MASTER

FINAL PROGRESS REPORT

EY-76-S-02-2310 A003

Period March 1973 through November 1979

"Research on Data Analysis in the Physical Sciences"

John W. Tukey, Professor
Co-Principal Investigator

Peter Bloomfield, Professor
Co-Principal Investigator

Department of Statistics
Princeton University
Princeton, New Jersey 08544

DISCLAIMER

In September, 1973, the Atomic Energy Commission awarded contract number AT(11-1)-2310 to the Department of Statistics, Princeton University, with the title Data Analysis in the Physical Sciences. The contract, which was renewed in turn by the Energy Research and Development Administration and the Department of Energy, has promoted a broad spectrum of activities within the Department oriented towards energy problems and the overall mission of the Department of Energy. In its most general terms, the work carried out under the contract consists of the development of new data analytic methods and the improvement of existing methods, their implementation on computer, especially mini-computers, and the development of non-statistical, systems-level software to support these activities. The association with the Department of Energy has also led to interaction between members of the Department and personnel from other DOE - supported institutions such as the National Laboratories, both through the Annual Statistical Symposium and through visits to individual sites. The data analysis developments were also of great importance in the execution of a separate contract awarded to the Department of Statistics by the Energy Information Administration of DOE, for evaluation of oil and gas resource estimates.

We shall review briefly the work reported or completed under the following general heads:

1) Computer facilitation of data analysis

2) Fitting, regression, residuals

3) Techniques stimulated by specific areas of application

4) General techniques of statistics and data analysis

5) Miscellaneous

1. Computer facilitation of data analysis

In December, 1974, the Department installed a Digital Equipment Corporation PDP-11/40 mini-computer, purchased in part with contract funds and in part with other funds raised by the Department. The objective was to explore the extent to which such a small, relatively inexpensive computing facility could be used for interactive data analysis, and the scope of the tasks that it could support. In March, 1975, the Unix operating system was installed, as it was then the only operating system to provide support for flexible multi-user timeshared operation of machines of the size of the PDP 11/40. The current efforts elsewhere to make Unix available for larger computers of other manufacturers underscore the value of that decision, since the software

developed under the contract will be compatible with a wide range of machines.

The first major piece of software to be developed was a Fortran language system to replace the inadequate support then provided by Unix. The Fortran 77 compiler provided with Version 7 of Unix will now replace both earlier compilers. However, in the interim the Princeton Fortran System has been in great demand. We have distributed it or arranged its distribution to 59 installations, listed in Appendix E, and some of these have acted as further distribution centers.

The second major software development was an interactive statistical processor (known as Isp), for which the Fortran system was indispensable. This processor incorporates many graphical features to aid in data analysis, some conventional methods and some of the newly developed techniques. It is now in use at other Unix installations, listed in Appendix F. The documents provided for Isp consist of a user's manual and two tutorial introductions, which are reproduced in Appendix G.

2. Fitting, regression, residuals

Fitting of regressions. Least-absolute deviation fitting was reviewed by Bloomfield and Steiger (TR 137).

Multiple regression. The use of higher moments to establish more nearly causal connections was investigated by Mayor de Montricher (TR 63).

Polynomial regression. Behavior near the ends of a polynomial fit was examined by Mayor de Montricher (TR 64) and by Mayor de Montricher and Tukey (TR 66).

Regression design. Aspects of optimum design for regression were investigated by Silvey (TR 75).

Regression generally. A variety of aspects, many inadequately known, were set out in a book by Mosteller and Tukey (1977).

Residuals. Distributions of residuals from regression were considered by Bloomfield (TR 56) and Horn (TR 155). See also work above under "polynomial regression".

Robust/resistant smoothing. Brief accounts were published by Tukey (1976) and Velleman (1977).

Splines. The use of splines in data analysis was reported by Anderssen, Bloomfield and McNeil (TR 69). Their

application to a specific problem in interpolation was considered by McNeil and Turner (TR 79), and published by McNeil, Trussel and Turner (1977).

3. Techniques stimulated by specific areas of application

Biomedical problems (including demography). The problems of multiplicity in clinical trials were reviewed by Tukey (1977). Topics in demography were studied by Turner (1976) McNeil and Hill (TR 109), McNeil and Kafadar (TR 111). Issues of technique in the analysis of event- related brain potentials were discussed by Tukey (1978).

Molecular structure. These problems were introduced by Tukey (1974d) discussed by him (TR 44, 1974e), and reviewed by him (1974f).

Particle physics. An algorithm for searching projections of higher-dimensional data was developed by Friedman and Tukey (TR 45, 1974). A computer-graphic system (PRIM-9) for examining higher dimensional data was developed by Fisherkeller, Friedman and Tukey (1974). Work in progress on dealing with point clouds concentrated near intermediate-dimensional manifolds was discussed by Freidman, Tukey, and Tukey (1979).

4. General techniques of statistics and data analysis

Data geometry. Affine geometric aspects of data patterns were discussed by Tukey (1975). Techniques of data mapping were criticized by Tukey (1979) and solutions to some of the criticisms were presented by Bloomfield (1979).

Display techniques. The likely advantages of alternagraphic displays were stressed by Tukey (TR 48). The higher-order diagnosis of two-way tables was studied by McNeil and Tukey (TR 65, 1975).

Estimation techniques. Problems in estimation were discussed by Watson (TR 47), McNeil and Weiss (TR 76) and Besag (1977).

General expositions. Broad questions of data analysis were treated by Tukey (TR 40), (1980).

Inequalities for maxima. Inequalities important in dealing with statistics defined as maxima were studied by Hunter (TR 73, TR 88, 1976).

Interactive data analysis. Applications were discussed

by McNeil and by McNeil and Tukey (TR 65). See also PRIM-9 under "particle physics" in Section 3.

Measures of dependence. The monotone coefficient, important in relating one ordinal and one measured variate, was studied by Robinson (TR 70) and by Mayer and Robinson (TR 72, TR 95, TR 133). A Multivariate generalization of Kendall's Tau was proposed by Simon (TR 78).

Multidimensional techniques. (See also last item above). Questions of multidimensional scaling were discussed by Simon (TR 62). Questions of 2-sample, 2-dimensional significance were discussed by Delfiner (TR 110).

Robust/resistant. An exposition of the current state of the art was made by Tukey (1975). The techniques needed for effective Monte Carlo Study of rob/res techniques were discussed by Simon (TR 91). Some points in the theory were treated by Mayor de Montricher (TR 61). See also "robust/resistant smoothing" in section 2, above.

Spectrum analysis, etc. Questions of multiple spectrum analysis were discussed by Bloomfield and Zlotnick (TR 121). Questions of complex demodulation were discussed by Bloomfield and Stine (TR 144). The relation of spectrum analysis to regression was treated by Watson (TR 46). The general problems of which technique to use where and when were treated by Tukey (1978). Testing independence in random fields was discussed by Kazim (TR 68).

5. Miscellaneous

The behaviour of the Phillips phenomenon was discussed by Simon (TR 43).

Data describing the performance of a large batch-processing computer were analyzed and modelled by Turner (TR 81).

APPENDIX A: Personnel

Personnel partially supported by AEC, ERDA and DOE contract during period March 1973 - November 1979.

April 1973 - August 1974

Principal Investigators

J. W. Tukey
G. S. Watson

Faculty Supported:

P. Bloomfield - July, August 1973; 50%.
J. Robinson, July, August 1974; 100%.
G. Simon - July, August 1974; 100%.
G. S. Watson - July, August 1973; 50%.
D. McNeil, August 1974; 100%.

Visitor:

C. F. Mayor de Montricher, April 1973 thru May, 1974; 100%.

Graduate Students:

N. Cressie - July, August 1974; 50%
D. Hunter, July 1973, June 1974, July, August 1974; 100%.
F. Kazim, July, August 1974; 100%.
J. Turner, July, August 1974; 100%>
P. Velleman, July August 1974; 50%

September 1974 - August 1975

Principal Investigators:

J. W. Tukey
G. S. Watson

Faculty Supported:

G. Simon, July, August 1975;100%.

Visitors:

J. Besag, February thru June 1975; 25%.
July, August 1975; 100%.
S. D. Silvey, September, October 1975;
November, December 1975; 25%.

Graduate Students:

P. Delfiner, September thru June 1975; 100%.
D. Hunter, September thru June 1975; 100%
K. Lin, September 1975; 100%, November thru June 1975; 50%.
J. Turner, September 1975; 100%, November thru August 1975; 50%.
H. Zlotnik, September 1975, November thru August 1975; 50%.

September 1975 - August 1976

Principal Investigators:

J. W. Tukey
G. S. Watson

Faculty Supported:

None

Visitor:

R. Anderssen, September thru December 1975; 50%.

Graduate Students:

S. Arthur, August 1976; 100%.
B. Carney, September 1975 thru April 1976; 50%.
K. Kafadar, September 1975 thru April 1976; 50%.
F. Kazim, August 1976; 100%.
D. Mohr, october 1975 thru April 1976; 50%.
J. Turner, September, October; 50%.
December thru April 1976; 30%, July 1976; 100%.
D. Tyler, August 1976; 100%.
A. Silverberg, July, August 1976; 100%.

September 1976 - August 1977

Principal Investigators:

J. W. Tukey
P. Bloomfield

Faculty Supported:

None

Graduate Students:

S. Arthur, September thru January; 100%.
B. Guarino, October thru February; 50%.
K. Kafadar, September thru May; 50%.
C. Lancaster, September thru May; 50%.
A. Silverberg, September thru May; 50%.
D. Tyler, April, May; 100%.
T. Watt, September thru January; 100%.
February thru May; 50%.

September 1977 - August 1978

Principal Investigators:

J. W. Tukey
P. Bloomfield

Faculty Supported:

P. Bloomfield, July; 100%.
G. S. Watson, July; 100%.

Graduate Students:

K. Bell, September thru June; 50%.
J. Gosling, September thru June; 50%.
D. Coleman, September thru February; 100%.
R. Guarino, September thru June; 50%.
P. Horn, November thru June; 50%.
A. Houtman, September thru June; 50%.
K. Kafadar, September thru June; 50%.
L. Kates, September thru June; 50%. July, August; 100%.
C. Lancaster, September thru January; 50%.
A. Silverberg, September thru June; 50%.
A. Wilks, September thru June; 50%.
P. Zulke, July, August; 100%.

September 1978 - November 1979

Principal Investigators:

J. W. Tukey
P. Bloomfield

Faculty Supported:

P. Bloomfield, July; 100%.
J. W. Tukey, July, August; 100%.

Visitor:

W.Steiger, August; 100%.

Graduate Students:

K. Beltrao, July, August; 100%.
Y. Benjamini, July, August; 50%.
September thru November; 100%.
R. Guarino, July, August; 100%.
P. Horn, September thru June 1978; 50%.
September thru November 1979; 100%.
A.Silverberg, September thru June; 100%.
A. Wilks, September thru November 1978,
July thru November 1979; 100%.
P. Zulke, September thru June 1979,
September thru November 1979; 100%.

APPENDIX B: Publications

Reprints prepared in connection with Department of Energy Contract EY-76-S-02-2310

Besag, J., "Errors-in-variables estimation for Gaussian lattice schemes," Journal of the Royal Statistical Society, Series B, Vol. 39, No. 1, 1977, pp. 73-78.

Bloomfield, P. "An interactive statistical processor for the Unix time-sharing system," Computer Science and Statistics: Tenth Annual Symposium on the Interface, eds D. Hogben, D. W. Fife, NBS Spec. Publ. 503, 2-8, March 1978.

Bloomfield, P., "Analysis of cancer mortality in Texas," Proceedings of the 1976 Workshop on Automated Cartography and Epidemilogy, March 18-19, 1976, Arlington, Virginia, DHEW Publication No. (PHS) 79-1254, 18-26, August 1979.

Hunter, D., "An upper bound for the probability of a union," J. Appl. Prob., Vol. 13, 1976, pp. 597-603.

McNeil, D., Interactive Data Analysis: A Practical Primer, John Wiley & Sons Publishing Company, New York.

McNeil, D., Trussell, T. J., Turner, J. C., "Spline interpolation of demographic data," Demography, Vol. 14, Number 2, May 1977, pp. 245-252.

McNeil, D., Tukey, J. W., "Higher order diagnois of two-way tables, illustrated on two sets of demographic empirical distributions," Biometrics Vol. 31, no. 2, June 1975, pp. 487-510.

Tukey, J. W., Friedman, J. H., "A projection pursuit algorithm for exploratory data analysis," IEEE Transactions on Computers, Vol. C-23 No. 9, 881-890, 1974.

Tukey, J. W. "Introduction to todays's data analysis," Critical Evaluation of Chemical and Physical Structural Information, D. R. Lide, Jr., and M.A. Paul, eds, National Academy of Sciences, Washington, D. C. 3-14, 1974.

Tukey, J. W. "Data analysis for molecular structure: today and tomorrow," Critical Evaluation of Chemical and Physical Structural Information, D.R. Lide, Jr., and M.A. Paul, eds., National Academy of Sciences, Washington, D.C. 48-58

Tukey, J. W., "Instead of Gauss-Markov least squares, what?," Applied Statistics, (Proceedings of the Conference at Dalhousie University, Halifax, May 2-4, 1974), R. P. Gupta, ed., North-Holland Publishing Company, 351-372, 1975.

Tukey, J. W., "Mathematics and the picturing of data," Proceedings of the International Congress of Mathematicians, Vancouver, Canada, August 21-29, 1974, Vol. 2, 523-531, 1975.

Tukey, J. W., "Usable resistant/robust techniques of analysis," Proceedings of the First ERDA Statistical Symposium, Los Alamos, New Mexico, November 3-5, 1975, eds., Wesley L. Nicholson and Judith L. Harris, Battelle Pacific Northwest Laboratories, Washington, 1-31, 1976.

Tukey, J. W., Fisherkeller, M.A. and Friedman, J.H. "PRIM-9: An interactive multidimensional data display and analysis system," Proceedings of the 4th International Congress for Stereology, Sept. 4-9, 1975, Gaithersburg, Maryland.

Tukey, J. W., "Robust/resistant smoothing," Selected Examples of ERDA Research in the Mathematical and Computer Sciences, National Technical Information Service, Springfield, Virginia, ERDA 76-118, 80-82.

Tukey, J. W., "Modern statistical techniques in Data Analysis," Proceedings of the Fifth Biennial International Codata Conferece, (University of Colorado June 28-July 1, 1976) ed. D. Dreyfus, Pergamon Press, Oxford and New York, 1977.

Tukey, J. W. "Some thoughts on clinical trials, especially problems of multiplicity," Science, Vol. 198: 679-684 November, 1977.

Tukey, J.W., Mosteller, F., Data Analysis on Regression: A Second Course in Statistics, Addison -Wesley Publishing Co., Reading, Massachusetts, 1977.

Tukey, J.W., McGill, R. and Larsen, W.A., "Variations of box plots," American Statistician, Vol. 32 No. 1: 12-16, February, 1978.

Tukey, J. W. Panel Discussion: Proceedings of the 1977 DOE Statistical Symposium, Pacific Northwest Laboratories Richland, WA., October 26-28, 1977, ed. Donald A. Gardiner, Tykey Truett, National Technical Information Service, Springfield, Va., 175-183.

Tukey, J. W. "A data analyst's comments on a variety of points and issues," Event Related Brain Potentials in Man, ed. Academic Press, New York, 139-154. 1978.

Tukey, J. W. "Comment on H. L. Gray, et al, "A new approach to ARMA modeling," Commun. Statist. Simula. Computa., Vol. B7, No. 1, 79-84.

Tukey, J. W., "When should which spectrum approach be used?," presented at Time Series Analysis meeting, Kings College, Cambridge, England July 12-15, 1978, The Statistician double issue September/December 1978.

Tukey, J. W., "We need both exploratory and confirmatory," presented at the ASA meetings, Session: The Teaching of Statistics, held in San Diego, CA, August 15, 1978, The American Statistician. (To appear 1980).

Tukey, J. W., "Statistical Mapping: What should not be plotted," Proceedings of the 1976 Workshop on Automated Cartography and Epidemiology, March 18-19, 1976, Arlington, Virginia DHEW Publication No. (PHS) 79-1254, August 1979, 18-26.

Tukey, J.W., Friedman, J.H. and Tukey,P.A., "Approaches to analysis of data that concentrate near higher-dimensional manifolds," Preproceedings Second International Symposim on Data Analysis and Informatics, IRIA, Versailles, France, 17-19 October, 1-7.

Turner, J. C., "Calculation of roots of Lotka's equation," Theoretical Population Biology, Vol. 9, No. 2, April 1976, 222-237.

Velleman, Paul F., "Robust nonlinear data smoothers: definitions and recommendations," Proc. Natl. Acad. Sci Vol. 74, No. 2, February 1977, 434-436.

Watson, G. S., "A simple treatment of spectrum estimation and its relation to regression with stationary errors," Methoden und Verfahren der mathematischen Physik, 14, pp. 149-160, 1975.

Watson, G. S. "Estimation when the errors are approximately independent and Gaussian, Methoden und Verfahren der mathematischen Physik, 14, pp. 161-170, 1975.

APPENDIX C: Technical Reports

Technical Reports prepared under DOE Contract EY-76-S-02-2310 for period April 1973 through November 1979

Graduate Students:

Delfiner, P., "A spatial two-sample test," Tech. Report #110, series 2, Department of Statistics, Princeton University, Princeton, N. J. October 1975.

Horn, P., "Heteroscedasticity of residuals: a non-parametric alternative to the Goldfeld-Quandt peak test," Tech. Report #155, series 2, Department of Statistics, Princeton University, Princeton, N. J. August 1979.

Hunter, D., "Bounds for the probability of a union," Tech. Report #73, series 2, Department of Statistics, Princeton University, Princeton, N. J. December 1974.

Hunter, D., "Approximating percentage points of statistics expressible as maxima, Tech. Report #88, series 2, Department of Statistics, Princeton University, Princeton, N. J. June 1975.

