

SetYscale turns off auto-scaling for the Y axis, and forces the *Statistics Graphics Toolkit* to use the range *y1* to *y2* for subsequent graphs. The Toolkit will continue to use this range until you change it again, or use SetAutoScale to restore auto-scaling for the Y axis, or your program stops.

This Toolkit rounds scales to good-looking numbers, so the final *y* scale may not be exactly what you asked for.

AskYscale (y1, y2)

AskYscale returns the current range, running from *y1* to *y2,* where $y1 \le y2$. If you've previously called SetYscale, these values are the ones you supplied. Otherwise they'll be automatically computed from the data you gave.

GraphInit

GraphInit resets graph controls to their default values:

graph type	**xy**	grain	**64**
autoscale	**on**	spline order	**3***
axes	**on**	I-beam size	**0**
grid lines	**off**	mesh sizes	**20 x 20***
ticks	**LRBT**	least squares	**off**
frame ticks	**	axes ticks	**

*Used by *Scientific Graphics Toolkit* only.

** The number of pixels used for frame and axes ticks varies between computers and may change in later versions of the Toolkit.

Customized Legends

Complex graphs – which plot multiple datasets – automatically display legends based on the labels you pass in a *legends$*() array. These legends identify the different datasets. By default, they go into a horizontal row just below the graph's title.

If you wish, you can control the shape and location of the legend box and also control what appears in the box.

Opening Your Own Legend Window

To open your own legend window, call the OpenLegend subroutine before you draw a graph. You must pass the screen coordinates of the new legend window. Remember that these are screen coordinates, not window coordinates.

```
call OpenLegend (left,right,bottom,top)
```

This legend window will then be used in place of the default legend window for the next graph. If you want to use this window for several graphs, you must call OpenLegend before you draw each graph.

You can also reserve legend space just below the title, where the default legends go, by calling ReserveLegend instead of OpenLegend. The DrawFrame routine will then open a legend window for you when it draws the graph's frame.

```
call ReserveLegend
```

Making Your Own Legends

If you pass a *legends$*() array to the multi-dataset graph routines, they will automatically place legends in the legend window. You can, however, control these legends yourself. Here's how.

First, call OpenLegend or ReserveLegend to create a legend window. Next, call the appropriate graph routine – e.g. ManyObsPlot – with a *legends$*() array having no elements. Then call the AddLegend routine for each legend you want to display. Finally, call DrawLegend to display the legends.

You can also call GotoLegend to switch to the legend window. Then you can use True BASIC statements to draw directly in the window.

Call up the FP program from your disk for an example.