Kafadar, K., McNeil, D. R., "On the distribution of Fecundability," Tech. Report #111, series 2, Department of Statistics, Princeton University, Princeton, N. J. June 1976.

Kazim, F., "The distribution of a criterion for testing temporal independence in random fields," Tech. Report #68, series 2, Department of Statistics, Princeton University, Princeton, N. J. October 1974.

Stine, R., Bloomfield, P., "Complex demodulation of Quasiperiodic time series," Tech. Report #144, series 2, Department of Statistics, Princeton University, Princeton, N. J. March 1979.

Turner, J., "Preliminary analysis of Princeton University's computer system data," Tech. Report #81, series 2, Department of Statistics, Princeton University, Princeton, N. J. March 1975.

Turner, J. McNeil, D.R., "Using splines to interpolate age-specific fertility," Tech Report #79, series 2, Department of Statistics, Princeton University, Princeton, N. J. February 1975.

Zlotnik, H., Bloomfield, P., "Spectral Analysis of multiple time series data," Tech. Report #121, series 2, Department of Statistics, Princeton University, Princeton, N. J. March 1977.

Faculty:

Bloomfield, P.,"Note on a central limit theorem of I. A. Ibragimov, with an application to the sojourn times of the M/G/infinity queue," Tech. Report #41, series 2, Department of Statistics, Princeton University Princeton, N. J. July 1973.

Bloomfield, P., "On the distribution of the residuals from a fitted linear model," Tech. Report #56, series 2, Department of Statistics, Princeton University, Princeton, N. J. January 1974.

Bloomfield, P., Anderssen, R.S., McNeil, D.R., "Spline functions in data analysis " Tech. Report #69, series 2, Department of Statistics, Princeton University, Princeton, N. J. October 1974.

Bloomfield, P., "Analysis of cancer mortality in Texas," Tech. Report #117, series 2, Department of Statistics, Princeton University, Princeton, N. J. August 1976.

Bloomfield, P., Zlotnik, H., "Spectrum analysis of multiple time series data," Tech. Report #121, series 2, Department of Statistics, Princeton University, Princeton, N. J. March 1977.

Bloomfield, P., "An interactive statistical processor for the Unix Time-Sharing System," Tech. Report #125, series 2, Department of Statistics, Princeton University, Princeton, N. J. July 1977.

Bloomfield, P., Steiger, W., "Least Absolute Deviations Curve-Fitting," Tech. Report #137, series 2, Department of Statistics, Princeton University, Princeton, N.J. September 1977, revised September 1979.

Bloomfield, P., Stine, R., "Complex demodulation of Quasiperiodic time series," Tech. Report #144, series 2, Department of Statistics, Princeton University, Princeton, N. J. March 1979.

McNeil, D.R., "Interactive exploratory data analysis using A Programming Language," Tech. Report #54, series 2, Department of Statistics, Princeton University, Princeton, N. J. February 1974.

McNeil, D.R., Tukey, J.W., "Demographic data analysis using an interactive computer," Tech. Report #65, series 2, Department of Statistics, Princeton University, Princeton, N. J. October 1974.

McNeil, D.R., Bloomfield, P., Anderssen, R.S., "Spline functions in data analysis," Tech. Report #69, series 2, Department of Statistics, Princeton University, Princeton, N. J. October 1974.

McNeil, D.R., Weiss, G.H., "A large sample approach to estimation of parameters in Markov population models," Tech. Report #76, series 2, Department of Statistics, Princeton University, Princeton, N. J. January 1975.

McNeil, D. R., Turner, J., "Using splines to interpolate age-specific fertility," Tech. Report #79, series 2, Department of Statistics, Princeton University, Princeton, N. J. February 1975.

McNeil, D.R., Hill, A., "Geographic variations in U. S. fertility by state," Tech. Report #109, series 2, Department of Statistics, Princeton University, Princeton, N. J. March 1976.

McNeil, D.R., Kafadar, K., "On the distribution of fecundability," Tech. Report #111, series 2, Department of Statistics, Princeton University, Princeton, N. J. June 1976.

Simon, G., "Another look at the Phillips phenomenon," Tech. Report #43, series 2, Department of Statistics, Princeton University, Princeton, N. J. August 1973.

Simon, G., "An explicit relationship between multidimensional scaling and principal components - use of nonmetric multidimensional scaling to find principal components," Tech. Reprot #62, series 2, Department of Statistics, Princeton University, Princeton, N. J. June 1974.

Simon, G., "Multivariate genralization of Kendall's Tau," Tech. Report #78, series 2, Department of Statistics, Princeton University, Princeton, N. J. January 1975.

Simon, G., "A non-parametric test of total independence based on Kendal's Tau," Tech. Report #90, series 2, Department of Statistics, Princeton University, Princeton, N. J. August 1975.

Simon, G., "Swindles to improve computer simulation, with applications to the problems of appraising estimates of location and dispersion in univariate samples," Tech. Report #91, series 2, Department of Statistics, Princeton University, Princeton, N. J. August 1975.

Tukey, J. W., "Introduction to today's data analysis," Tech. Report #40, series 2, Department of Statistics, Princeton University, Princton, N. J. June 1973.

Tukey, J. W. "Data analysis for molecular structure: today and tomorrow," Tech. Report #44, series 2, Department of Statistics, Princeton University, Princeton, N. J. June 1973.

Tukey, J.W., Friedman, J.H., "A projection pursuit algorithm for exploratory data analysis," Tech. Report #45, series 2, Department of Statistics, Princeton University, Princeton, N. J. June 1973.

Tukey, J.W., "Some thoughts on alternagraphic displays," Tech. Report #48, series 2, Department of Statistics, Princeton University, Princeton, N. J. October 1973.

Tukey, J.W., McNeil, D.R., "Demographic data analysis using an interactive computer," Tech. Report #65, series 2, Department of Statistics, Princeton University, Princeton, N. J. July 1974.

Tukey, J. W., Mayor deMontricher, G., "Polynomial fitting near the ends of equally spaced data; the empirical facts," Tech. Report #66, series 2, Department of Statistics, Princeton University, Princton, N. J. July 1974.

Watson, G., "A simple treatment of spectrum estimation and its relation to regression with stationary errors, Tech. Report #46, series 2, Department of Statistics, Princeton University, Princeton, N. J. September 1973.

Watson, G., "Estimation when the errors are approximately independent and Gaussian, Tech. Report #47, series 2, Department of Statistics, Princeton University, Princeton, N. J. September 1973.

Visitors:

Anderssen. P., Bloomfield, P., McNeil, D.R., "Spline functions in data analysis," Tech. Report #69, series 2, Department of Statistics, Princeton University, Princeton, N. J.

Mayor deMontricher, G., Martin, R.D., "Locally max-min robust tests," Tech. Report #61, series 2, Department of Statistics, Princeton University, Princeton, N. J. May 1974.

Mayor deMontricher, G., "Proof of contribution by multiple regression: use of the third moments to eliminate the indetermination caused by noise in the carriers," Tech. Report #63, series 2, Department of Statistics, Princeton University, Princeton, N. J. June 1974.

Mayor deMontricher, G. "Polynomial fitting; asymptotic aspect of the variance of the residuals," Tech. Report #64, Department of Statistics, Princeton University, Princeton, N. J. June 1974.

Mayor deMontricher, G, Tukey, J. W., "Polynomial fitting near the ends of equally spaced data; the empircal facts," Tech. Report #66, series 2, Department of Statistics, Princeton University Princeton, N. J. July 1974.

Robinson, J., "On the consistency of the monotone coefficient," Tech. Report #70, series 2, Department of Statistics, Princeton University, Princeton N. J. October 1974.

Robinson, J., Mayer, L.S., "The monotone coefficient as an estimator of the correlation ration between an ordinal variable and interval variable," Tech. Report #72, series 2, Department of Statistics, Princeton University, Princeton, N. J. December 1974.

Robinson J., Mayer, L.S., "Multiple and partial monotone coefficients for Regression models with ordinal predictor variables, Tech. Report #95, series 2, Department of Sttistics, Princeton University, Princeton, N. J. September 1975.

Robinson, J., Mayer, L. S., "The monotone coefficient as a measure of association for relating an ordinal variable and an interval variable in the presence of ties," Tech. Report #133, series 2, Department of Statistics, Princeton University, Princeton N. J. November 1977.

Silvey, S.D., "Some aspects of the theory of optimal linear regression design with a general concave linear function," Tech. Report #75, series 2, Department of Statistics, Princeton University, Princeton, N. J. December 1974.

Steiger, W., Bloomfield, P., "Least absolute deviations curve-fitting, Tech. Report #137, series 2, Department of Statistics, Princeton University, Princeton, N. J. September 1977.

APPENDIX D: Ph.D. Theses

Ph.D. Theses Supported by DOE Contract EY-74-76-S-02-2310

Noel Cressie (1975) - Testing for Uniformity Against a Clustering Alternative.

David Hunter (1975) - Bounds for the Probability of a Union of Random Events and Applications.

Paul Velleman (1975) - Robust Non-Linear Data Smoothers - Theory, Definitions, and Applications.

John Turner (1976) - The Statistical Analysis of Computer System Behavior.

Pierre Delfiner (1978) - Shift Invariance Under Linear Models.

Hania Zlotnik (1978) - CJohert Fertility in the United States 1901-1930.

Susan Peterson Arthur (1979) - Skew Stretched Distributions and the t Statistic.

Karen Kafadar (1979) - Robust Confidence Intervals for the One and Two Sample Problems.

Arthur Silverberg (1979) - Statistical Models for the Q-Permutations.

David Tyler (1979) - Redundancy Analysis and Associated Asymptotic Distribution Theory.

Ph.D. Theses in Progress (tentative titles).

Kathy Bell, Estimation of location using data modifications based on order-pushback techniques.

Kaizo Beltrao, "----"

Yoav Benjamin, Shapes of distributions and their affect on the tails of the t-statistic.

Roberta Guarino, Compromise estimates of location.

Anne Houtman, "----"

h.D. Theses in Progress (tentative titles) cont'd.

Paul Horn, The mid-hinge as an estimate of location.

Louis Kates, Zonal spherical functions in linear models.

Donna Mohr, Estimating the parameters for fractional Gaussian noise.

Allan Wilks, Optimal design of time arrays for unequally-spaced time-series data.

PRINCETON UNIX FORTRAN USERS:

Battelle, Frankfurt, Germany
Bell Telephone Laboratories, Murray Hill, NJ
Case Western Reserve University, Cleveland, Ohio
Columbia University, New York, NY (College of Physicians and Surgeons)
Computer Consoles, Inc., Rochester, NY
Department of the Interior, Geological Survey, Denver, Colorado
Harvard School of Public Health, Boston, Mass.
Lincoln Laboratory, Cambridge, Mass. (MIT)
Los Alamos Scientific Laboratory, Los Alamos, NM (Group C8)
Moravian College, Bethlehem, Pa. (Computer Center)
National Bureau of Standards, Gaithersburg, MD (Computer Science Section)
National Security Agency, Fort Meade, MD
New York Blood Center, NY, NY
New York Medical College, NY, NY
Polytechnic Institute of New York, Farmingdale, NY
Queen's University, Kingston, Ontario, Canada (Computer Center)
Rosemount, Inc., West Prarie, Minn.
Stanford University, Stanford, CA (Geophysics Dept.)
Suny College of Technology, Utica, NY
University of California, Berkeley, CA (Computer Science and Survey Research Center)
University of California, San Diego, CA (Chemistry Dept.)
University of California, Santa Barbara, CA
University of Aberdeen, Scotland (Computer Science)
University of Connecticut, Storrs, Conn.
University of Glasgow, Scotland (Computer Science)
University of Nottingham, England (Psychology)
University of Texas, El Paso, Texas (Electrical Engineering)
University of Toronto, Toronto, Ontario, Canada
University of Utah, Salt Lake City, Utah
Vanderbilt University, Nashville, Tenn. (Medical Center)
Yale University, New Haven, Conn. (Psychology)
University of Technology, Dept. of Computer Science (Loughborough Leicestershire, England)
Department of Defense, Fort George G. Mead, MD
U.S. Air Force Academy, DFACS/ERCC, (Colorado)
Katholieke Universiteit, (Computer Section), Netherlands
Mathematics Institute, Department of Computing Mathematics, Cardiff, Wales
University of California, Los Angeles (School of Engineering and Applied Science)
Department of Commerce, Boulder Colorado
University of Oklahoma, Norman, Oklahoma
University of Hawaii, Honolulu, Hawaii (Institute for Astronomy)
Naval Underwater Systems Center, Computer Science Dept., Newport, Rhode Island
University of Edinburgh, Edinburgh, Scotland
University of California, San Diego (Marine Physical Laboratory)
SOFTECH, Waltham, Massachusetts
Fisons Limited, Levington Research Station, England
Cetus Corporation, Berkeley, CA
Naval Ocean System Center, San Diego, CA
Tennessee Valley Authority, Muscle Shoals, Alabama 35660
Universiteit van Amsterdam, Amsterdam, Netherlands (Psychologisch Lab)

Princeton Fortan Users (cont'd.)

Rockefeller University, NY, NY
University of Maryland, College Park, MD
California Polytechnic State University, San Luis Obispo, CA
Stevens Institute of Technology, Hoboken, NJ
Procurement Directorate, Aberdeen Proving Ground, MD
Interactive Systems, Santa Monica, CA
University of Illinois Medical Center, Chicago, Illinois
RLG Associates, Reston, VA
University of Washington, Seattle, Washington
University of Bradford, Bradford, West Yorkshire, England

ISP Users:

Cetus Corporation, Berkeley, CA
Columbia University, New York, New York
Harvard School of Public Health, Boston, Mass.
Lincoln Laboratory, Cambridge, Mass.
Katholieke Universiteit, Netherlands
University of California, Berkeley, CA
University of Toronto, Toronto, Ontario, Canada
Tennessee Valley Authority, Muscle Shoals, Alabama 35660
Federal Aviation Administration, NAFEC, Pamona, NJ
SUNY College of Technology, Utica, NY
Procurement Directorate, Aberdeen Proving Ground, MD
Interactive Systems, Santa Monica, CA
RLG Associates, Reston, VA
University of Washington, Seattle, Washington
Rockefeller University, NY, NY

APPENDIX G: ISP Documents

1. A tutorial introduction to the let Command in Isp.
2. A tutorial introduction to ISP
3. ISP System Documentation

1. A Tutorial Introduction to the let Command in isp

David Donoho
Department of Statistics
Princeton University

A Tutorial Introduction to the let Command in isp

David Donoho
Department of Statistics
Princeton University

A Short manual for isp users explaining the use of the system reexpression/manipulation command, **let.**

0. Introduction

A. Purpose.

This tutorial covers usage of the **let** command in conjunction with the data structures and other tools found in **isp**. Basic familiarity with the **isp** system is supposed; see the isp Tutorial for an introduction to the whole system.

B. What let is.

let is a program in the isp system which performs arithmetic and reshaping operations commonly needed in the course of an analysis of data. **let** operates on arrays present in the isp user's active area; it understands a formulaic language that directs the reexpression of arrays already present in the active area and/or the creation of new arrays.

let is invoked in **isp** by commands like:

```
* let a = (x-y)/2; b = (x+y)/2; c = a/b;
* let wave = sin(omega * t); gauss = sqrt(poiss);
```

The general form being

```
* let expression ; expression ; ...
```

where each expression is a sentence in **let**'s language (see below for a description). **let** can also be called 'in-line' by calling a normal **isp** command with a **let expression** in

parentheses as an argument:

```
* boxplot (log(x)); stemleaf (sqrt(y+1) + sqrt(y))
* scat (1/x,log(y))
```

let facilitates data manipulation to such an extent that isp users may be drawn to explore transformation and reshaping of their data simply because it is so easy (and fun!)

C. Conventions.

let doesn't interact with the user at the keyboard level: it passively accepts commands, and then executes them. It will only type a message to the user if an error in syntax or meaning has been detected. Some examples of let error messages are

```
syntax error
b: array not square
a: not found
```

Because let is essentially 'dumb', dialog with it is no more than monolog. In the examples that follow, the fragments always represent what the user typed; we represent let's output by underscoring it. Thus:

```
* let z = 1/x
error: divide by zero
* let w = exp(z)
z: not found
*
```

To illustrate the effects of let commands we will often use other commands like print and list; in those cases, as

in the isp Tutorial, we assume that the user types what follows the isp shell's prompt, and that the computer responds with the rest. A typical fragment might be

```
* let z = a^2
* print a z
1.000     2.000     3.000
1.000     4.000     9.000
*
```

The user types what follows the prompt on the first two lines; the computer types the rest.

The ensuing chapters are designed to show you first how to use let for simple data analytic/manipulative tasks, and next, how to use it for realistic problems.

I. The Rules of the Game

Here we cover the elementary operations that let can perform. We begin with backgroung information.

A. Data Configuration in isp

The raw elements that **isp** and **let** manipulate are arrays of floating-point numbers. On PDP-11's these are essentially numbers of 6 significant digits in the range ±(10^-38,10^+38). In **isp** arrays are sets of floating numbers indexed by no more than 3 subscripts and recorded internally in row-major order (rightmost subscript varying fastest). Thus an array is shaped k x n x p, containing k tables(matrices), each with n rows and p columns. In a sense shape is the single most important fact of data organization. We typically describe our data not as arrays, but according to their shape as columns or rows or tables. Perhaps the most often-used **isp** command, list, describes arrays solely by name and shape.

We can even guess the type of analysis the data were collected for by the shape of the array into which thay are put. Thus, an n x 2 array generally suggests x-y plot, regressions and other types of function-relationship investigrations. A 1 x n array may be a time series or a completely unstructured batch of numbers. If the matrix is n x p, with p>2, we may have a whole collection of responses ob-

served under each of n distinct circumstances to be related, say by correlation methods. And so on.

The concern with shape leads even to the device of denoting values as **missing**. If we observe p responses for each of n individuals but can't get a full record for each respondent (for example if (s)he refuses to answer certain questions), then we record unknown values as **missing** values, denoted by empty parentheses in **let**--"()". Filling in the 'holes' in the table preserves the n x p structure of the table of responses. In recording closing prices of a stock for each business day, if we denote holiday readings by () then readings whose indices are two apart were taken two days apart: the ideal structure of the data is maintained even in the absence of ideal data to put in it.

A final word on shape. Many **let** operations expect their arguments to have a certain shape--binary operations generally need both arguments to have the same shape, and only square matrices can be inverted. Arrays which meet such requirements are <u>conformal</u>. Most **let** error messages are simply complaints that arguments aren't conformal.

<u>B. Arithmetic</u>.

let will perform arithmetic operations and will compute the most common functions of trigonometry and analysis. The language for expressing these is much the same as Fortran or Basic, and hence much the same as standard handwrit-

ten notation of algebra.

let accepts well-formed formulas involving the symbols +,-,*,/,=,(,), such as

```
a = b + (c*b)
d = 2*frog - brown/(grass+lily)
e = -(f*2)/(c-3.1)
```

Just as in Fortran & Basic the meaning of these symbols is

+,-	addition, subtraction
*,/	multiplication, division
(,)	parentheses for grouping
=	assignment

These operations are performed elementwise, so that any two arrays combined using these binary operations must either have the same shape, or one must be a constant (1 x 1 x 1).

```
* print a (a+1) (a/3-4)

 1.0000   2.0000   3.0000   4.0000   5.0000

 2.0000   3.0000   4.0000   5.0000   6.0000

-3.6666  -3.3333  -3.0000  -2.6666  -2.3333

* print b (b-(b/10))

 0.1000   0.2000  -0.5000

 0.0900   0.1800  -0.4500
```

But,

```
* print (a-b)
Matrix shape error
```

Precedence is interpreted as in Fortran: in a left-to-right scan, execute first items in parentheses, then operations * and /, with + and - having lowest priority.

```
* let c = 1-a; b = c/a;
* print (a + b*5) ((a + b)*5)
```

```
1.0000   -0.5000   -0.3333    0.2500    1.0000

5.0000    7.5000   11.6666   16.2500   21.0000
```

In addition to these simple operations, one can also perform exponentiation; it has a higher precedence than multiplication or division.

```
* print (c * a^b) ((c*a)^b)
```

```
1.0000   -0.2928   -1.5192   -2.6464   -3.7240

1.0000    1.2840    1.5596    1.7550    1.8964
```

let also can compute most of the important functions from trigonometry and analysis. These take arrays as arguments and generate conformal arrays as outputs, by computing the indicated function elementwise. Thus,

```
* print a (exp(a)) (log(a))
```

```
1.0000    2.0000    3.0000    4.0000    5.0000

2.7182    7.3890   20.0855   54.5981  148.4131
```

```
0.0000    0.3010    0.4771    0.6020    0.6989

* print (w = a/3) (sin(w)) (atan(sin(w)/cos(w)))

0.3333    0.6665    1.0000    1.3333    1.6666

0.3271    0.6183    0.8414    0.9719    0.9954

0.3333    0.6666    1.0000    1.3333    1.6666
```

A complete list of these functions is

sin,cos	sine, cosine
exp,log,ln	exponential function, base 10 & natural logs.
sqrt	square root
atan,acos,asin	arc tangent,cosine,sine
atanh	arc hyperbolic tangent
abs	absolute value
int	truncate to integer
sgn	signum function
pi	the value 3.141592

C. Data Manipulation

Many useful operations in data analysis involve selecting subsets of data, or reshaping arrays, or joining several datasets together into one. We group all these operations under the heading 'Data Manipulation'; as opposed to 'Data Reexpression', which changes the values in an array, but not its shape, these operations change shapes but 'copy over' values directly.

1. Dimensioning. Dimensioning reshapes an array by changing the numbers of rows/columns/tables it contains. If

a is an m x q matrix, a(n,p) will be an n x p array with its values filled in row-major order from a. If m.q > m.p the extra values are taken as missing.

```
      * list
a           array(4,3)
      * let b = a(3,4); c=a(2,7); d = a(2,2)
      * print a b c d

    1.0000    2.0000    3.0000
    4.0000    5.0000    6.0000
    7.0000    8.0000    9.0000
   10.0000   11.0000   12.0000


    1.0000    2.0000    3.0000    4.0000
    5.0000    6.0000    7.0000    8.0000
    9.0000   10.0000   11.0000   12.0000


    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000
    8.0000    9.0000   10.0000   11.0000   12.0000 ********* *********


    1.0000    2.0000
    3.0000    4.0000
```

Note that the arguments in parentheses are constants, not variables. A handy fact is that if a is n x 1 -- a column -- then a(1,n) is a row.

2. Subscripting. Subscripting selects subsets of an array based on the value of their subscripts. The simplest example is the selection of slices, such as rows or columns, from the array. The let notation for all the elements of column i is [*,i]; for all the elements of row j, [j,*]. Thus x[1,*] denotes the first row of x.

```
      * list
```

```
x           array(4,3)
    * print (x[1,*]) (x[*,3])

    1.0000     2.0000     3.0000

    3.0000
    6.0000
    9.0000
   12.0000
```

Sets of columns or rows are selected simply by listing out the indices of the desired slices:

```
    * print (x[1 2,*]) (x[3 4,2 3])

    1.0000     2.0000     3.0000
    4.0000     5.0000     6.0000

    8.0000     9.0000
   11.0000    12.0000
```

Note that '*' is equivalent to '1 2 ... k' if subscripting on that dimension runs as high as k.

A notational shortcut for long lists within subscript expressions is the use of 'k:l' in place of 'k k+1 ... l' (or 'k k-1 ... l' if l<k). Thus if x is 4 x 3, 'x[*,1]' ≡ 'x[1:4,1]' ≡ 'x[1 2 3 4,1]'.

An example of the most general combination of notations would be

```
x[1:3 5 9 21:12,1 4 3 8:11,*]
```

A practical application of these methods would come in

rearranging columns of an array to be used for generating plots. The **isp** program **scat** assumes (although this can be overridden) that the x-coordinates for a plot are stored in the first column of the argument and the y-coordinates in the second column. Supposing we had originally gotten this backwards, we could reverse the columns by a simple instruction of the form

```
* let x = x[*,2 1]
```

nothing could be simpler!

3. Concatenation. An extremely common operation in **isp** is the joining together of several arrays into one array. The reason for this is that when performing reexpressions of a variable, we typically store that variable in an array by itself; but when performing analysis using **isp** plotting or fitting routines, all the variables we wish to discover relations between need to be stored as columns in the same array. Thus a typical cycle in a data analysis involves breaking down an array into parts, reexpressing the parts, and then piecing them back together to fit a model-- such as in the following fragment:

```
* let x = dat[*,1]; y=dat[*,2]
* scat (1/x,log(y))
```

In **let** the ',' concatenates columns of the same length into one array. THus 'x = x[*,2],x[*,1]' is a feasible solution to the problem of reversing theorder of two columns in an array. Matrices with the same number of rows can be

joined by ',' also; in 'a,a' every column of the original matrix a appears twice.

What happens if we need to add extra rows to our dataset, for example because we have aquired new observations to analyze?

Here we use the ',' in its unary form. If x is k x n x p, ',x' is 1 x 1 x (k.n.p), a single row containing in row major order all the elements of x.

```
     * print x (,x)

 0.6104    0.5768    0.9897
 0.1210    0.5570    0.9878
 0.7772    0.9392    0.6463
 0.8930    0.4416    0.6734

 0.6104    0.5768    0.9897    0.1210    0.5570    0.9878    0.7772
           0.9392    0.6463    0.8930    0.4416    0.6734
```

Using this 'unravelling' operation together with concatenation, we can solve the problem of adding rows by reshaping, adding columns, and shaping back:

```
     * list
x          array(35,4)
y          array(15,4)
     * let a = (,x),(,y)
     * let x = a(50,4)
     * list
a          array(200)
x          array(50,4)
y          array(15,4)
     *
```

So we can add new rows to x by unravelling it, by concatenating the unravelled new rows to the end of the unravelled

old ones, and then reshaping the whole thing. the ',' appears useful both in unary and binary forms.

4. Transposition. Sometimes, especially for algebraic reasons, the transpose of an array is of interest; it is computed by exchanging rows with columns, so that an n x p matrix becomes p x n and the element accessed by [i,j] in the old array is accessed by [j,i] in the new one. In let the transpose of x is given by trn(x).

When an operator like ',' works only columnwise and we need it to work rowwise, we can use trn to interchange the roles rows and columns and solve our problem. For example, the previous section's method of adding new rows to a matrix is much improved upon using transposition:

```
* let x = trn(trn(x),trn(y))
```

D. Special Functions.

let has certain functions which operate on an array and return an array of the same shape in which the values have been changed by more complicated means than the application of a simple function elementwise. The following is a quick itemization.

iota: returns an array with the first k.n.p natural numbers in row major order.

```
      * list
x           array(3,2)
```

```
* print (iota(x))

1.0000    2.0000
3.0000    4.0000
5.0000    6.0000
```

sort: returns a copy of the argument with sorted values stored in ascending, row-major order.

```
* print (6-iota(x)) (sort(6-iota(x)))

5.0000    4.0000
3.0000    2.0000
1.0000    0.0000

0.0000    1.0000
2.0000    3.0000
4.0000    5.0000
```

rnd: returns an array filled with independent, uniform[0,1] random numbers.

```
* print(rnd(x)) (rnd(x))

0.6104    0.5768
0.9897    0.1210
0.5570    0.9878

0.7772    0.9392
0.6463    0.8930
0.4416    0.6734
```

gau returns an array filled with independent, Gaussian(0,1) random numbers.

```
      * list
z     array(50)
      * let z = gau(z)
      * stemleaf z


-02|0
-01|865
-01|4443321000
-00|99988755
-00|444432210000
 00|1123444
 00|663389
 01|234
```

diff computes first differences of its argument.

```
      * print x (diff(x))

    1.3093     1.8805
    0.2645     0.0309
    0.7859     0.2925

    1.0000     1.0000
********** **********
********** **********
```

The missing value is present because 6 numbers generate only 5 first differences - x[2]-x[1],...,x[6]-x[5].

scan computes a running sum of its argument.

```
      * print (scan(x))

    1.3093     3.1899
    3.4544     3.4854
    4.2713     4.5639
```

Note that **scan** and **diff** are essentially inverses to each other, in the same way that integration and differentiation are inverses.

```
* print (y=,x) (diff(scan(y))) (scan(diff(y)))
```

1.3093	1.8805	0.2645	0.0309	0.7859	0.2925
*********	1.8805	0.2645	0.0309	0.7859	0.2925
*********	0.5712	-1.0448	-1.2784	-0.5234	-1.0167

diff(scan(.)) amounts to making the first value in the series missing, while **scan(diff(.))** combines this with a translation so the first value in the series is zero.

smooth computes a smoothed version of its argument by Tukey's '3rssh*2' procedure.

```
* let y = smooth(1,-1,3,-.05,5,9,0,8,9,5)
* print y
```

1.0000	1.0000	1.6250	3.3750	5.0000	6.1250	7.3750
	3.0000	8.0000	8.0000			

To demonstrate these tools in action, we perform Tukey's "Guided Diagnosis of Reexpression". Suppose we are presented with two unstructured batches of numbers, x and y, and are asked to find a function f such that **y=f(x)**. Tukey suggests a method involving only the functions of **let** we

have already seen.

First let's generate two samples.

```
* let z(100,1) = 0
* let y = cos(rnd(z)); x = gau(z);
```

Thus we will be looking for the function which transforms Gaussian random numbers into cosines of random angles in [0,1].

We proceed by sorting:

```
* let y = sort(y); x = sort(x);
```

and then we difference the sorted values and estimate the derivative.

```
* let dy = diff(y); dx = diff(x);
* let df = dy/dx;
```

To refine our estimate, we smooth and then integrate the smoothed derivative to get a smooth functional form:

```
* let df = exp(smooth(ln(df)))
* let f = scan(df*dx)
```

The results--

```
* scat (x,f)

0.500                    |    33*43 *   *
                         |    3
                         |   3
                         |  28
                         | *7
                         |67
                         |45
                         |*3
                         |7
                         |*
                         |35
                         |4
                         |2
                         |3
                         |2
                         |2*
                         |2
0.000    ---*------------|----------------
    -3.000                          4.000
```

This complicated but useful simple data analysis procedure shows let to be convenient and powerful.

E. Logical Functions

let offers a set of logical operations which provide powerful manipulation/reexpression capabilities. Conditional reexpressions are especially easy with let.

1. Logical Operators. The symbols >,<,<=,>=,==,!= provide binary truth-valued operators. Conformal arrays are compared elementwise according to the sense indicated by the opertor's symbol, and the elementwise answer is 1 when the

comparison is true and 0 when it is false.

```
* print a (a > 0) (a < 1) (a == 2) (a != 0)

-1.0000   -3.1000    0.0000   4.0999   2.0000   1.6000   2.0000
           2.9000

 0.0000    0.0000    0.0000   1.0000   1.0000   1.0000   1.0000
           1.0000

 1.0000    1.0000    1.0000   0.0000   0.0000   0.0000   0.0000
           0.0000

 0.0000    0.0000    0.0000   0.0000   1.0000   0.0000   1.0000
           0.0000

 1.0000    1.0000    0.0000   1.0000   1.0000   1.0000   1.0000
           1.0000
```

Conjunction is indicated by &, disjunction by |:

```
* print (a < 0 | a > 2) (a != 0 & a != 2)

 0.0000    0.0000    1.0000   0.0000   1.0000   1.0000   1.0000
           0.0000

 1.0000    1.0000    0.0000   1.0000   0.0000   1.0000   0.0000
           1.0000
```

Naturally, comparisons bind tighter than con- or dis- junctions.

2. If-Then-Else. Although logic-valued variables have some utility by themselves, they primarily are intended for use with the let

if ? then : else

This expression takes the values indicated by **then** when **if** is true, and the values indicated by **else** otherwise.

```
* print a (a > 0 ? sqrt(a) : .123456) (a >= 3 ? a-3 : 3-a)
```

-1.0000	-3.1000 2.9000	0.0000	4.0999	2.0000	1.6000	2.0000
0.1234	0.1234 1.7029	0.1234	2.0248	1.4142	1.2649	1.4142
4.0000	6.0999 0.0999	3.0000	1.1000	1.0000	1.4000	1.0000

the expressions given by **if**, **then**, and **else** must be conformal--all the same shape, except that some may be constants.

```
* let b = a >= 1.5 ? 1.345 : -2.001
* print b
```

-2.0010	-2.0010 1.3450	-2.0010	1.3450	1.3450	1.3450	1.3450

Of particular use is the symbol () for missing value, which can be used to obliterate unwanted data.

```
* let d = a==2 ? () : a
* print d
```

-1.0000	-3.1000 2.9000	0.0000	4.0999	*********	1.6000	*********

So outliers can be made missing:

```
* let x = gau(x(15)=0); x= abs(x)>1.5 ? () : x;
```

This makes missing all elements in this Gaussian sample which are more than 1.5 standard deviations from their mean.

Use of the 'if-then' construct allows choice of subsamples. Suppose that we want to regress y on x in two separate groups: those in which and auxiliary variable z is 0 and those in which it is two. A simple strategy is to mangle those cases not in the group were studying; so we try

```
* let x0 = z==0 ? x : ();
* let x2 = z==2 ? x : ();
* regress (x0,y); regress (x2,y)
```

II. Beyond the Rules

Here we discuss additional functions and isp commands which extend the usefulness of let into more complex domains.

A. Matrix Manipulations

Multiplication of two matrices is accomplished in let with the "#*" operator; the arrays must be n x p and p x m, respectively, and the result will be an n x m array each element of which is the inner product of a row of the first and a column of the second.

Inversion of a square matrix is performed by the inv function. inv(x)#*x is the identity matrix (up to roundoff error). Thus the system of equations

$$v = C\#*w$$

is solved by w = inv(C)#*w.

As an example, the linear regression of y (n x 1) on x (n x p) is given by

$$\hat{y} = x\#*B$$

where

$$B = inv(trn(x)\#*x)\#*trn(x)\#*y$$

B. Selection/Reduction Functions.

Certain special functions not provided by **let** are performed by isp commands written to meet the deficiencies.

select extracts those rows from an array for which a given condition is true. Thus the rows of x with nonzero first element are given by

```
* select x (x[*,1] != 0) > y
```

The problem at the end of Chap. I. of splitting a dataset into two subsets based on an auxiliary variable can easily be handled:

```
* select (x,Y) (z == 0) > r0; regress r0
* select (x,y) (z == 2) > r2; regress r2
```

coalesce allows one to summarize a set of (x,y) pairs with identical x's according one of several measures of size, location or dispersion. For example, if **pres** contains the Gallup poll's recorded presidential popularity, and **year** contains the year of each poll taken, then the yearly average popularity is given by

```
* select (year,pres) > npr
```

where npr becomes a set of pairs: (year,average popularity for the polls of that year).

2. A Tutorial Introduction to ISP

David Donoho
Department of Statistics
Princeton University

A Tutorial Introduction to ISP

David Donoho
Department of Statistics
Princeton University

ISP, the Interactive Statistical Package, is a system for data analysis running on PDP-11 minicomputers under the UNIX operating system. It is designed with two goals in mind: (i) to make statistical analysis quick and painless by exploiting the interactive capabilities of a timesharing system; and (ii) to make data analysis exploratory and/or robust by utilizing the newest available statistical "technologies."

This introduction is meant to simplify learning ISP and to give examples of ISP and the exploratory paradigm in action. Hopefully, it will also shed some light on the strengths and weaknesses of the ISP approach relative to a given user's special needs.

To learn ISP most easily, read this paper and simultaneously use ISP on your UNIX system to work the exercises provided.

0. Getting Started

We assume to begin with that you have at least

(i) A vague acquaintance with with the ideas underlying much of data analysis: description of batches by center and spread; the idea of fitting a line to data or an additive relation to a table; and the use of transformations to simplify data description.