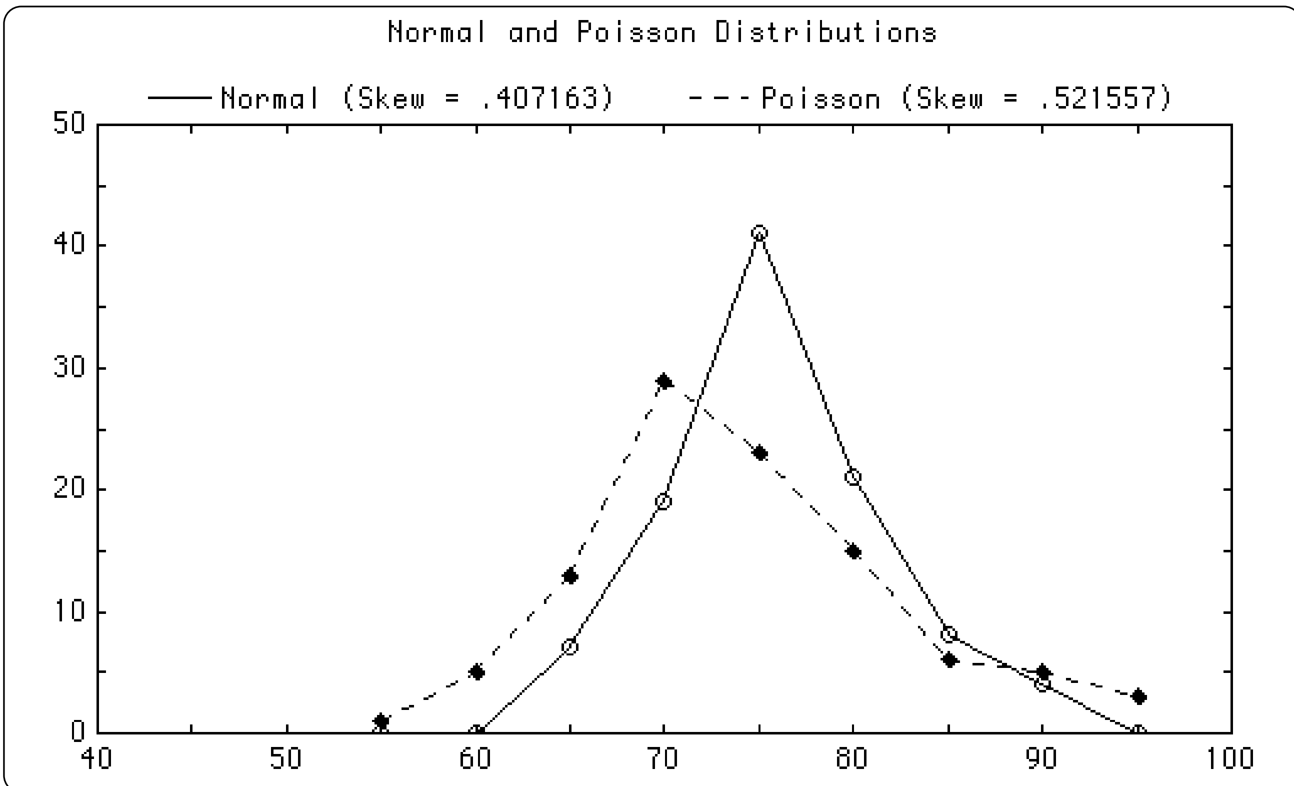

Figure 43.47: Output of the FP program. Note its custom legend.

AddLegend (text$, type, style, color$)

AddLegend adds a new legend to the legend window. It does not draw the legend, however. After you've added all the legends, call DrawLegend to show the legends.

The *text$* string gives the text of a legend. Its *type* controls the kind of symbol shown next to the text: 1=box, 2=line, 3=point. If *type*=0, no symbol is shown.

The *style* controls the line or point style for a legend's symbol. Line and point styles are described in the "Making Graphs" section. This parameter is ignored for *type* = 1 symbols (boxes).

If you make a mistake with *type* or *style*, you won't get an error until you call DrawLegend.

The *color$* controls the color for this item's symbol. Its text, however, is always shown in the frame color.

DrawLegend

DrawLegend draws accumulated legends in the legend window. It will do its best to fit the legends in the window you've created (by OpenLegend), but it's not perfect.

Therefore you may have to juggle the coordinates in OpenLegend to get something that works and looks good.

When it's done, it returns to your original window. Call GotoLegend if you want to add more True BASIC graphics to the legend.

Exceptions:

101	Graph's legend is too wide.
115	Unknown line style: n
116	Unknown point style: n
119	No legend window yet.

Trouble-Shooting Graphics

This section describes some common problems, and what to do about them.

Colors flicker or are wrong.

Check your computer's graphics mode. You may be using more colors than that mode can handle, or a combination of colors that doesn't work in that mode. For instance, the color scheme "red white blue" will switch between the red-green-yellow and magenta-cyan-white palettes in the IBM PC "graphics" mode. It will also cause problems in the IBM "hires" mode since that mode can handle only one color.

Solution: Change the color scheme or use another mode.

The background color is not appealing.

Solution: Add a **set back** statement to change the background color of the entire screen. You may also wish to call SetCanvas to change the background color for the canvas.

The data display is not visible.

The data and canvas colors are identical, or the data and background colors are identical.

Solution: Change the data color (in the color scheme), the canvas color, or the background color.

The title is too wide.

If your title is too wide for the graph window, you'll get an error message.

Solution: Shorten the title. Or use a bigger window for drawing the graph. Or switch to a mode which allows more characters; for instance, use the "hires" mode on the IBM PC instead of the default "graphics" mode.

The horizontal or vertical label is too wide.

If your horizontal or vertical label can't fit, you'll get an error message.

Solution: Shorten the label. Or use a bigger window for drawing the graph. Or switch to a mode which allows more characters; for instance, use "hires" on the IBM PC instead of the default "graphics" mode.

You need more room for the horizontal or vertical marks.

This is like the preceding problem, but for marks rather than labels.

Solution: Read the paragraph above for some ideas. You may also be able to get more space for horizontal marks by shortening the vertical marks, since they restrict the space available for horizontal marks.

The ticks should be drawn somewhere else.

The *Statistics Graphics Toolkit* draws ticks inside all four edges of the graph. You may wish to omit ticks from some edges, draw ticks outside the frame, or get rid of ticks entirely.

Solution: Use SetInTicks to control tick placement. For example, SetInTicks("LB") draws ticks on the left and bottom edges only. SetInTicks("") switches to ticks along the outside of the frame. Or you can use SetTickSizes(0,0) to eliminate ticks entirely.

The ticks are too short or too long.

This Toolkit draws ticks inside the graph's frame and along the X and Y axes. These ticks may be too small or too large for your taste.

Solution: Call SetTickSizes to control the size of the frame's horizontal and vertical ticks; see the "Low-Level Control" section. Or call SetAxesTick to control ticks on the axes; see the "Advanced Graph Control" section.

The graph looks off-center.

Graphs are drawn so they use as much of the current window as possible. Since the

left edge contains a label and marks – and the right edge doesn't – the canvas may look off-center to the right.

Solution: Use the SetMargins subroutine to increase the number of unused pixels on the right side of the window. For instance, you might balance the graph by leaving an 80-pixel margin at the right side of the window. This is done by **call SetMargins(0,80,0,0)**.

You gave DATA in the wrong order.

Suppose you're using the PlotManyObs routine and want to give it 3 datasets of 15 points each. You should use **dim** *data*(3,15) and have three **data** statements with 15 values each. But perhaps instead you have 15 data statements, each of which has values for the three bands. What do you do?

Solution: Don't retype all the **data** statements! Instead, change the **dim** statement to create an array *data*(15,3). The **mat read** will now read this array as 15 bands of 3 points each. After the **mat read**, add the following True BASIC statement:

```
MAT data = Trn(data)
```

This transposes the data array so it becomes a 3 x 15 array from a 15 x 3 array. Now everything should work. (Aren't you glad that True BASIC has matrix transposing built in?)

The axes look ugly.

Sometimes the X and Y axes are convenient but sometimes they're distracting. How can you get rid of them?

Solution: Call SetAxes(0) to get rid of the axes. Or you can keep the axes but get rid of their tick marks by using SetTickSizes(0,0).

You want to control a graph's scale.

The *Statistics Graphics Toolkit* automatically picks scales for graphs, but you may not like the scale it picks. Or you may want to produce a series of graphs with the same scale.

Solution: Use the SetXscale and SetYscale routines. They're described in the "Advanced Graph Control" section.

You want to add text or graphics to a graph.

Once you've drawn a graph, you may want to add more text or graphics. For instance, you may want to label some data points, add a caption, or plot a function on top of data.

Solution: Call GotoCanvas to enter the canvas window. Then use True BASIC statements such as **plot** or **plot text** to add graphics directly to the canvas. The GraphPoint, GraphLine, and GraphText routines may also be useful. See the "Advanced Graph Control" section for details.

You want to customize a graph's legend.

You may want to add a legend to a graph, or to somehow customize where the legend appears or what goes in the legend.

Solution: Read the section on "Customized Legends."

You want to show 2 or more graphs on the screen.

Solution: See the "Advanced Graph Control" section for a description of how to do this.

Low-Level Control of the Frame

This section gives complete information on low-level routines that control the frame's appearance. These routines are for advanced users only – they are rather hard to use and not generally useful.

These routines are in the FRAMELIB library. This library is shared by the *Statistics, Scientific,* and *Business Graphics Toolkits*. If you have these other toolkits, you need only one copy of FRAMELIB.

NOTE: If your copies of FRAMELIB have different version numbers, use the latest version.

Some routines can be used together with the higher-level routines in the Toolkits. But most of them cannot, since the Toolkit routines override your own settings. Each routine that can't be used with the Toolkit routines is noted in its description.

Terminology

Please refer to the following figure. A frame contains a *title*, a *horizontal label*, and a *vertical label*. (Any of these three may be omitted.) See Figure 43.48.

It also contains *marks* along the left and bottom edges of the frame. These marks can be textual, such as "January", or numeric, such as 120. Statistics graphs usually have numeric marks along both the left and the bottom edges, but some observation plots have one edge textual and the other numeric.

The vertical marks run along the left edge. Horizontal marks run along the bottom edge. They are joined to the inner frame by vertical and horizontal ticks, respectively. Vertical ticks are thus *horizontal* lines, and vice versa.

The *canvas* is the area (inside the frame) where the graph or chart itself is drawn.

The *legend window* is where the legend goes. It's a separate window, distinct from the canvas window. The legend window is optional; it doesn't exist for many graphs.

GotoCanvas

GotoCanvas acts like True BASIC's **window** statement. It switches to the canvas window for the most recently created chart.