(ii) Some familiarity with the idea of collecting data into arrays and manipulating them as such.

(iii) Above all, knowledge of the typing conventions and rudiments of life under UNIX.

See J.W. Tukey's EDA for an introduction to data analysis; see "UNIX for Beginners" in the Documents for Use with UNIX book for an introduction to the operating system.

A. Invoking ISP

After logging on to UNIX, so that the shell has prompted you with "% ", you may invoke ISP by typing "isp" followed by a carriage return. When ready to accept input from the keyboard, ISP will prompt you with a tab followed by "* ". The following fragment is representative of what you might see printed at your terminal at the beginning of an ISP session:

```
*** name: statfiend
Password:
% isp
    *
```

B. A Convention

Since ISP is interactive, a session with the system is a dialog. You command; the computer responds. The computer queries; you respond. In this tutorial, we reproduce dialog verbatim, as it would appear printed at the terminal, without markings of 'who' (you or the computer) typed what. It is therefore important to remember that ISP prompts the user, and so in a dialog, all lines beginning with the prompt are finished by what the user typed. Also, more or less, non-prompted lines are typed by the computer.

C. Conversing with ISP

ISP understands a 'language', and, to perform an analysis you must learn how to speak it. As with any language, both syntax and semantics are important. The syntax to get started with is very simple--an ISP command is generally of the form

```
verb noun
```

as in

```
print x
delete junk
```

The semantics for getting started are also very simple--in an ISP command, the verb is generally the name of a program and the noun is generally the name of an array of real numbers upon which an analysis, display procedure, or manipulation is to be performed. Obviously, for an ISP command

to make sense, both the verb and the noun must be in ISP's vocabulary. This means for now that the verb must refer to a preexisting command supplied with the system and the noun must refer to an array present in your active workspace--data of your own that you have entered into the workspace yourself, or else arrays from the system workspace which you have 'loaded' (i.e. made a copy of) into your own area.

D. Errors

If you make a mistake in understanding the ISP syntax or in spelling a name, ISP will generally indicate your mistake with an error message. For instance, if in trying the command "print x" you mistakenly type "pint x", the following dialog might result:

```
     * pint x
ERROR: can't find pint
     * print x

1.03251

     *
```

Here ISP tells the user that the verb "pint" has no established meaning. Don't panic when presented with an error message. Act as if the erroneous line hadn't been input, and error recovery is easy--simply figure out what should have been typed and try again.

E. Help

Frequently an error will arise because a user imperfectly understands the form or purpose of a given command. The "explain" verb is designed to provide the user with on-line documentation for each command in the ISP system's 'vocabulary' as well as for some concepts important to an ISP user. The usage of the explain command is "explain name", where "name" is the name of whatever you'd like explained.

EXERCISE 1: Try the command "explain explain". This should give you an indication of how "explain" works. Try the command "explain all". This should give a two-page brief summary of everything "explain" can explain to you.

EXERCISE 2: For the rest of this tutorial, use "explain" to get additional information about each command we introduce.

I. Analyzing Data

With a rudimentary description of the ISP language out of the way, we begin to analyze some data. Without exception, we look at data from Don McNeil's book Interactive Data Analysis, (J. Wiley, 1977), and datasets will be referred to by their name and page in that book.

The first batch of data we'll look at is the dataset "precip", [McNeil, p. 3]. To obtain a copy of this data, we load it from the system area as follows:

```
     * load sys precip
     * list
precip          array(69)
     *
```

Here we have loaded the data we wanted and listed out the contents of our active workspace, revealing that the load command did work, and provided us with an array of 69 numbers. These numbers give the average yearly amounts of precipitation, in inches, for each of the weather stations in the U.S.

EXERCISE 3: Print out the numbers in the precipitation array using the command "print precip". What would you say is a 'typical' level of yearly precipitation for a weather station in the continental U.S.?

A. Numerical Summary

Obviously, printing out a bunch of numbers at a terminal is going to give us more information, but less of a use-

ful sort, than we would like. So, let's try getting numerical summaries of the batch - by letting ISP calculate a few useful statistics which tell us about the array.

```
* fivenum precip

n       0%      25%     50%     75%     100%
69      7.000   29.10   36.20   42.80   67.00
```

Here "fivenum" is a command for producing estimates of 5 of the most useful percentiles - the extremes (0% and 100%), the median (50%), and the upper and lower quartiles (75% and 25%). From the above information it appears that while rainfall ranges from 7 to almost 70 inches, about half of the stations average between 30 and 40 inches yearly, and 35 inches a year is a good 'typical' value.

EXERCISE 4: Try other numerical summaries of the data: "condense" will calculate the median (as did "fivenum") and also compute the median distance from the median (think about what such a number describes!); "stat" will calculate the mean and standard deviation (brush up on the definition of these terms if you need to). Describe the data in a sentence of the form "U.S. rainfall is typically x inches plus or minus y inches." Does such a statement make sense?

B. Graphics

The next step beyond numerical summary of data is graphical summary. Using the the same "precip" array, we can examine its boxplot [McNeil, pp. 7-8] as follows:

```
     * boxplot precip

 7.00                                                        67.0
                        |--------------|
**   ooooox-------------|       *      |-------------x  o       *
22      2               |--------------|               2
```

This plot shows that the data are symmetrically distributed about the median (the star at the center of the "box") and pinpoints the outliers--both very dry and wet. Based on this plot a statement like "typically a U.S. weather station experiences 35 ± 10 inches of precipitation in a year" seems to have some meaning.

EXERCISE 5: Try a stem-and-leaf plot (a fancy histogram, see [McNeil, pp. 3-6]) of "precip"; the verb is "stemleaf". Do the data look at all 'bell-shaped' or 'Gaussian'?

C. Transformations

Having been introduced to the ISP system with a fairly boring and easily-analyzed dataset, let's examine less prosaic situations. The batch we'll look at is called "rivers" [McNeil, p. 14] and contains the lengths of 141 major rivers in North America. Again, we load it from the system area.

```
     * load sys rivers
     * list
precip          array(69)
rivers          array(141)
     * delete precip
     * list
rivers          array(141)
     *
```

Here for no reason other than tidiness, we have deleted the now fully-analyzed and hence presumably useless batch "precip" from out workspace.

We begin with a boxplot of our data and find that the river lengths vary greatly--from hundreds to thousands of miles:

```
* boxplot rivers

  135.                                                        0.371e 04
    |------|
 x--| *    |-----xooo** *     * *      **  *                          *
    |------|          2  2
```

Clearly, 'miles' is not the unit most amenable to analysis of our data. Fortunately, ISP allows us to easily reexpress data in forms more suitable for our use. The mechanism for this is the "let" command, which is typically of the form

let var = expression

where "var" is the name to be attached to the result of the "expression", a mixture, according to the rules of algebra and functional notation, of isp array names and real constants with

+,-	addition & subtraction	a+b,c-d
*,/	multiplication & ddivision	e*f/g
^	exponentiation	a^.012,b^(a-1)
()	parentheses for grouping	(a+b)*c
=	for assignment	a = z/y
f()	mathematical functions	a=sin(omega) b=exp(1-1/x)

For example, expressing the rivers data in 'miles^2' would

be accomplished by the command

```
     * let river2 = rivers ^ .5
     * list
rivers          array(141)
river2          array(141)
     *
```

EXERCISE 6: Define S by 2*(median - lower quartile)/(upper quartile - lower quartile). Using "let" to compute reexpressions and "fivenum" to compute the needed percentiles, find the units in which S is most nearly 1. Try miles^-1, miles^-2, exp(miles), miles^2, and miles as different units in which to express the data. Hint: an expression in parentheses where a name of an array ought to be will cause the indicated calculation to be performed by "let". Hence the problem can be solved using the commands

```
     * fivenum rivers
     * fivenum (rivers^.5)
     * fivenum (rivers^-.5)
     * fivenum (1/rivers)
     * fivenum (log(rivers))
```

A visual answer to the above exercise can be gotten using the graphics commands at our disposal. Following [McNeil, p. 15], we try

```
     * stemleaf (rivers ^ -.5)
```

```
 01|6
 02|01134
 02|6633899
 03|01122333444
 03|556677777788899
 04|00000111122333344444
 04|5556677777888999999
 05|011111122333333333344
 05|555555667778888999
 06|00000111222333 4
 06|56668899
 07|0
 07|
 08|
 08|6
```

```
* boxplot  (rivers ^ -.5)

0.164e-01                                                    0.861e-01
                    |----------------|
o  x----------------|        *       |----------x                    *
                    |----------------|
```

which gives a fairly appealing symmetric look to the data. Thus we might conclude that ISP has succeeded in showing us that river lengths are naturally expressed in miles^-2, and that a typical major river in the U.S. has length .05±.01 miles^-2.

EXERCISE 7: The dataset "islands" [McNeil, pp ??] in the "sys" workspace contains the sizes of the world's 48 largest islands. Australia is the 7th largest; Greenland is the 8th. Choose a reexpression of "islands" in which Australia is clearly a continent and Greenland clearly an island -- that is, pick units so that islands cluster with islands, continents with continents. Can you find a scale in which the two countries appear close in size relative to the other islands? Hint: try both negative and positive powers, as well logarithms. Use "stemleaf" to evaluate your results.

II. Relations

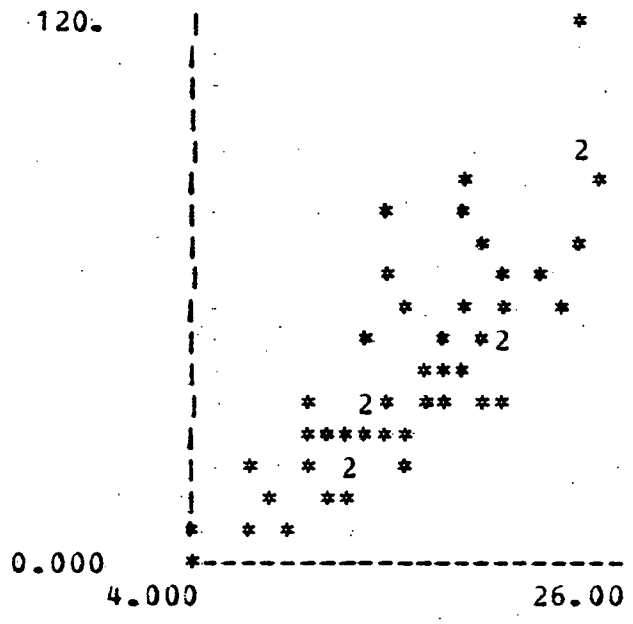
The preceeding sections focused on single batches, their graphical and numerical summaries, and reexpressions. Often we find more complicated situations. For example, we may be measuring the height and weight of many women, the I.Q. of children and their fathers, or the height from which a ball is dropped and the time it takes to reach the ground. In such instances, we are observing several different variables simultaneously; we may be interested in finding a relation between them.

A. X-Y Data

Consider the following example. In an effort to determine a car's stopping distance as a function of its its initial speed, measurements were made in an experiment where 50 different drivers were asked to halt their cars abruptly. The array "cars" [McNeil, p.52] has 50 rows and 2 columns; in the first column are recorded the drivers' initial speeds and in the second are the corresponding stopping distances. The units are miles/hour and feet, respectively.

The first, most natural display of the data we might want is an "x-y plot" of the scatter of data. The "scat" command will do this in ISP. To get our analysis started, we type:

```
     * load sys cars
     * list
cars       array(50,2)
     * scat cars
```



The relation is clear: as initial speed (x axis) increases, stopping distance (y axis) also increases. It appears that a "straight line fit" of these data is possible. The command "line" will compute such a fit, using an algorithm suggested by [Tukey, p.???]. Let's try it:

```
* line cars > resids; scat resids

intercept  -9.143
slope       3.143

 50.0 |
      |        *
      |
      |                      *
      |
      |     *   *     **
      |
      |         * **
      |    *   2
      |      * *       *  * *
      |      2 3  * *
      |--*-*-------*--*----*---
      |    * 23 3  *
      |    *3  2 *
      |   * *  *
      |    *
      |
-30.0 |
      10.00                  90.00
```

The output of the "line" program seems reasonable -- it says that for every 10 mph increase in speed, there is about a 30 foot increase in stopping distance, which suggests that, if anything, the "1 car length per 10 mph of speed" rule is conservative!

The above command line showed us two things we hadn't seen before: first, two commands 'stacked' on a single line, separated by a semicolon--this works in general; second, a command with two extra arguments--a '>' and a variable name. This last feature is very new and also very useful. As it

turns out, many of the more complicated programs can generate various arrays as results, in addition to any results they may print at the terminal. The 'arrow' notation is used to indicate where, meaning in what variable(s), the result(s) are to be placed, or if they are to be generated at all. So, "line cars > resids" would mean "fit a line to the 'cars' data and generate a new array, 'resids', containing in the first column the predicted y-values and in the second column the residuals about the fitted line." The command "scat resids" would then plot the residuals from the fitted line vs. the fitted values themselves. Any nonlinearity in the relation would be most likely to show up at this point. Judging from the visual information, the fit is pretty good.

B. Reexpressions

While data transformation is an important technique in the single batch case, multivariable relations can be so subtle that a well-chosen reexpression of the data will make the difference between a muddled and a crystal-clear picture of the phenomenon underlying the data. Indeed, one could argue that most basic successes in the physical sciences have been discoveries of transformed viewpoints in which the relationships between observed quantities became simple or elegant.

Consider the previous section's "cars" example. While

we may be relatively satisfied with the fit obtained there, the residual plot shows a hint of additional structure. Even in the original "scat cars" plot we may detect a hint of upward concavity--a sign of nonlinearity. [McNeil, pp. ??] suggests that we change our point of view and fit

stopping distance = a*speed + b*(speed^2)

omitting a y-intercept, since a car travelling 0 mph has a stopping distance of 0 ft. The extra term in 'speed^2' models the upward concavity.

How do we fit this non-linear equation? The answer is to transform the variables so the equation takes a linear form, fit, and then transform the fitted equation back into its original form. Here that is easy, since if we divide both sides by 'speed' we get

distance/speed = a + b*speed

a linear equation in the 'speed' variable. The indicated change in viewpoint is easily accomplished using the "let" command:

```
* let speed = cars[*,1]
* let dist  = cars[*,2]
* let ncars = speed, (dist/speed)
```

Here the brackets '[ ]' provide for subscripting in "let". Generally, to access the j'th element of row i in array "x", type "x[i,j]"; "x[i,j:k]" accesses elements j through k of row i. To access all the items in the i'th row, use "x[i,*]". (See the section on subscripting in the "let tu-

torial" for details).

Thus the above fragment makes "speed" a copy of column 1 of "cars"; "dist" a copy of column 2 and makes "ncars" a 2-column array with speed as the first column and dist/speed as the second.

EXERCISE 8: Fit "line ncars". Is this a better fit than "line cars"? Hint: one way to answer is to define

R = spread of predicted y's / (spread of predicteed y's + spread of residua

The statistic 'R' then measures approximately the proportion of variation in the y-data accounted for by the fitted model; naturally, the higher 'R' is, the better we think the fit is. If we use as a measure of spread the MAD as calculated by "condense", the question could be answered with the commands

```
* line cars > r1; condense r1
* line ncars > r2; condense r2
```

EXERCISE 9: The dataset "vapor" [McNeil, p. 52] contains observations of temperature and saturation vapor pressure of mercury. Due to the Clausius-Clapeyron equation of Physics, we know that, approximately,

$$\log(\text{pressure}) = a + b/\text{temperature}$$

Transform the columns of "vapor" so this equation can be fit using "line". Rewrite the fitted equation in terms of the original variables.

C. Time Series

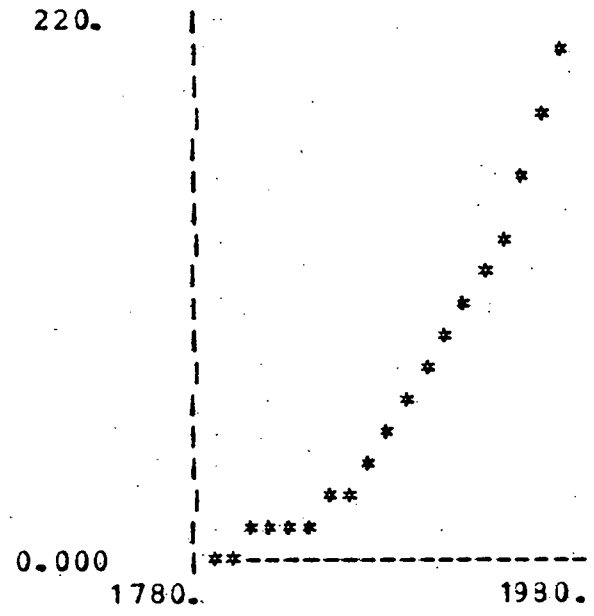
An important type of data organization is that of the time series, a sequence of observations taken at different points in time -- an example is the outside temperature at noon, recorded daily. One way to look at such a series, if the data are sampled at points equally spaced in time, is as

a single batch where order matters, so the first element of the array is the observation for the first time period.

D. Forecasting

Typically we think of a time series as being a smoothly varying function of time, so that there is only a small change in the series' level from time t to t+1. It makes sense in such instances to try and predict the next value of the series from the immediately preceding one; such is an example of forecasting. Here we give an example of this idea using the dataset "uspop" [McNeil, p.52], which contains, for each of the 19 decennial censuses, the year the census was conducted and the corresponding U.S. population for that year. We display the data below, using "scat".

```
     * load sys uspop
     * list
uspop           array(19,2)
     * scat uspop
```



Clearly, population is increasing over time; we would like to perhaps fit an equation of the form

$$pop(c+1) = a + b*pop(c)$$

where 'c' is the particular census we're considering. In other words, we want to fit a line to a pair of variables: (i) pop'n for each census from 1790 to 1960; and (ii) pop'n for 1800 to 1970. Using "let" this can be easily set up:

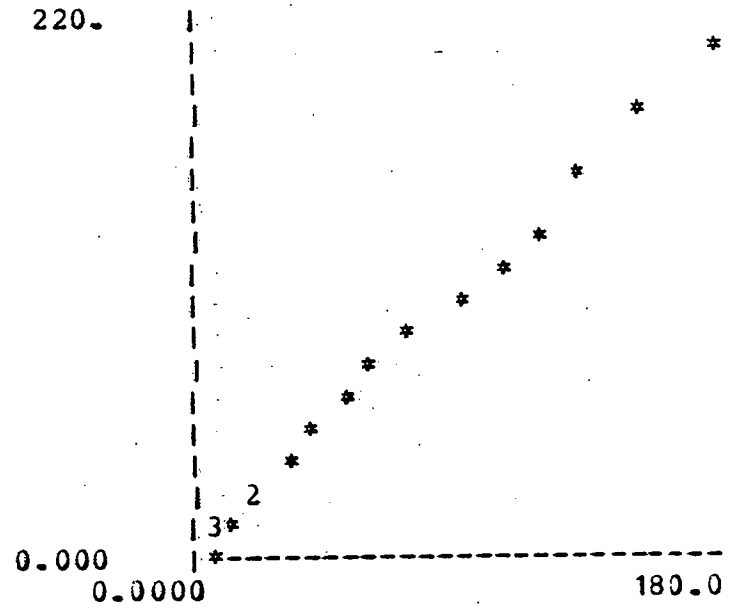
```
     * let pop = uspop[1:18,2],uspop[2:19,2]
     * list
pop         array(18,2)
uspop           array(19,2)
     *
```

Here "uspop[1:18,2]" gives the pop'n for 1790 through 1960; similarly, "uspop[2:19,2]" is an expression for the last

eighteen census populations. The ',' operator concatenates these two columns into a single array whose first column is the 1790-1960 stretch and whose second column is the 1800-1970 batch.

When we scatterplot these data, we see clearly the implicit dependence of one census' reading on the preceding one:

```
      * scat pop
```




```
      * line pop

intercept    2.442
slope        1.136
```

Thus we get a simple one-step forecasting rule for U.S. population:

$$pop(c+1) = 2.4 + 1.1*pop(c)$$

Based on this rule, we project the 1980 census to give a population of 230 million. Since the current official projection for 1980 is about 10 million less than this, it may be that growth is declining from its historical rate.

E. Smoothing

A reasonable problem when given a noisy time series is to 'smooth' it so that the noise is suppressed and any smooth variation is more clearly seen. To give an example of this, we consider the array "sunspots" [McNeil, p.69], which contains the monthly relative sunspot numbers for Jan. 1958 to Dec. 1972. Using the command "six", we can produce a six-line plot [Tukey, p??] of this data:

```
* load sys sunspots
* six sunspots


typical values :  25.15       77.00       128.9

 +2 : 110201
 +1 : 231131 1 0                                    0
 +0 :       31300 0                      2221233231122110 00000
 -0 :           11132223 33          311000                1211012
 -1 :                 0000 011100000
 -2 :
```

Here, time is the x-axis and along the y-axis the data are measured in terms of spread-lengths from the median in base 4: a '2' on the line with +2 on the left-hand side indicates

a value of between 2.5 and 2.75 'spreads' above the median. From this plot the basic trend is downward until the middle of the period, then upward, then down again towards the end. However, the noise present in the data makes us wish for a 'smoother' estimate of the data before firmly establishing our informal diagnosis. Thus we use the command "smooth" as follows:

```
* smooth sunspots > ss
* six ss


typical values :  28.89       80.00       131.1

 +2 : 0110
 +1 :  33 21
 +0 :       32300                          10012222311221 00   00
 -0 :           12132233 33           210                 0121 012
 -1 :                 000  111111000
 -2 :
```

There is no question that the trend is clearer in the smoothed data.

EXERCISE 10: Try using the approach to forecasting of the last section to fit a model in which this month's sunspot number is a linear function of last month's.

EXERCISE 11: Using "let", generate a time series of unrelated random numbers. Then use "smooth" to see if your realized series appears to have trends or some sort of underlying smooth variation. What do you see? Hint: to generate a random series try:

```
* let ser = gau(x(50) = 0)
```

which should produce a batch of 50 uncorrelated unit Gaussian (i.e. Normal) numbers.

III. Several Batches

A data organization with even more useful structure than either the bivariate array or the time series is the multiple batch array. Here we think of one variable observed repetitively in each of several distinct situations. Crop yield measured on many different 1-acre plots each sprayed with one of several fertilizers gives an example of this type of structure. The diversity allows us to speak of both within-batch structure (yield variation in plots treated with the same fertilizer) and between-batch structure (yield variation between plots with different fertilizers). ISP has several tools for dealing with the several batch situation.

A. Comparisons

In [McNeil, chap. 2] this situation is considered in detail; we find there our first example. The dataset "chickwts" [McNeil, p. 31] contains 6 batches of data referring to the weights of chickens under each of 6 different dietary supplements. Presumably any between-batch difference in the "typical" weight of chickens is due to varying feed supplement effectivenesses. To look for such variation, we want to compare the different batches as to typical value -- be it median or mean.

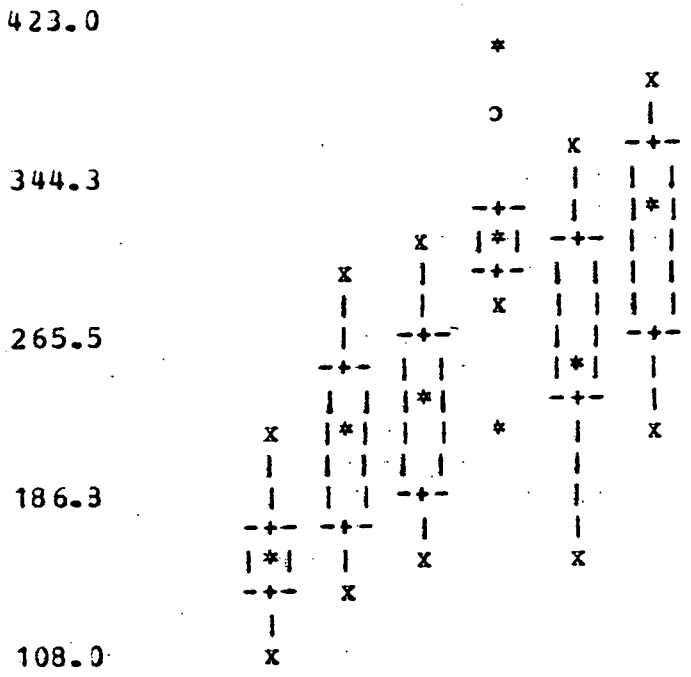
```
     * load sys chickwts
     * list
chickwts        array(14,6)
     *
```

Notice that chickwts is a 14 by 6 matrix of numbers. This corresponds to 14 individuals per treatment with 6 distinct treatments.

EXERCISE 12: Print out the array "chickwts." The stars indicate missing values -- cells where no observations were recorded. This is because up to 14 chicks were assigned per treatment, but not exactly 14 to each. Some treatments had as few as 10 chicks assigned to them. Can you think of other instances where the ability to indicate that data were not observed might be useful? Question: why is it better to be able to internally identify values as missing than to simply choose some "weird value", out of range of the data, and externally recognize that value as special? Hint: what happens when you reexpress data containing a weird value (e.g. -1), say by logarithms?

A useful graphical summary of this data can be obtained by a schematic plot [Tukey, p??] which reduces each of the batches to a single boxplot and prints out the boxplots 'in parallel', all with the same scale; this plot is produced by the "compare" command.

* compare chickwts



* condense chickwts

var.	n	MED	MAD
1	10	151.5	22.00
2	12	221.0	39.50
3	14	248.0	36.00
4	12	328.0	12.50
5	11	263.0	52.00
6	12	342.0	42.50

Here "condense" verifies the output of "compare": there does appear to be a real difference in the effectiveness of the feed supplements.

B. More Complicated Structure

As the situations we examine become more complicated, the techniques we use to solve our problems must also become more complicated. In the single batch case, we are looking for a summary of the form

$$x = m + e$$

where m is the model--a typical value--and e is the error--structureless and symmetric about 0. In the multiple batch case, we must ask for

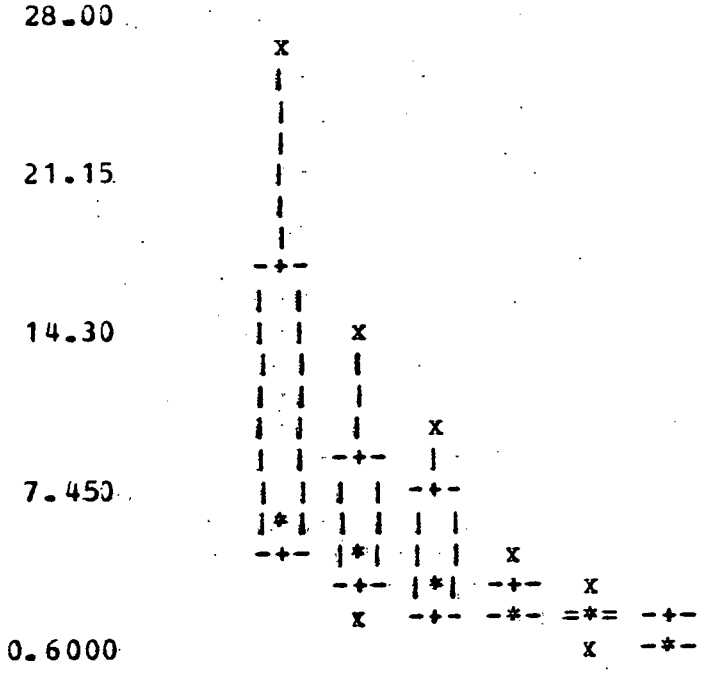
$$x(j) = m + a(j) + e$$

where j labels the circumstance (i.e. which treatment), m is the overall typical value for the data, a(j) is the treatment effect for the j'th circumstance and e is the error term -- again structureless and symmetric, but now also independent of circumstance. The last assumption is 'key', for on it rests our ability to estimate the parameters of the model. The errors must be of the same form in each batch -- same shape, same scale, same typical value (0). Obviously we can think of many situations where these assumptions are not met. In such cases it is our job to 'coax' the data into the correct form by reexpressing it until the assumption appear to be met.

A look at the exploratory paradigm in action can be had by attempting to 'coax' the dataset "illit" [McNeil, p. 42] into good form. "Illit" contains, for each of 9 geo-

graphical regions (Northeast, West, etc.) and each of 6 censuses (1900, 1920, ..., 1960), the illiteracy rate as a percentage of the population of that region. Since illiteracy has declined in general, and declined more in some regions (e.g. the 'deep South') than others (the Northeast), we expect that some reexpression of the data is necessary. To verify this hypothesis, we consider the comparison plot of the data. Here the batches are implicitly the 'decades':

```
* load sys illit
* compare illit
```



Both the prevalence of illiteracy and its interregional variation have gone down over time. Thus our modelling error is not independent of treatment (time). We can also consider each region as a 'treatment', which would require

treating the rows of "illit" as the individual batches. To do this we want to exchange rows with columns (also known as *transposing* a matrix) and "compare" again.

```
* compare(trn(illit))


  28.00
                                                     o

                                             *
  21.15

                                                             o
  14.30                                      -+-
                                            | |
                                            | |
                                    -+-     | |
                                    | |     | |     -+-
  7.450                             | |     |*|     | |
             x      -+-                     |*|     | |     |*|      o
            -+-     | |      x       *      | |     | |     | |              x
            | |     |*|     -+-             | |     -+-     -+-     -+-     -+-
            -*-     -+-     -*-     -*-     -+-      x       x      |*|     -*-
  0.6000     x       x       x      -+-                             -+-      x
```

Note that "trn()" is the "let" function for transposing the rows and columns of an array. From this schematic the huge variation in illiteracy rate for the Southeast is evident.

What reexpression should we use to get the different batches to all demonstrate the same basic shape and scale? McNeil suggests performing a *median* *polish* of the table, getting an additive fit of the form

$$illit(r,c) = m + a(r) + b(c) + e(r,c)$$

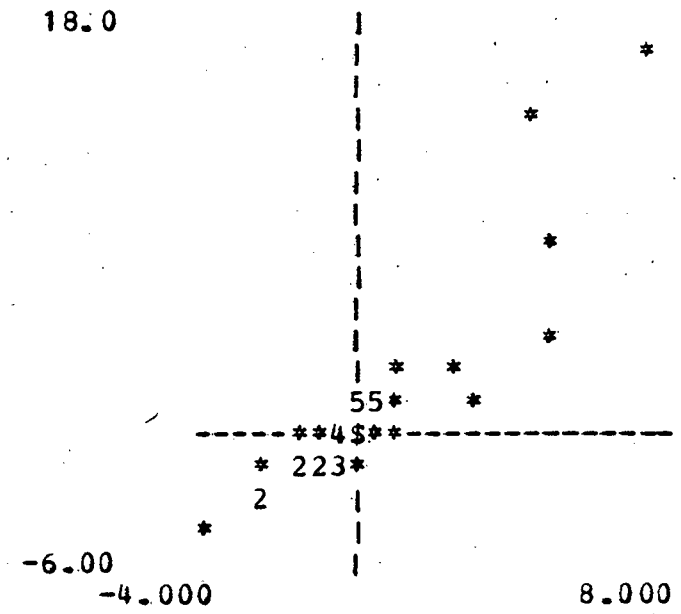
and then plotting

e(r,c) vs. a(r)*b(c)/m

if the plot shows a linear structure, he suggests fitting a line to the above pair and reexpressing the data by raising it to the power p = 1-s where s is the slope of the fitted line. Without justifying this procedure, we demonstrate its use:

```
     * medpolish illit >  dia @ d  res @ r1
  1.600
 0.5500
 0.7000


center    2.600
spread   0.7000

columns
       3.700       2.300       0.4000      -0.4000      -1.100       -1.900

rows
       0.0000      0.7000     -0.5500       -1.150       3.200        5.600
       3.700      -0.5000     -0.1000
```

```
    * scat 1
```




```
    * line d

intercept  0.7012e-01
slope       2.017
```

The fitted line has a slope of 2 so we will reexpress the data by taking the power 1-2 = -1.

```
    * let nill = illit^-1
    * medpolish nill > res @ r2
 0.1852
 0.3501e-01
 0.4025e-01
 0.4134e-01
 0.3990e-01
 0.3861e-01
 0.3922e-01
 0.4696e-01
 0.4302e-01
 0.5208e-01
 0.4298e-01
 0.5203e-01
 0.4297e-01
```

```
0.5208e-01
0.4297e-01
0.5208e-01
0.4297e-01
0.5203e-01
0.4297e-01
0.5208e-01
0.4691e-01
0.5331e-01
0.4691e-01
0.5086e-01

center  0.3982
spread  0.5086e-01

columns
     -0.2095     -0.1204     -0.4714e-01 0.4714e-01 0.1611     0.6132

rows
      0.1848e-01-0.1023      0.9224e-01 0.3059     -0.1889     -0.2601
     -0.2345      0.5512e-01 0.0000
```

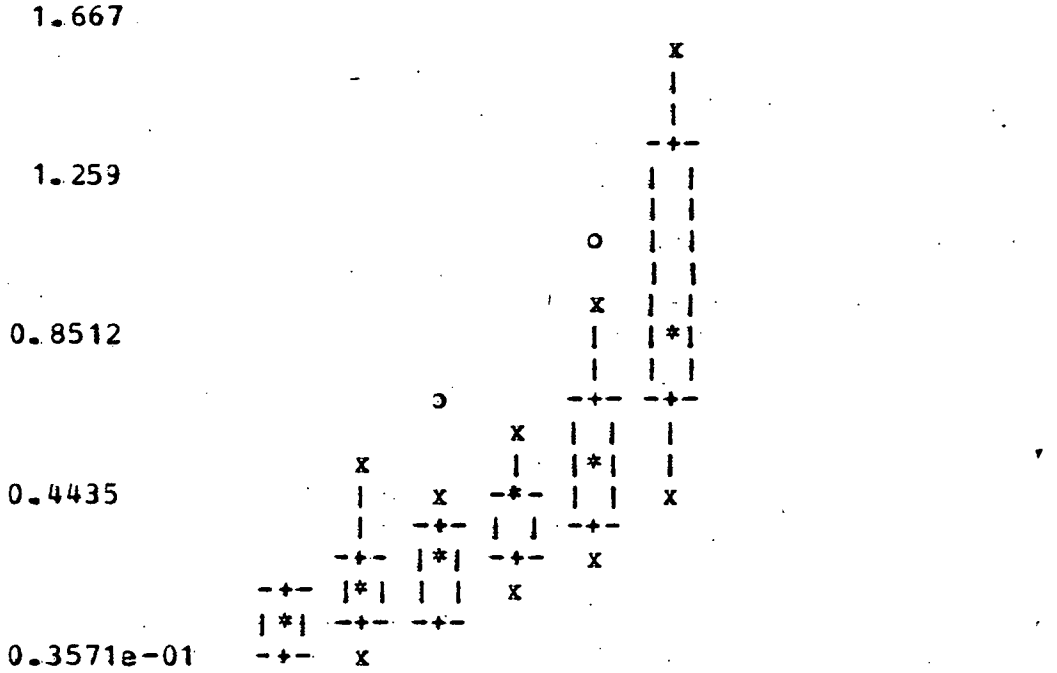
*

"Medpolish" prints a sequence of unlabelled numbers when it is first called; the ratio of the last to the first is in a sense the percentage of variation in the data left unexplained by the model. It is clear in this instance that our reexpression has markedly improved the explanatory power of the additive model. But we can go further: it has markedly improved the error's 'randomness'. The "code" program will display the residuals in a kind of contour map schematic plot: a 14 by 6 array of characters will be printed, with + and # standing for large resp. very large positive residuals; - and = standing for large resp. very large negative residuals; and . standing for small residuals:

```
     * code r1

- - . . + +
# # + - - =
# # # = = =
# + + . - =

     * code r2

- - . . # #
+ . . . . .
- - . . . #
= - + - # #
+ . . . . -
+ + . . . =
+ + . . - =
- . . . # #
```

The second coded display is by far less structured. Of course, without being technical we could have seen that the reexpression improved things: we could have just looked at the schematic plot of the "nill" array.

```
     * compare nill
```



Using ISP we have seen how the different tools can guide us to the correct form for expressing our data, how easily they can be fit together, and how well the tools work from a data-analytic point of view. To emphasize the notion of the tools fitting together, we examine the current contents of our active area:

```
     * list
d          array(54,2)
illit           array(9,6)
nill       array(9,6)
r1         array(9,6)
r2         array(9,6)
     *
```

While we started with just one array "illit", the various commands in ISP produced several others, and we used those arrays as input to other commands. Thus the active area forms a kind of temporary repository where we store information to be communicated between programs. This is the most powerful way in which the ISP tools fit together.