Exception:

118 No canvas window yet.

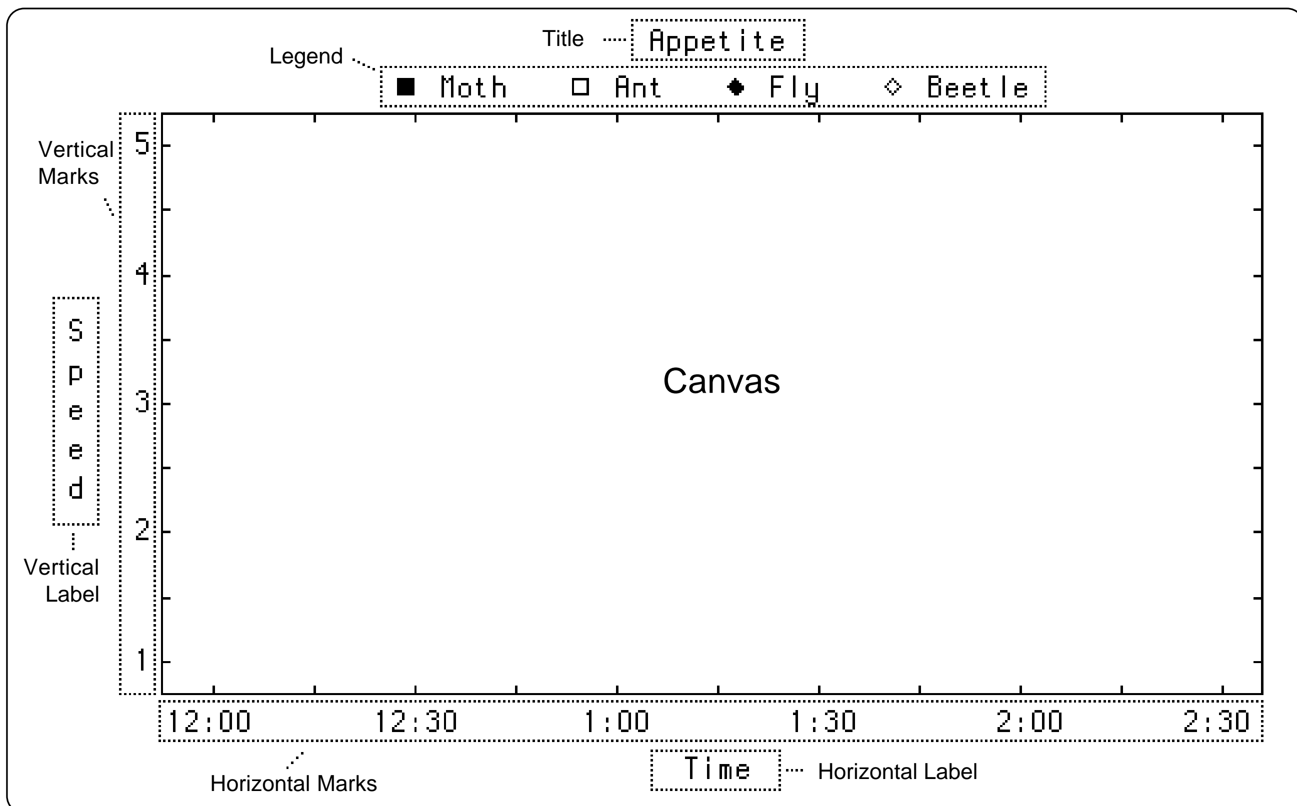

Figure 43.48: Parts of a Frame.

GotoLegend

GotoLegend acts like True BASIC's **window** statement. It switches to the most recently created legend window.

Exception:

119 No legend window yet.

SetGrid (style$)

SetGrid turns on/off the horizontal and vertical grid lines. You can pass any of these values (upper or lower case) in *style$*:

""	turn off all grid lines
"H"	turn on horizontal grids, turn off vertical
"V"	turn on vertical grids, turn off horizontal
"HV"	turn on both horizontal and vertical grids

If you turn on grid lines, DrawFrame will draw these lines automatically every time it draws a frame. It draws them at the same places that it puts tick marks on the frames. The grid lines are drawn in the frame color.

You can also ask for dashed, dotted, or dash-dotted grid lines. To do so, follow the "H" or "V" with one of the following symbols:

"-"	dashed grid
"."	dotted grid
"-."	dash-dotted grid

For example, "H-.V" asks for a dash-dotted horizontal grid, and a solid vertical grid.