IV. Data Manipulation

Having examined ISP solutions to standard data-analytic problems, we now turn to the features of ISP designed for attacking problems on your own--problems in all likelihood much more complex than those illustrated earlier.

A. The ISP Environment

Workspaces* are used by ISP for storing data arrays, programs, or text files. The system itself, for example, has 3 workspaces: a data workspace (known as "sys" to the load command) which contains all the data we've used as examples and exercises here; a program workspace (known as "bin") which contains the programs that perform all the commands we've mentioned; and a text workspace which contains all the "explain" files for various commands and features of ISP. When you invoke ISP, the system creates two workspaces--"data" and "text"--where you can store data arrays and text files. It also creates an active workspace which exists only during the current invocation of ISP; in contrast, "data" and "text" and their contents will be around even after you log off of UNIX. Thus a typical data management strategy is to "save" whatever you've created in a given session and plan to use another time in the "data" or "text" areas.

---

* actually UNIX directories

```
% isp
     * load sys illit
     *
     * let jogit = log(illit*(100-illit))
     * list
illit           array(9,6)
jogit           array(9,6)
     * save jogit
     * list data
jogit           array(9,6)
     *
```

(another day)

```
% isp
     * load jogit
```

"Load" and "save" thus perform functions necessary to large and lengthy investigations which can only be conducted in pieces.

Note in the above fragment that "list data", as expected, causes a listing of the "data" area rather than the active one; other useful commands in this vein are "list text" (lists the "text" area) and "list sys" (lists the system data area--the datasets that can be loaded using the "load sys" command.)

B. Generating Data

We now know how to store and retrieve arrays we have created; but how do we create them in the first place? If all we could use in ISP were Don McNeil's data we'd be pretty limited!

Supposing we had a batch of numbers we wanted to analyze, the first step would be to get them in a text file.

The command "make" is useful for this purpose:

```
     * make numbers
1 2 3 -2
 4.5 6 7.001 8 99e12
^D
     * list
numbers         text
     * show numbers
1 2 3 -2
 4.5 6 7.001 8 99e12
     *
```

Note that now "numbers" is a text file which is exactly the image of what we typed in. Also ^D (control d) is the ascii EOF (End-Of-File) signal and tells "make" that our input to "numbers" is complete.

Once we have the numbers stored in a textual form on the computer, we need to convert them into an ISP data array. This is done with the command "read". In a sense, "read" is the opposite of "print"; "read" takes in text and produces an ISP array while "print" takes in an array and prints it out as text.

```
     * read numbers > n
     * list
n          array(6)
numbers         text
     * print n

    1.0000     2.0000     3.0000    -2.0000     4.5000     6.0000     7.0010
               8.0000  9.90e+13
```

With "read" we can, if we like, specify shape and specify missing values as options:

```
     * read {dims=3,3;missing=-1} > r
-1 2 3
4 -1 6
7 8 -1
^D
     * print r

*********    2.0000    3.0000
   4.0000 *********    6.0000
   7.0000    8.0000 *********
```

Note that if "read" is not given the name of an input teext file, it scans the standard input for text to be typed in at the keyboard. Realize also that the shape of the data array produced by "read" is independent of the visual shape of the input text: spots in the array are filled in row-major order from the stream of text input.

```
     * read {dims=2,2} > bogus
1 2 3
50.62301
^D
     * list
bogus           array(2,2)
     * print bogus

    1.0000    2.0000
    3.0000   50.6230
```

Finally if you already have a text file with numbers you'd like to analyze, "read" will accept a full UNIX pathname and produce the corresponding data array, so you needn't always type in data which is available in machine-readable form already. At Princeton, our computer has a phone link to an IBM 360 and can receive card-image files over the phone which it places in the UNIX directory "/usr/91d/punch". If

you had sent over this phone link some data you already had in 360-readable format, you might try something like

```
* read "/usr/91d/punch/MYDATA" {dims=96,13} > mydata
```

The "" are necessary to indicate a full UNIX pathname is being used.

C. Finding Out What You've Done

The most important factor in data management is remembering the manipulations you've performed on your data. Inevitably the question of which datasets you have loaded, created, saved or deleted comes up; it is important to verify that you and ISP 'understand' each other. The "list" command is obviously the most useful ISP verb in this regard. Remember that "list" will describe, in addition to your active workspace, the contents of your "data" and "save" areas as well as the system's data area ("sys").

Another aid in remembering what manipulations have been performed on your data is to use the "explain" facility. For example, if you had a need to document the contents of a variable "precious" you had created, a plausible command sequence might be

```
     * make doc precious
"precious" contains, for 1927 to 1977,
     col 1: Yearly GNP for Fredonia
     col 2: Yearly Widget production
     col 3: Yearly gadget consumption
^D
     * explain precious
"precious" contains, for 1927 to 1977,
```

```
     col 1: Yearly GNP for Fredonia
     col 2: Yearly Widget production
     col 3: Yearly gadget consumption
     * list text
precious  doc
     *
```

By using "make" with the "doc" option, you can create permanent documentation for any object you like.

D. Cleaning Up

The tools of ISP allow you to create a large number of arrays in a short period of time. Unfortunately, the UNIX system has a finite amount of disk to store these arrays; often you will have to clean up after yourself, and not just for space reasons--it's hard to keep track of an active workspace with a large number of often cryptically-named arrays ("x", "y", "z", etc.)

The "delete" command gives the user the necessary ability to remove unwanted or unnecessary files.

```
     * list
x          array(8)
y          array(8,8)
zebra           array(64,8)
     * delete x y
     * list
zebra           array(64,8)
     *
```

In an alternative usage, the argumentless invocation of "delete" interactively deletes files via prompting.

```
     * list
gofer           array(125,3)
gopher          array(125,5)
     * delete
```

```
gofer ?y
gopher ?n
     * list
gopher          array(125,5)
     *
```

FInally, note that the same effect as typing

```
     * exit
% isp
     *
```

that is, the effect of starting over with a "clean" active area, can be obtained by

```
     * delete all
all ?y
     * list
     *
```

There are also times when it is convenient to clean up the "data" or "text" areas. The "unsave" command is available for this function; one merely specifies which area needs to be cleaned up, then provides a list of names of files to be removed. If no area name is specified, "data" is assumed.

```
     * unsave text a.doc b.doc data a b
     * unsave x y
```

In the first line "a.doc" and "b.doc" are removed from "text" while "a" and "b" are removed from "data"; in the second line "x" and "y" are removed from "data".

V. Advanced Data Analysis Tools

In addition to the essential exploratory tools introduced earlier, ISP offers many more sophisticated statistical procedures. Rather than giving detailed descriptions of these procedures, the theory they are based on and the conditions under which they are useful, we will suggest through examples and exercises some analysis sequences you might like to try.

A. Time Series Analysis

For a discussion of the relevant theory, see [Bloomfield, 1976]. ISP offers the fast Fourier transform with the "fft" command; a variant "pgram" is useful for estimating power spectra.

EXERCISE 13: The following sequence will graphically display the autocovariance of the "sunspots" data introduced earlier.

```
* load sys sunspots
* pgram sunspots > p
* fft p {inverted;} > ac
* six ac
```

(This approach uses the inverse Fourier transform of the power spectrum to calculate the autocovariances.) How strongly does one month's sunspot number depend on another's? (ac[i]/ac[1] gives the correlation of month t with month t+i-1.) Is there any seasonality (i.e. is the lag 12 correlation greater than, say, 2/sqrt(180))? How about on the dataset s=sqrt(sunspots)? (It often makes sense to take the square root of counts data to make the distribution symmetric.)

The verbs "cohphs" and "xpgram" compute coherence and

phase and cross-periodogram respectively. Try "explain" on these names. See [Bloomfield, Ch. 10] for a quick explanation of the theory.

B. Classical Least-Squares

The following commands would provide the meat of most statistical packages; in ISP they are added almost as an afterthought. There is good reason for this--classical description of batches and relations is very much oriented towards testing hypotheses, where the form of the model generating the data is well known. If we ever really knew the model generating our data, we certainly wouldn't need an interactive computer to help us study it; consequently we assume the model relatively unknown and exploration necessary. Nevertheless...

The "stat" command provides the classical statistics for 1 or several batches: means, standard deviations, min. and max. "Corr" provides, in addition, the correlation matrix.

EXERCISE 14: The 24 by 4 array "demopct" [McNeil, p.??] in the "sys" area has, for each of the '60,'64,'68 and '72 elections, the percentage of Democrat vote in each of 24 urban states. Find the basic statistics and correlation matrix for this data using the "corr" command. Which election years are 'most alike' (largest correlation)? Which 'least alike' (smallest correlation)?

The "eigen" command computes the eigenvalues and eigen vectors of real symmetric (e.g. correlation) matrices. See

[Rao, p. ??] for an explanation of the spectral decomposition of a data matrix.

Using the "demopct" data from Ex. 14, we will characterize the elections by calculating their 'principal scores.' First we need to compute the correlation matrix:

```
* corr demopct {quiet} > cm
```

(Here the "quiet" option suppresses the printing of the statistics you should have gathered for Ex. 14.) Second, we obtain the eigen- values and vectors:

```
* eigen cm > vals @ va vecs @ ve
```

var	e-val	pct	e-vect
1	3.292	82.3	
2	0.456	11.4	
3	0.190	4.8	
4	0.062	1.5	

```
* list
cm          array(4,4)
demopct     array(24,4)
va          array(4,1)
ve          array(4,4)
*
```

Now, to compute principal scores, we post-multiply the original data matrix by the matrix of eigenvectors; this is done using the '#*' matrix multiplication symbol in "let":

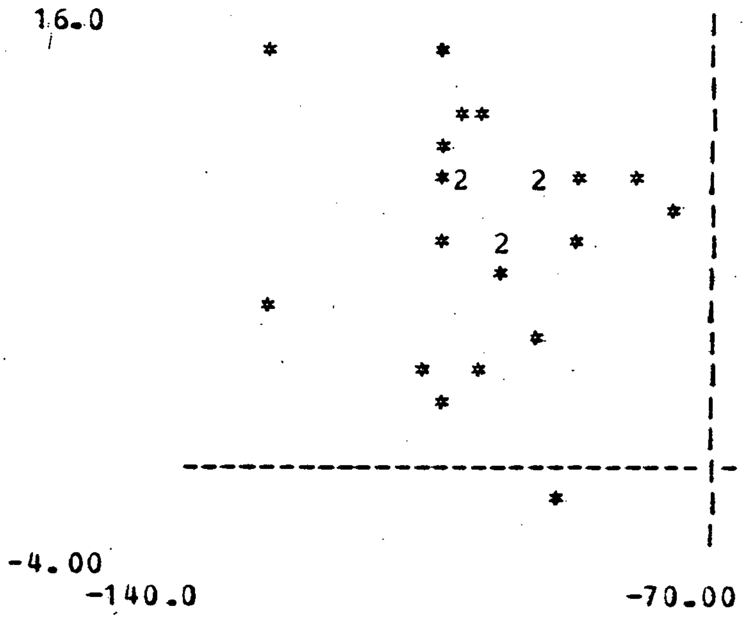
```
* let pscores = demopct #* ve
```

To plot the first set of principal scores, we use "stem-leaf":

```
      * stemleaf (pscores[*,1])

column #  1


-14|
-12|09
-10|0933775542200
-08|95543092
-06|8



column #  2


-00|
-00|2
 00|233
 00|55788889
 01|0000000123
 01|55



column #  3


-02|
-01|6
-01|30
-00|9988888777766666555
-00|33



column #  4


-14|
-12|86
-10|82531
-08|8842986110
-06|8545
-04|841
```

To plot the first pair of principal scores, we use "scat":

```
* scat (pscores[*,1:2])
```



From these displays it appears that an urban state's democratic tendency over the past 15 years or so can adequately be summed by a single number--its first principal score. It also appears that these states make up a homogenous group, since the set of first p-scores is Gaussian-like in shape.

EXERCISE 15: Perform the preceding type of analysis on the 50 by 4 array "crimes" [McNeil, p. 132] which contains, for each of the fifty states, murder, assault, and rape (as measured in arrests per 100,000) along with the percentage of the population living in urban areas. You may wish to reexpress the counts and percentages data before doing the analysis. How many principal scores do you need, if you wish to describe the data adequately?

The "regress" command will fit a multiple linear regression model by the least-squares procedure. Try fitting the model

(% demo in '72) = a + b*(% in '60) + c*(% in '64) + d*(% in '68)

Interpret the coefficients in terms of the candidates and issues of the day, if you can.

C. Robust Analysis

Robustness theory (as developed in the last 15 years or so) provides the data analyst with a set of tools that are essentially as "efficient" and "powerful" as those of the least-squares genre, without being as sensitive to 'disutopianities' as the Classical techniques.

The central idea behind the robust procedures supplied with ISP is that of weighting observations in accordance with an a-priori measure of credibility relative to the assumptions of normality and of the appropriateness of the model being fit. The a-priori measure is provided by the biweight function; an observation is assigned weight 0 if a datapoint is apparently an 'outlier' and a weight near 1 if the observation seems to come from the 'Gaussian' center of the data. See [Tukey and McNeil, 1975] for a description.

The "biweight" command gives output like "condense"--center and spread statistics--but these are computed according to the biweight algorithm. "Biweight" will also output the weights it calculates in the operation of its algorithm; these can be used data-analytically.

EXERCISE 16: Find a transformation to symmetrize the dataset "islands". Using the weights from "biweight",

determine if Greenland is much more a 'credible' island than Australia. Hint: you will first want to sort "islands so that Australia is the 7th element of the array, and Greenland the 8th--try

```
* sort islands {descend} > s
```

which sorts the island sizes in descending order.

The "rsvd" command performs a robust singular-value decomposition of a data array. For theory behind the svd see [Rao, ??].

EXERCISE 17: Compute and plot the first two principal scores of "demopct". Do the weights suggest 'outliers' among the urban states? Hint:

```
* rsvd demopct > v @ ve w @ wt
* let pscores = demopct #* ve
```

The robust commands for multiple batches are "oneway" and "twoway". The first fits column effects only; the second, both row and column effects.

EXERCISE 18: Try

```
* oneway chickwts
```

and

```
* twoway (log(illit))
```

Refer the given statistics to an *F*-table; do the different chicken feeds differ in effectiveness? Does the additive model fit the log(illit) data?

The "robust" verb provides for a multiple regression. Fit a multiple regression model to the "crimes" data.

3. ISP System Documentation

Department of Statistics
Princeton University

ISP SYSTEM DOCUMENTATION

*** these are all documented in explain files ***

GENERAL INTEREST

isp	explains philosophy of system. lengthy
changes	explains the changes from the old isp
syntax	syntax of isp commands
semantics	semantics of isp commands
macros	how to make & use macros
strings	how to make & use strings
results	how to use the results feature
options	how to specify options to a command
document	how to understand the documentation
oldisp	how to run a command written for the old isp

SYSTEM COMMANDS

make	create a text file
edit	edit a text file
read	read a text file, converting to isp variable
save	save an isp variable or text file for future use
load	load a previously saved variable or file
delete	delete active variable(s).
unsave	unsave saved objects
list	list contents of active, data, text, or system areas
print	print variable or string
let	algebraic and manipulative capability
explain	how to explain something
echo	echo command arguments
select	logical subscripting

DATA ANALYSIS

boxplot	boxplots
stemleaf	stemleafs
code	coded displays
fivenum	fivenums
condense	Med. & MAD
biweight	biweights
compare	compares
scat	scatterplots
line	line fits
medpolish	table polishes
six	six line plots

ROBUST FITTING

robust	regression
1eway	anova
2way	anova

LEAST SQUARES

stat	statistics

corr	correlations
regress	regressions
eigen	eigenvalues real symmetric matrix

TIME SERIES

smooth	Tukey's smoothers
fft	Fast Fourier Transform
pgram	periodogram
xpgram	cross periodogram
cohphs	coherence and phase

UTILITIES

gplot	gsi,tek graphics
trn	matrix transposes
sort	sorting

BIWEIGHT

USAGE:

*biweight var [{options}] [> results]

DESCRIPTION:

Biweight computes the standard biweight location and scale estimates for one or several batches of data. The biweight is an adaptive weighted mean (e.g. 'explain resistant'); both the sum of the computed weights and the individual weights are of interest.

OPTIONS:

x=int array	Specifies which columns to compute biweight for; Default is 1:nc.
c=float	Specifies how many multiples of sigma a point must lie from the center to be considered an outlier. Default is 6.

RESULTS:

[out @]	nc by 3; contains computed center, scale, and sum of weights for each column biweighted.
[wts @]	nr by nc; contains computed weights for each observation in each biweighted column.

EXAMPLES:

```
*load sys chickwts
*compare chickwts
*fivenum chickwts
*biweight chickwts > z; scat z;
```

In this example, each column contains the weights of chicks under a different diet plan. The scatterplot shows the linear relation between typical weight and scale for each feed.

BOXPLOT

USAGE:

*boxplot var [{options}]

DESCRIPTION:

Boxplot displays Tukey's boxplot for one or several batches of data. Var may be either a matrix whose columns are to be plotted or a single row vector.