These grid line values will remain unchanged until you call SetGrid again. So you can set them once, and they will be used repeatedly for a number of different charts and graphs.

Exception:

113 No such SetGrid direction: xxx

AskGrid (s$)

AskGrid returns the current values of the grid line values. The resulting string *s$* is always in uppercase. It can be:

""	all grid lines off
"H"	horizontal grids on, vertical off
"V"	vertical grids on, horizontal off
"HV"	both horizontal and vertical grids on

Either the "H" or "V" can be followed by "-", ".", or "-." to indicate dashed, dotted, or dash-dotted grid lines respectively.

DrawGrid (style$)

DrawGrid draws grid lines on the canvas. Usually you ask for grid lines before you graph data – then the data display is drawn over the grids.

However, you can also draw grids after you've drawn your data. These grids will go over the data. The *style$* string is as in SetGrid and becomes your new grid style. The grid is drawn in your current color.

Here's a trick for charting areas on a monochrome display. Use SetGrid("h") to get solid grid lines. Then draw your chart. Then switch to the background color and draw a dotted grid:

```
call SetGrid("h")
call AreaChart(...)
set color "background"
call DrawGrid("h.")
```

This will give you a dotted grid across the canvas and a dotted grid across the filled areas.

Exceptions:

113 No such SetGrid direction: xxx
118 No canvas window yet.

The next six routines control the title and the horizontal and vertical labels. They are alternatives to the more general SetText routine, which controls all three.

SetTitle (text$)

SetTitle changes the current text for the title. This text will be used in every frame until you call SetTitle again. If you give *text$* = "", then no title is shown, and that space is added to the room for the data display.

AskTitle (text$)

AskTitle sets *text$* to be the current text of the title. If there's no current title, *text$* will be the null string.

SetHlabel (text$)

SetHlabel changes the current text for the horizontal label. This text will be used in every frame until you call SetHlabel again. If *text$* = "", no horizontal label is shown, and that space is added to room for the data.

AskHlabel (text$)

AskHlabel sets *text$* to be the current text of the horizontal label. If there's no current horizontal label, *text$* will be the null string.

SetVlabel (text$)

SetVlabel changes the current text for the vertical label. This text will be used in every frame until you call SetVlabel again. If *text$* = "", no vertical label is shown, and that space is added to room for the data.

AskVlabel (text$)

AskVlabel sets *text$* to be the current text of the vertical label. If there's no current vertical label, *text$* is the null string.

SetTitleColor (c$)

SetTitleColor changes the title's current color. This color will remain intact until you call SetTitleColor again, or until you call a high-level routine that takes a color scheme as an argument.

The color can be any True BASIC color name, or it can be a number such as "1", "7", and so forth. These numbers are converted to the corresponding True BASIC color number.

NOTE: You cannot use this routine with the higher-level routines in the *Business, Scientific,* or *Statistics Graphics Toolkits*, since they override your setting.

AskTitleColor (c$)

AskTitleColor returns the current title color. It returns the string you used to set the title color, so it may be a True BASIC color name such as "red" or a color number such as "1".

SetFrameColor (c$)

SetFrameColor changes the frame's current color. This color will remain intact until you call SetFrameColor again, or until you call a high-level routine that takes a color scheme as argument. The frame color is used for horizontal and vertical labels but not for the title.

The color can be any True BASIC color name, or it can be a number such as "1", "7", and so forth. These numbers are converted to the corresponding True BASIC color number.

NOTE: You cannot use this routine with the higher-level routines in the *Business, Scientific,* or *Statistics Graphics Toolkits*, since they override your setting.

AskFrameColor (c$)

AskFrameColor returns the current frame color. It returns the string you used to set the frame color, so it may be a True BASIC color name such as "red" or a color number such as "1".

SetCanvas (c$)

SetCanvas changes the canvas color. You may use any foreground color as the new canvas color, but be careful not to use the same color for both the canvas and the data! And if you use the same color for the frame and canvas, the axis and grid lines will not be visible.

This color will remain intact until you call SetCanvas again. The color can be any True BASIC color name, or it can be a number such as “1”, “7”, and so forth. These numbers are converted to the corresponding True BASIC color number.

AskCanvas (c$)

AskCanvas returns the current canvas color. It returns the string you used to set this color, so it may be a True BASIC color name such as “red” or a color number such as “1”.

SetHmarks (mark$())

SetHmarks changes the current horizontal marks to those given in the *mark$*() array. It does not check to make sure that these marks fit in the current window; that is done when you call DrawFrame.