OPTIONS:

x=int array	Specifies which columns to be plotted. Default is 1:nc.
unravel	Specifies that a multicolumn file should be viewed as a single variable rather than as several. Not default.
width=int	Specifies page width to use in # of characters per line. Default is 64.

EXAMPLES:

```
*load sys rivers
*boxplot rivers
*let re = rivers ^ -.5
*boxplot re
```

In this example, rivers contains the lengths of the world's 141 longest rivers. Inverse square roots make the lengths distribution nearly symmetric.

CHANGES

The principal changes from the old isp are as follows:

1. Inputs can be tagged. That is, a command with multiple inputs can recognize the different inputs by their tags. for example:

```
* fft real @ r imag @ i > real @ rt imag @ it
```

generates a fourier transform of a complex series with real part r, imaginary part i, and the transform so generated will have real part rt, imaginary part it.

2. Strings in macros can be generated in new ways. In addition to the old $0-$9 formalism for argument passing to macros, new commands for getting tagged inputs and tagged outputs in and out of macros, for generating unique local names and for getting options lists to macros into internal strings are available.

3. If the first word in a line is in "", it is executed as a unix pathname. For example:

```
* "/bin/mv" a b
```

renames a as b.

4. To redirect standard output, use '>>' rather than the old '> pr @' convention:

```
* print x >> '/dev/tty8'
```

this prints the contents of x on tty #8.

Coalesce

Usage:

coalesce x {options} > result

description:

condenses file so x-values are unique. condenses y values with common x-values by one of many choices.

options:

x=int array	x variables to be lexicographically sorted & uniquified
y=int array	y to be carried along
eps=float	equality criterion. default 10^-6

COHPHS

USAGE:

```
* fft ser > fr fi
* xpgram fr fi > xp
* smooth xp > sxp
* cohphs sxp > coh phs
```

DESCRIPTION:

"Cohphs" computees coherence and phase from the cross-periodogram matrix output by "xpgram."

INPUT:

[xpgram ∂]	the (presumably) smoothed output of "xpgram."

OUTPUTS:

[coh ∂]	n by p(p-1)/2 matrix with subdiagonal coherence matrix for each row's frequency stored in lower triangular form.
[phs ∂]	n by p(p-1)/2 matrix with subdiagonal phase matrix for each row's frequency stored in lower triangular form.

COMPARE

USAGE:

*compare var [{options}]

DESCRIPTION:

Compare plots Tukey's schematic plot of a multi-column variable via parallel boxplots tied to the same scale. Useful for comparing centers and spreads in one and two way tables.

OPTIONS:

x=int array	specifies which columns are to be plotted. Default = 1:nc, where nc = # of columns in var.
height=int	height of the plot in lines. Default is 30; 55 gives a full-page plot.

EXAMPLES:

```
*load sys warpbreaks
*compare warpbreaks
```

Warpbreaks is a nine by six file comparing the breakage counts of six different looms using nine different yarns. The first loom isn't very good; the last one is great. The others are fair to middlin.

```
*load sys illit
*compare illit
*compare (trn(illit))
```

Illit contains illiteracy rates for several regions of the US for the decades 1910,20,30,40,50,60. Compare shows that in this period, not only has median illiteracy dropped, but also, the different regions of the US have improved to a common level - virtually no illiteracy. As the second plot shows, certain regions over the decades in question have had consistently higher illiteracy rates.

CONDENSE

USAGE:

```
* condense [in @] x [options] [> results]
```

DESCRIPTION:

"Condense" computes, for one or several batches, the median and meedian absolute deviation from the median (MAD), useful as measures of location and scale.

INPUT:

[in @]	nr by nc array.

OPTIONS:

x=int array	columns in array to "condense;" default is 1:nc.
unravel	treat array as 1 by nr*nc.

STAT, CORR

USAGE:

```
*stat var [{options}] [>results]
*corr var [{options}] [>results]
```

DESCRIPTION:

Stat gives means, standard deviations, extreme values, and intervariable correlations, if desired.

OPTIONS:

x=int array	columns to be stated. Default: all.
unravel	view multivariable set as 1 variable.
quiet	print no evil
long	force printing of correlation matrix if commad was invoked as_stat

RESULTS:

[stat @]	nc by 5; means,sdev,min,max,sum sq dev.
[corr @]	nc by nc; correlation matrix

EXAMPLE:

```
*load demopct
*stat demopct {long}
```

Demopct is the percent democratic in the presidential elections of the Imperial era 1960-1972, for 24 industrial states. Columns are elections; rows are states.

```
*load sys sunspots
*let
* s1 = (),sunspots[1:179]
* s2 = (),s1[1:179]
* s3 = (),s2[1:179]
* s = trn(sunspots),trn(s1),trn(s2),trn(s3)
^D
*scat s {x=3*1;y=2:4}
*corr s
```

Here sunspots is a variable containing the Vienna sunspot numbers; the correlation matrix produced will give the correlations between this months sunspot count and that of each of the previous three months in row one. The scatterplots display visually the information represented in the lag 1, 2, and 3 correlations.

Delete - delete active variables

usage:

*delete files

or

*delete [all | prompt |]

description:

In the first usage, files is a list of active variables. If the variable does not exist, ".text" is appended and an attempt is made to delete that file. Enclosing the filename in "" suppresses the .text appending.

In the second usage, all implies all files are to be deleted (after verification!). Prompt implies that each file's name be printed and a response by the user of 'y' or 'n' determines its fate. Calling delete without an argument is equivalent to "delete prompt".

examples:

*delete a b c "crud"
*delete all
all? y
*delete prompt
x? y
yab? n
x.text? n

DOCUMENT - syntax

DESCRIPTION:

The syntax for documentation is generally what you expect. However there are a few subtle points.

1. Items enclosed in [ ] are optional; that is, you don't type the stuff between braces if you don't need to. e.g. in

*six var [{options}]

the { options } is only typed in if you have nonstandard options to invoke. for example, if you wanted the plot on 120 column instead of 64, you might try

*six var {width=120;}

but

*six var {width = 64}

and

*six var

are equivalent.

2. In the documentation for system commands, alternation is used so that [a | b | c |] might be seen. For '|' read 'or' then this reads a or b or c or nothing at all. For example, one can load items from his data, text, or from the system's data areas. Thus he can prefix a name, according to the load documentation, with [data | text | sys |] with nothing at all indicating data. So these might be valid:

*load text a b c
*load sys warpbreaks illit

and these would be equivalent:

*load data frog toad newt
*load frog toad newt

3. In the documentation for options, underlined items like int_array may be seen; these denote specific items explained under 'options'. Thus we have the following examples:

int array	13,5:4,9:11,3*-1 13,5,4,9,10,11,-1,-1,-1
float	1.2345e6; 789.0564
char string	abcdefghijklmnop
char array	a,b,c,3*d,+,-,*,^ 1,b,c,d,1,d,+,-,*,^

4. In the results section of documentation, tags are enclosed in [ ]. This indicates that if you order your outputs in the order

the options are indicated in the documentation, then you needn't type the tags themselves. For example, for twoway the standard order for outputs is (res,dia,ref,cef,wts) so that these are equivalent:

```
*twoway mytable > res @ r dia @ i ref @ re cef @ ce wts @ w
*twoway mytable > r d re ce w
```

while these aren't:

```
*twoway mytable > wts @ w cef @ c ref @ r dia @ d res @ rz
*twoway mytable > w c r d rz
```

since the first has its items listed not in standard order. In the second line, the residuals would be put in w, diagnostics in c row effects in r, column effects in d, weights in rz - precisely the reverse of the line before, where residuals go in rz, etc.

Edit - edit a file

usage:

*edit [text | here |] file

description:

Edit calls the UNIX editor (see Ed Tutorial, Documents for Use with UNIX) to look at a text file. If the file does not exist, edit creates it; whether it is put in the text or active area depends on whether "text" is specified.

Enclosing the filename in "" permits use of an explicit UNIX pathname - e.g. "/mnt/usr/wierdo/isp/text/myownjunk".

examples:

*edit junk
(creates a text file named junk in the active directory)

*edit text junk
(creates a text file named junk in the text directory)

*edit here junk
(same as edit junk)

EIGEN

USAGE:

*eigen sym_mat [{options}] [>results]

DESCRIPTION:

Eigen gives the spectral decomposition of the real symmetric (e.g. correlation) matrix sym_mat. It prints eigenvalues and, if desired, eigenvectors.

OPTIONS:

x=int array	Compute sp.d. on submatrix with specified rows and columns. Default is x=1:nc.
long	Print out eigenvectors as well

RESULTS:

[vals @]	1 by nc matrix of eigenvalues
[vecs @]	nc by nc matrix with eigenvectors for columns

EXAMPLE:

```
*load sys demopct
*corr demopct > c
*eigen c {long}
```

Demopct gives the percentage democratic vote in the presidential elections of 1960-1972 for 24 east & midwest states. Eigen tells how similarly the vote behaved across years for the same states.

Explain - explain an object

usage:

```
*explain arg
```

description:

Explain looks first in the user's text area for a file named "arg.doc", and if it finds one, prints it. If not, it looks in the system text area.

One can explain his own objects, like datasets, by typing

```
*edit text object.doc
```

where object is something the user wants to keep documentation about.

If explain fails to find any documentation about an object, it reports this.

example:

```
*explain explain
```

(this types out the file you are now reading)

```
*make text mycrud.doc
mycrud is file of percapita crud consumption
statistics in the U.S. from 1920 to 1970
*list text
mycrud.doc      text
*explain mycrud
mycrud is file of percapita crud consumption
statistics in the U.S. from 1920 to 1970
*
```

(this shows how one creates perpetual documentation for a file or keeps notes for one self)

FFT, PGRAM

USAGE:

```
* fft   [real @] ri [imag @] ii [ {options} ] [> results]
* pgram [real @] ri [imag @] ii [ {options} ] [> results]
```

DESCRIPTION:

"Fft" computes the complex fast Fourier transform of a complex argument. If either part of the input is omitted, it is assumed to be zero. "Pgram" is an alias whose default output is the calculated periodogram.

INPUTS:

[real @]	Real part of the input argument. If omitted, taken to be zero.
[imag @]	Imaginary part of the input argument. Zero if omitted.

*** One of these two must be provided ***

OPTIONS:

taper=float	Series tapering length as a proportion of series length; default is 0.
invert;	specifies that an inverse transform is to be performed.

OUTPUTS:

[real @]	real part of transform.
[imag @]	imaginary part of transform.
[pgram @]	periodogram (= real^2 + imag^2)

FIVENUM

USAGE:

*fivenum var [{options}] [>results]

DESCRIPTION:

Fivenum prints, for each of one or several batches, the extreme values, quartiles, median, and N of the batch.

OPTIONS:

x=int array	Specifies which columns to fivenum. Default: x=1:nc.
unravel	Treat multivariable file as single variable.

RESULTS:

[out @]	nc by 3; rows are calculated N, extrema, quartiles, and median for each column.

EXAMPLE:

```
*load sys illit
*fivenum illit
```

Illit is a file containing, as columns, illiteracy rates of several regions of the US for given decades. Fivenum will display a marked tendency for the spread in illiteracy rates to drop over time (i.e. as we go from one column to the next); see also scat, compare.

Gplot -- plot on gsi300 or tek4013

USAGE:

gplot input {options}

DESCRIPTION:

Gplot produces a number of plots on a single set of axes. The plots may be joined up or not, and may be produced on a gsi300 or tek4013 terminal.

OPTIONS (defaults in parentheses):

```
x=list   x variables (1)
y=list   (2)
xmin=val          (found from data)
xmax=val
ymin=val
ymax=val
ch=<list of chars>      plotting characters (o)
join=<list of y or n's> y for a plot to be joined up (n)
tek      terminal is tek4013 -- default is gsi300
nosort   omit attempt to optimise motion
index    plot the specified y variables against their indices
```

IRIS DATA

This is Fisher's well-known iris data set used in his initiation of the linear discriminant function technique. The data consists of 150 flowers, 50 (indexed as 1 to 50 by dimension 2 of iris) from each of the three iris species: Setosa, Versicolor and Virginica (corresponding to elements 1 to 3 respectively of dimension 1 of iris). The responses were the four measurements of Sepal Length, Sepal Width, Petal Length and Petal Width (corresponding to elements 1 to 4 of dimension 3, respectively).

EXAMPLE

Let a(*)b denote the Kronecker product of a and b: [a(i,j)b]; let P(x) denote the orthogonal projection onto the range of x and let vec[x] denote column vector formed by stringing out rows of matrix x, one by one. Under the one-way MANOVA model:

```
   y    = I(*)1    b  +  e   (vec[e] is N(0,     I (*) V), V unknown)
150:4   3:3 50:1  3:4 150:4              600:1 150:150 4:4 <-- dimensions
```

a test for the hypothesis of equality of species characteristics, namely

$$H0: cb=0 \qquad c = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \end{pmatrix}$$

is the largest eigenvalue of

$$[y'(I-P(a))y]^{-}[y'P(a(a'a)^{-}c')y]$$

(where a denotes the design matrix: I(*)1). If L denotes this largest eigenvalue, L/(1+L) can be used as a test statistic to enter the Heck charts with s=2, m=.5 and n= 71 degrees of freedom. We would find it significant at the 1% level. The eigenvector corresponding to L is that linear combination of the observations which maximally discriminates among the species. We calculate this eigenvector by noting that if x(n) is defined recursively as x(n+1)=Ax(n), then x(n) converges to the eigenvector of A's largest eigenvalue.

```
      * load sys iris
      * let y=iris(150,4)
      * let a=int((iota(y[*,1])-1)/50); a=(a==0),(a==1),(a==2)
      * let c=1,-1,0,0,1,-1; c=c(2,3)

      * let reg=inv(trn(a)##a)##trn(a); b=reg#*y
      * let e=y-a#*b; E=trn(e)##e
      * let H=trn(c#*reg); H=H#*(inv(trn(H)##H)#*(trn(H)#*y))
      * let H=trn(H)##H
      * let EH=inv(E)#*H

      * let x=trn(1,1,1,1); x=EH#*x; x=EH#*x; x=EH#*x
      * let x1=EH#*x

      * print (x1[1,1]/x[1,1]) (trn(x1/(trn(x1)##x1)^.5))
```

BIBLIOGRAPHY

Fisher, RA (1936): The Use of Multiple Measurements in Taxonomic Problems, Annals of Eugenics, vol 7, pp 238-249.

Morrison, DF (1976): Multivariate Statistical Methods, McGraw-Hill, New York, pp 180-182, 199-200, 242-243.

ISP - New ISP release

USAGE:

```
        (from UNIX)
% isp [-rsp] [file] [args ...]
        (from isp)
        *isp [-p] file [args ...]
        *macro_name [args ...]
```

DESCRIPTION:

ISP is an interactive statistical package run by a shell-type program that parses and interprets a general language for manipulating objects (e.g arrays) that are at the core of statistical work. These objects are handled within a protected environment designed to facilitate orderly conduct of statistical analysis. The philosophy behind isp is that the structure of computer tools available to the data analyst is a powerful, possibly determinate, influence on his (her) statistical investigation. Thus it tries to initialize a model for data analysis and in fact generalizes the language accepted in McNeil's (Interactive Data Analysis, Wiley, 1977) model.

The features of isp relevant to data analysis fall into three broad groups:

I) The Language. With few exceptions, the structure of an isp command is given by

```
command [input variables] [ {options} ] [ > results ]
```

e.g.

```
fft x > x.r x.i ;
regress (x,y) > resids
print x (log(x)); stemleaf (x ^ .5) {scale = 2}
```

Here "command" is the name of a subcommand in the isp system (see III); "input variables" are, in general, 0 or more arrays or array-valued expressions; "options" are command specific parameters which represent user-controllable factors in the operation of the command; and "results" specify where the 0 or more array-valued results of a command are to be stored.

Hopefully, this language is large enough to accomodate the most commonly-encountered types of operations desired by an analyst; since it has essentially been given a trial run in McNeil's book, it certainly has been shown sufficient for most interactive data analysis needs.

II) The Environment: Isp establishes a protected environment which interaction is to take place, with three logically separate environments:

a) the active area: where the user is 'placed' upon entering isp; where all variable manipulation and reexpression takes place; where data must be accesible from for statistical commands

to operate on them. The active area literally 'is created' upon entry to isp and 'disappears' upon exit.

b) the data area: where the user saves data from one session to another; the only direct contact with files in this area that a user can have are to either copy data into it, copy data out of it, or delete data from it. Thus, the data area is relatively quiescent.

c) the text area: where the user's nonnumeric information is kept - for example, text documenting a user's own data; text describing a user's own (FORTRAN or C) isp-compatible programs; or text containing a list of isp commands, with symbolic parameters, that the user can invoke as a command rather than type in each time he wishes those commands to be executed as a group (i.e. macros). Access to this area is of the same restricted nature as that to the data area, hence this area is quiescent also.

The data and text areas are permanent, in the sense that they and their contents continue to exist even when the user is not using the isp program, or not logged into UNIX. The active area is transient, and exists only while the isp program is in use. Thus, by saving active objects into the data or text areas, the user can terminate a session and return later to restore exactly his original environment by reloading the saved items.

There are two other areas of importance which the user may access from time to time. One is the system data area, a "library" of data arrays, all taken from McNeil's book, and all illustrating some basic feature of exploratory data analysis. The other is an area the user may create and place his own specialized commands in... this area is his "bin".

III) The Commands: The most important feature of a computer language is its semantics - namely, the actions a given statement of the language may generate. Isp's commands are heavily oriented towards statistical use, which means a) that they have lots and lots of options and parameters to be set and b) that they generally focus on processing a single array (e.g. a two-column array to generate a y vs x plot). The user who finds a need for other commands can easily add his own or structure existing commands into new ones with macros.

USE OF ISP: The simplest way to use isp is to learn by doing. Simply invoke "isp" from UNIX, then begin by typing "explain isp" and go from there. The "explain" subcommand will print out information and examples for almost any command in the system; hence a good idea of the isp's capabilities can be gotten by trying the different subcommands on illustrative data from McNeil's book which the system will provide you. As a consequence, a copy of that book may come in handy.