NOTE: You cannot use this routine with the higher-level routines in the *Business, Scientific* or *Statistics Graphics Toolkits*, since they override your setting.

AskHmarks (mark$())

AskHmarks returns the current horizontal marks, in order, in the *mark$*() array. If there are no horizontal marks at present, it returns an array with no elements. The lower bound of *mark$*() is always 1.

SetVmarks (mark$())

SetVmarks changes the current vertical marks to those given in the *mark$*() array. It does not check to make sure that these marks fit in the current window; that is done when you call DrawFrame.

NOTE: You cannot use this routine with the higher-level routines in the *Business, Scientific* or *Statistics Graphics Toolkits*, since they override your setting.

AskVmarks (mark$())

AskVmarks returns the current vertical marks, in order, in the *mark$*() array. If there are no vertical marks at present, it returns an array with no elements. The lower bound of *mark$*() is always 1.

SetHRange (low, high)

SetHrange sets the current horizontal range to run from *low* to *high* numeric values. It does not decide how to divide the range into intervals; that is done when you call DrawFrame.

It does pick a nice interval that encloses your *low* and *high* values. If you call SetHRange(1.1,2.7) for instance, it will use the nicer range 1 to 3. A subsequent call to AskHrange will return the nice version – not necessarily the range you gave. (And DrawFrame may tamper with this range a second time in its efforts to get numeric marks that fit in the space provided!)

NOTE: You cannot use this routine with the higher-level routines in the *Business, Scientific* or *Statistics Graphics Toolkits*, since they override your setting. Use SetXscale instead.

AskHrange (low, high)

AskHrange returns the current horizontal range. The low end is given in *low* and the high end in *high*. If the horizontal marks are textual, instead of a numeric range, it returns *low* = 1 and *high* = the number of marks.

SetVRange (low, high)

SetVrange sets the current vertical range to run from *low* to *high* numeric values. It does not decide how to divide the range into intervals; that is done when you call DrawFrame. As with SetHRange, the range finally used may not be the same as the range you passed in *low* and *high*.

NOTE: You cannot use this routine with the higher-level routines in the *Business, Scientific* or *Statistics Graphics Toolkits*, since they override your setting. Use SetYscale instead.

AskVrange (low, high)

AskVrange returns the current vertical range. The low end is given in *low* and the high end in *high*. If the vertical marks are textual, instead of a numeric range, it returns *low* = 1 and *high* = the number of marks.

SetCanvasCoords

SetCanvasCoords sets the canvas' window coordinates to match the current horizontal and vertical ranges and mark indentations. (SetHrange and SetVrange control the ranges, and SetMarkIndent controls the mark indentation.)

SetHmarkLen (n)

SetHmarkLen sets the maximum number of characters used to display a numeric horizontal mark. (Remember that a mark is a textual label.) By default, this number is 9. If you set it to less than 1, this routine will cause an error. If you set it to show more digits than your computer can handle, you will get meaningless results in the extra digits.

Exception:

120 Can't use less than 1 character for mark: n

AskHmarkLen (n)

AskHmarkLen returns the current maximum number of characters used when showing a numeric horizontal tick mark.

SetVmarkLen (n)

SetVmarkLen sets the maximum number of characters used to display a numeric vertical mark. (Remember that a mark is a textual label.) By default, this number is 9. If you set it to less than 1, this routine will cause an error. If you set it to show more digits than your computer can handle, you will get meaningless results in the extra digits.

Exception:

120 Can't use less than 1 character for mark: n

AskVmarkLen (n)

AskVmarkLen returns the current maximum number of characters used when showing a numeric vertical tick mark.

SetGridStyle (h, v)

SetGridStyle is an alternate form of SetGrid. It controls the appearance of the horizontal and vertical grid lines. The horizontal and vertical line styles are given by *h* and *v* respectively. Each of these must be a number in the range 0 to 4. (Styles 2 to 4 are significantly slower than style 1.)

Style	Meaning
0	no grid line
1	solid line
2	dashed line
3	dotted line
4	dash-dotted line

Exceptions:

116 Grid styles must be in range 0 to 4: h,v

AskGridStyle (h, v)

AskGridStyle returns the current line styles for the horizontal and vertical grid lines in *h* and *v* respectively.

SetAspect (r)

SetAspect changes the number used to compensate for the aspect ratio of your computer's screen. As you remember, most monitors don't show pixels as squares – they're usually somewhat taller than they are wide.

The Toolkit, however, tries to compensate for this whenever it draws circles. Therefore if you're using a standard monitor, circles should look circular on your screen.

If not, you can call SetAspect to adjust the internal aspect ratio. Your new ratio, *r*, will then control circles. A larger ratio will make circles wider. You will probably have to use trial and experiment to find the right value for your computer. Ratios generally lie in the range .4 to .85 or so. To switch back to using the internal number, pass *r* = 0.

NOTE: The SetAspect routine has no effect on the *Statistics Graphics Toolkit*. Its description is included here only for completeness.

AskAspect (r)

AskAspect returns the current ratio *r* used to compensate for your computer monitor's aspect ratio. This number varies, depending on the kind of computer you're using and what mode it's in.

NOTE: Switch to the right graphics mode before you call AskAspect – it returns the aspect ratio for your current mode.

If you've used SetAspect to establish the ratio, AskAspect will return whatever you've said.

SetMargins (left, right, bottom, top)

SetMargins controls the pixel margins around the frame. These margins are the unused space between the edge of the window and the edge of the frame. You can use this to change the positioning of graphs within a window.

The parameters control the *left, right, bottom* and *top* margins respectively. These new margins will stay in effect until your program stops, or you call SetMargins again.

By default, all margins are 0. Therefore the entire current window is used for the frame. But if you call SetMargins(1,2,3,4), for example, then the frame will be drawn a little bit smaller. There will be 1 unused pixel on the left side of the window, 2 on the right side, 3 on the bottom, and 4 on the top.

AskMargins (left, right, bottom, top)

AskMargins returns the current frame pixel margins in *left, right, bottom* and *top*.

SetTickSizes (h, v)

SetTickSizes sets the current pixel sizes of the horizontal and vertical ticks. These sizes are used both for the ticks drawn along the frame and for the ticks drawn on axes. Tick sizes less than 1 pixel mean that no ticks are drawn.

AskTickSizes (h, v)

AskTickSizes returns the current sizes, in pixels, of the horizontal and vertical ticks.

SetTickStrides (h, v)

SetTickStrides sets the distance between ticks (in canvas window coordinates). It changes the current distances (strides) for both horizontal and vertical tick marks.

NOTE: You cannot use this routine to control the ticks in the *Business, Scientific,* and *Statistics Graphics Toolkits* since these Toolkits set their own tick strides.

This routine does have one use with the Toolkits, however. You can call SetTickStrides after drawing a chart, but before calling DrawGrid. It will control the spacing of the grid lines drawn by DrawGrid. Don't pass zero or negative numbers for strides, or DrawGrid will go into an infinite loop.

AskTickStrides (h, v)

AskTickStride returns the current distances between horizontal and vertical ticks. These numbers are measured in canvas window coordinates.

SetMarkIndent (h, v)

SetMarkIndent changes the frame's current mark indenting for both the horizontal and vertical ticks along the frame.

A mark indentation of 0 means that the first and last ticks are flush against the edge of the canvas. A non-zero setting indents these first and last ticks somewhat from the edge. Some graphs look better one way, and some look better the other!

NOTE: You cannot use this routine with the higher-level routines in the *Business, Scientific* or *Statistics Graphics Toolkits*, since they override your setting.

AskMarkIndent (h, v)

AskMarkIndent returns the current mark indenting for the horizontal and vertical marks along the frame.

AskPixel (dx, dy)

AskPixel returns the size of a pixel in the canvas in terms of the canvas window coordinates.

It's handy if you're drawing precise figures in the graphing region. First call AskPixel to find *dx* and *dy*. Then, for example, you can plot a line 10**dx* units long horizontally, to give a line 10 pixels long.

DrawFrame

After you've set up all the frame parameters by calling preceding routines, you call DrawFrame to actually draw the frame.

DrawFrame does a fair amount of computation. It will, for instance, compute all the numeric tick marks, position ticks, calculate the canvas window coordinates, and so forth. It will go so far as to change the horizontal and vertical ranges in order to get marks that fit in the room given.

NOTE: You cannot use DrawFrame with the higher-level routines in the *Business, Scientific,* or *Statistics Graphics Toolkits*, since they call DrawFrame themselves!

Error Messages

This section lists the error messages that you can get when using the *Statistics Graphics Toolkit*. In general, it should be clear how to fix an error. Please see the "Trouble-Shooting Graphics" section for help with the trickier problems.

The following errors are given by the FRAMELIB routines, and hence are the same for the *Statistics, Scientific,* and *Business Graphics Toolkits*.

100 Graph's title is too wide.
101 Graph's legend is too wide.
102 Graph's horizontal label is too wide.
103 Graph's vertical label is too long.
104 Need more room for graph's vertical marks.
105 Need more room for graph's horizontal marks.
106 Need greater width for graph.
107 Need greater height for graph.
108 Vertical marks aren't wide enough–use SetVmarkLen.
109 Horizontal marks aren't wide enough–use SetHmarkLen.
110 Data arrays have different bounds in *routine*.
111 Data and unit arrays don't match for *routine*.
112 Data and legend arrays don't match for *routine*.
113 No such SetGrid direction: *xxx*
114 Grid styles must be in range 0 to 4: *h,v*
115 Unknown line style: *n*
116 Unknown point style: *n*
117 Can't handle this graph range: *low* to *high*
118 No canvas window yet.
119 No legend window yet.
120 Can't use less than 1 character for mark: *n*
11008 No such color: *xxx*

Errors shared with the *Scientific Graphics Toolkit*:

150 No such scientific graph type: *xxx*
152 Grain must be in range 10 to 1000: *n*
159 Log-X scale can't have numbers <=0: *low* to *high*
160 Log-Y scale can't have numbers <=0: *low* to *high*

Errors specific to the *Statistics Graphics Toolkit*:

701 Bad line style: *n*

702 SetLineFit must be LS or MEDIAN: *xxx*

703 SetHist must be COUNT, REL, or %: *xxx*

704 SetObsSD can't be negative: *n*

705 Data point style out of range: *n*

706 Can't take geometric mean of non-positive numbers.

707 Can't take harmonic mean of data that includes 0.

708 Can't use Skew1 with SD = 0.

709 Skew1 needs data with exactly one mode.

710 Can't use Skew2 with SD = 0.

711 Quartiles 1 and 3 equal in SkewQ.

712 Percentiles 10 and 90 equal in SkewP.

713 Can't use SkewM with SD = 0.

714 Can't use KurtosisM with SD = 0.

715 Can't use KurtosisP with percentiles 10 and 90 equal.

716 Moment 'r' must be positive integer: *r*

717 Can't use LSFit with SSX, SSY, or SE = zero.

718 Number of samples must be > 0.

719 *Parameter* must be >= 0: *n*

720 *Parameter* must be > 0: *n*

721 p in SimFreqBinomial must be inside (0,1).

722 Can't have 0 step size in sequence.

723 Can't have zero values in sequence.

724 Left endpoint must be < than right in *routine*.

725 Sample size > population size in SampleNoRep: *n*

726 Interval and count tables don't match in TableToFreq.

727 Can't use FreqToRelFreq with no data elements.

728 Need >= 2 observations for a confidence interval.

729 Need >= 2 obs. in each dataset for a two-sample t-test.

730 Need >= 2 observations for a t-test.

731 Need >= 2 obs. in each dataset for a two-sample t-test.

732 Can't do contingency table with 0 row.

733 Can't do contingency table with 0 column.

734 *xxx* transform needs to create missing value.

735 Can't convert to number or missing value: *xxx*

736 Not enough data for *routine*.
737 DF must be integer >= 1 for *routine*.
738 Probability must be in range 0 to 1 inclusive for *routine*.
739 Probability must be in range 0 to 1 exclusive for *routine*.
740 Confidence coefficient must be in range 0 to 1 inclusive for *routine*.
741 Polynomial degree must be positive integer in *routine*: *n*
742 New row doesn't match array's shape in AppendRow.
743 New column doesn't match array's shape in AppendCol.
744 Not enough data for a box plot.
745 PolyLSFit: *n* coefficients for *d* data points.
746 Can't use polar graphs with Statistics Graphics Toolkit.
747 Data and label arrays don't match in *routine*.
748 Data and name arrays don't match in *routine*.
749 Can't compute F-test for these datasets.
750 Must have at least 2 observations in Kendall W.
751 Kendall W data must be integers.
752 Kendall W observations must be in range 1 to *n*.
753 Kendall W undefined for this data.
754 Bad critical value in GetCI.
755 Too many identical values for Kruskal-Wallis H test.
756 Too few observations in MultiLSFit.
757 Dependent variable must be integer between *i* and *j*: *n*
758 Singular matrix in MultiLSFit.
759 *routine* dataset can't contain missing values.
760 *routine* works on 2x2 tables only.
761 Need at least 99 data items for Percentiles.
762 Need at least 3 data items for Quartiles.