Reexpressing Variables

As anyone who has seen any of John Tukey or Don McNeil's propaganda should know, there are lots of instances in which reexpressions of data are useful: we can take logs, powers, combine variables arithmetically and so forth using the "let" program in ISP. The syntax for most of these operations is like Basic or Fortran. A few examples:

```
z=a+b*c+(e/(f+q));
x=log(y);
z=exp(a*b+1.);
x=sin(abs(int(rnd(y))));
a=astronauts/(1-astronauts^.5)
```

Note that spaces are irrelevant, parentheses may be used to alter the order of evaluation, and that exponentiation is indicated by '^' rather than '**' as in Fortran. To use let, type

```
let expr ; expr ; ...
```

or simply "let". If you type "let" without any argument, it will return with "* " and wait for an expression, so that this is an economical way to do many reexpressions in a row:

```
*let
* x=log(mydata);y=x+.01;
* w=x/y+y/x;
* z=(x+y)^w;
* <^D>
```

Here "let" takes an expression, evaluates it, and when finished, returns with "* ". It continues in this way until you type ^C or ^D.

A major shortcut is provided by the use of implicit calls to let: in an ISP expression where an argument is enclosed in parentheses ("()") it is passed to let and the result sent to the command:

```
print a (log(a)) (exp(a))
boxplot (ln(a))
```

Manipulating Variables

An important part of handling data is the manipulation f it into usable forms. For example, we may wish to break a single column out of a large matrix and deal with only that column for a while, reexpressing it, and then replacing the original column with the reexpressed one. We may also wish to create variables which are permutations of the

values in other variables, and so on. Most of such tasks can be carried out with the help of subscripting. Suppose x is an array that has 20 rows and three columns. Then x[*,1] references the first column of x; x[1,*] references the first row; and x[*,*] references all of x. Thus in "let" we could say

```
let c=x[*,1]+x[*,2]/x[*,3];x[*,2]=log(c);
```

which would create a column vector "c" of length 20, containing, column 1 of x added to the ratio of columns 2 and 3 of x. In turn, column 2 of x would be changed to contain the base 10 logarithms of c. The ability to pick out columns and rows of a matrix is extended by list specifications in the subscript. The item x[1:20,1] is equivalent to x[*,1] and is also equivalent to x[1 2 3 4 5 6 7 8 9 ... 20, 1] or even x[1 2 3:10 11:18 19 20,1] - all pick out the first column of matrix x (admittedly, some are easier to type than others!). We can even get fancy and reverse the order of columns in a matrix:

```
x = x[*,3 2 1]; or
x = x[*, 3:1];
```

will do this for the matrix x mentioned above. Subscript expressions can be used in let just as freely as algebraic ones:

```
y[1:10] = x[10:20] + z[*,1] / x[1:10];
```

To create a matrix that is not some function of another matrix, you must dimension it and fill it with a constant:

```
x(10,10) = 0;
y(134) = 20;
z(10,10,2) = sin(1.717189)*log(43);
```

There are functions which do convenient things with their matrix arguments: "iota(x)" takes matrix x and returns another matrix of the same shape, but with the first n integers, where n is the size of the matrix, stored in the matrix in row major order:

```
let x(2,2)=0;y=iota(x);
print x y

0.0000 0.0000
0.0000 0.0000

.0000 2.0000
.0000 4.0000
```

It is often useful to be able to take columns and put them together into a matrix, or to add a new column to a ma-

trix; the operator "," does this. The previous example of reversing the columns in matrix x could equally well be done by

```
let x= x[*,3],x[*,2],x[*,1];
```

The "," operator takes an array of size (k,n,p) and another of size (k,n,p') and delivers an new array of size (k,n,p+p') by laminating the two together. "," may also be used as a unary operator; it takes a (k,n,p) array annd returns a (1,1,k*n*p) array with the same elements in row-major order in a row-vector.

It can also be useful to be able to reexpress variables logically, that is, based on whether a certain condition is true or false. To do this, "let" accepts logical expressions using the operators

```
>, >=          greater than, greater or equal
<, <=          less than, less or equal
==, !=         equal, not equal
&, |       and, or
!          not
```

e.g.

```
a>b & x[*,1]<10
x != () | y==()
```

(note that "()" is let's code for the missing value). Such operations are done an element at a time, meaning the operators give results the same size as their operands - for each element of x and y, "x>y" returns 1 if the element of x is greater than the corresponding element of y and 0 otherwise.

The main use for logical expressions is in let's if-then-else construct (equivalent to Iverson's "mesh" operator). For example, if all zeros in a file were to be changed to ones, we could type

```
x = x==0 ? 1 : x;
```

The syntax of the operator is

```
logical_expression ? if_true : if_false
```

whenever logical_expression is true, the expression takes on the value of if_true; otherwise it takes on the value of if_false. Again, as in the case of algebraic and logical operators, the if-then-else is evaluated elementwise. A final example: the following statement changes all values of 'x" to missing whenever they are negative or zero.

```
let x = x<=0 ? () : x;
```

LINE

USAGE:

```
* line [in @] x {options} [> [res @] z]
```

DESCRIPTION:

Line performs a quick, exploratory quality linefit on the data contained in x. The predictor variable is taken to be the x= column, the predicted variable is the y= column. The residuals file contains the predicted and residual values for each observation.

OPTIONS:

x=int	column containing predictor variable. Default=1.
y=int	column containing predicted variable. Default=2.

EXAMPLE:

```
* load sys cars
* scat cars
* line cars > z
* scat z
```

Cars contains initial speed and stopping distance for each of fifty drivers. The residual plot displays perhaps a mild nonlinearity.

List - list the contents of a directory

usage:

list [here | text | data | sys |]

description:

Given a list composed of the names in the above brackets, list makes a listing of the contents of the directories and types of files contained therein.

An arbitrary pathname enclosed in "" is also acceptable.

examples:

*list
(lists the contents of your active area)

*list here text data
(lists the contents of your active, text and data areas)

*list sys "/mnt/usr/wolfman/isp/text"
(lists the contents of the system data area and the indicated directory)

Load - load a previously saved file

usage:

*load [data | text | sys | home |] files [data | ...

description:

Load loads the files with the indicated names from the indicated directories. The names are as usual, except in addition home implies the directory from which isp descended into its protected environment. The default directory is data.

UNIX pathnames enclosed in "" are also valid filenames.

examples:

*load sys ex511 ex71 insects
(loads the indicated files from the system data directory)

*load text a.doc b.doc
(loads from the text directory)

*load a b c
(equivalent to "load data a b c")

ISP - New ISP release

USAGE:

(from UNIX)

% isp [-rsp] [file] [args ...]

(from isp)

*isp [-p] file [args ...]

*macro_name [args ...]

DESCRIPTION:

ISP is an interactive statistical package run by a shell-type program that parses and interprets a general language for manipulating objects (e.g arrays) that are at the core of statistical work. These objects are handled within a protected environment designed to facilitate orderly conduct of statistical analysis. The philosophy behind isp is that the structure of computer tools available to the data analyst is a powerful, possibly determinate, influence on his (her) statistical investigation. Thus it tries to initialize a model for data analysis and in fact generalizes the language accepted in McNeil's (Interactive Data Analysis, Wiley, 1977) model.

The features of isp relevant to data analysis fall into three broad groups:

I) The Language. With few exceptions, the structure of an isp command is given by

command [input variables] [{options}] [> results]

e.g.

```
fft x > x.r x.i ;
regress (x,y) > resids
print x (log(x)) ; stemleaf (x ^ .5) {scale = 2}
```

Here "command" is the name of a subcommand in the isp system (see III); "input variables" are, in general, 0 or more arrays or array-valued expressions; "options" are command specific parameters which represent user-controllable factors in the operation of the command; and "results" specify where the 0 or more array-valued results of a command are to be stored.

Hopefully, this language is large enough to accomodate the most commonly-encountered types of operations desired by an analyst; since it has essentially been given a trial run in McNeil's book, it certainly has been shown sufficient for most interactive data analysis needs.

II) The Environment: Isp establishes a protected environment in which interaction is to take place, with three logically separate subenvironments:

a) the active area: where the user is 'placed' upon entering isp; where all variable manipulation and reexpression takes place; where data must be accesible from for statistical commands

to operate on them. The active area literally 'is created' upon entry to isp and 'disappears' upon exit.

b) the data area: where the user saves data from one session to another; the only direct contact with files in this area that a user can have are to either copy data into it, copy data out of it, or delete data from it. Thus, the data area is relatively quiescent.

c) the text area: where the user's nonnumeric information is kept - for example, text documenting a user's own data; text describing a user's own (FORTRAN or C) isp-compatible programs; or text containing a list of isp commands, with symbolic parameters, that the user can invoke as a command rather than type in each time he wishes those commands to be executed as a group (i.e. macros). Access to this area is of the same restricted nature as that to the data area, hence this area is quiescent also.

The data and text areas are permanent, in the sense that they and their contents continue to exist even when the user is not using the isp program, or not logged into UNIX. The active area is transient, and exists only while the isp program is in use. Thus, by saving active objects into the data or text areas, the user can terminate a session and return later to restore exactly his original environment by reloading the saved items.

There are two other areas of importance which the user may access from time to time. One is the system data area, a "library" of data arrays, all taken from McNeil's book, and all illustrating some basic feature of exploratory data analysis. The other is an area the user may create and place his own specialized commands in... this area is his "bin".

III) The Commands: The most important feature of a computer language is its semantics - namely, the actions a given statement of the language may generate. Isp's commands are heavily oriented towards statistical use, which means a) that they have lots and lots of options and parameters to be set and b) that they generally focus on processing a single array (e.g. a two-column array to generate a y vs x plot). The user who finds a need for other commands can easily add his own or structure existing commands into new ones with macros.

USE OF ISP: The simplest way to use isp is to learn by doing. Simply invoke "isp" from UNIX, then begin by typing "explain isp" and go from there. The "explain" subcommand will print out information and examples for almost any command in the system; hence a good idea of the isp's capabilities can be gotten by trying the different subcommands on illustrative data from McNeil's book which the system will provide you. As a consequence, a copy of that book may come in handy.

MACROS

USAGE:

(to create)
*make text macro_name
or
*edit text macro_name

(to invoke)
*macro_name args
or
*isp ['-'] macro_name args

Upon reading a command name, isp looks in the user's text and bin directories for files of the given name. If found there, it executes the named file; otherwise it searches the system directories. If the file is found, but not executable, it is taken to be a macro command and a new copy of the isp shell is spawned to interpret it. The new copy of isp reads in the named file and copies the given arguments into strings 0 through 9; thus the user can examine them using the string mechanism discussed under STRINGS.

strings 0-9 will contain the macro name and any arguments which were passed to the macro. Since arguments might be tagged, or might be options lists, special commands are available for accessing the argument list in commonly-needed ways:

* arg 'tag' pos str

This command takes the input with the given tag, or in the default position pos and places it in string str.

* option pos str

This command takes the pos'th options list and places it in string str.

*local str

This command makes str a unique local name. This is useful when temp files must be created within a macro.

* return tag pos name

This command returns the isp variable name from the macro with the new name given by the name corresponding to tag, or to the pos'th name in the return list. If there is no request for the variable, it is given the name provided by tag.

* creturn tag pos var

Same as return but operates conditionally on the request for the variable var to be output.

EXAMPLE:

*make text myline
```
arg in 1 a
option 1 o
local z
line $a $o > $z
```

```
option 2 p
scat $z $p

creturn z 1 $z
^D
        *myline myjunk {x=2;y=1;} {width=56;} > myresids
```

Here myline is a macro that executes line and also gives a scatterplot of residuals against predicted values. The first argument to myline is the data array, the second is the list of options(possibly '') to line, and the third is the list of options(again possibly '') to scat. The result has tag z and is returned only if asked for. In the example of use above, we would get precisely the same results by typing

```
        *line myjunk {x=2;y=1;} >_myline_z
        *scat _myline_z {width=56;}
        *rename _myline_z myresids
```

NOTE:

A macro returns to the invoker immediately upon a control C or syntax error in the macro, or upon a called routine's signalling error. To prevent such a return, invoking a macro by "isp '-' macro_name" forces the ignoring of such signals and return from the macro when the last command has been interpreted.

Make - make a text file

usage:

*make [text | here |] file

description:

Make reads the keyboard for character input which it places in the text file specified by the arguments - "text" argument implies it will be stored in the user's text area; otherwise it is placed in the active area.

To indicate end of keyboard input to make, type control d, which will echo as ^D.

examples:

```
*make junk
1 23 4 56
78 9 01
^D(control D)
*list
junk            text
```

(here we create a file with the indicated contents)

```
*make text junk
1 2 3 4 5
5  4 6 8
^D
*list text
junk            other
```

MEDPOLISH

USAGE:

```
* medpolish table [> [res @] z [dia @] d]
```

DESCRIPTION:

Medpolish performs a quick exploratory twoway addditive fit to a table. See Tukey's discussion in EDA.

OPTIONS:

x=int array columns to be included in table. Default 1:nc.

EXAMPLE:

```
* load sys illit
*medpolish illit > z d
* code z
* scat d
```

Here illit gives percent illiterate for 9 regions and the first six decades of this century. The coded display and the scatterplot of the diagnostic file show a highly structured, nonlinear pattern.

NSVD

USAGE:

*nsvd var [{options}] [> results]

DESCRIPTION:

Nsvd calculates a singular value decomposition for the matrix var with row weights which are changed iteratively. For a description and justification of the technique used see Princeton Technical Report number ( ) (to be published in the fall of 1977.)

OPTIONS:

rank=int	Specifies rank of fit and thus controls how weights are calculated. Default is nc.
c=float	Bisquare parameter (explain resistant.) Default is 6. To get an unweighted decomposition (all weights equal to one) use c=0.

RESULTS:

[u @]	nr by nc.
[d @]	1 by nc.
[v @]	nc by nc.
[w @]	1 by nr; these four matrices satisfy: a. u diag(d) v' = input variable b. v' v = identity matrix c. u' diag(w) u = identity matrix
[res @]	nr by nc; at each step new weights are calculated according to the row lengths of the residual matrix: u(rank+1) d(rank+1) v(rank+1)' +... + u(nc) d(nc) v(nc)' where u(1),...,u(nc) and v(1),...,v(nc) are the columns of u and v respectively and d(1),..., d(nc) are the elements of d. Res is the final value of the matrix and var - res is the best rank 'rank' fit using the most recent weights.

Every time nsvd is used it involuntarily prints out the first 'rank' singular values as well as the root-mean-square of the remaining nc-'rank' singular values.

Running old isp programs.

USAGE:

*name args ':old-style-options'

DESCRIPTIONS:

To run an old style isp progam, the above format is sufficient. If the command was a system command that has not been yet rewritten, its name will have an _ attached to the beginning. If it is your private command, invoke it by moving it into directory isp/bin, or else enter it in isp/text/cfile as

4:name:filename

where name is what command name you will use and filename is the name of the executable file.

ONEWAY

USAGE:

*oneway table [{options}] [>results]

DESCRIPTION:

Oneway performs a robust oneway analysis of variance based on adaptively weighted least squares.

OPTIONS:

x=int array	columns to include in analysis. Default 1:nc.
c=float	bisquare parameter (explain resistant)
long	prints grand effects and estimated parameters and sigmas.

RESULTS:

[res @]	nr by nc; residuals
[cef @]	nc by 3 ; effects, sigmas, est. d.f.
[wts @]	nr by nc; fitted weights

EXAMPLES:

```
*load sys demopct
*compare demopct
*oneway demopct > res @ r cef @ c wts @ w
*stemleaf r
*scat c
```

Demopct records for each of 24 states the percentage of democratic vote for the Presidential elections of 1960, 64, 68, 72. The compare plot shows clearly the overall pattern. The other two plots should help answer the questions:

i) In landslides is variance between states reduced (e.g. 1964)

ii) Are there any "weird" states in the batch (e.g. unusual election patterns caused by favorite sons or riots or unemployment)

OPTIONS

USAGE:

*command [inputs] [{options}] [>results]

DESCRIPTION:

Options in an isp subcommand are ways for the user to set parameters or control operation of the command. Generally a command has a set of default parameters that it utilizes if no options are specified so that options are in fact optional.

To utilize an option in an isp subcommand, one must first determine what sort of input is expected in order to utilize the option. A program might expect a single real number, an array of integers, a character string, or a pattern to be matched. To find out just what types of parameters the command expects, or to find out the default options if the user fails to specify these, one examines the documentation for the command (e.g. via EXPLAIN).

For example, the stem-and-leaf command, stemleaf, can accept three sets of options. The first set specifies which columns in a variable are to be plotted; the default is all columns. The user overrides this option by typing, say,

*stemleaf myvar {x=1,2,5;}

to get plots only of columns 1,2, and 5. The second set of options specifies how large the plots are to be: normally they will be up to 64 characters wide and blown up by a scale factor of 1. To change this one might request

*stemleaf urvar {scale=3.;width=120;}

to get a plot blown up to 3 times normal detail and using up to 120 characters of width on a line to plot. The third set of options determines whether separate plots are made for each of the columns in a variable, or whether the data are unravelled from the columns into one big batch. Normally a separate plot is made for each column, to change this, try typing

*stemleaf hisvar {unravel}

Of couse these options can be all combined and entered in any order:

*stemleaf anyvar {scale=.5789;x=1 ,2,4;unravel;width=96}

In general, then, to figure out how to specify options, consult the explanation for the command you are considering, and keep in mind the following hints for entering numbers:

Constants are standard as in fortran or basic:

Integers: 1, -32, 108, 011010
Real #'s: 0,1, .012,1.3406e12, -.01, 1e-27,5.67890

Arrays: an array of numbers or of single characters can be typed

...n with elements separated by commas:

```
1,2,3,45,6
-8.0,36e6   ,94.098
a,b,c,d,e,f
```

When the numbers follow a simple pattern, there are two aids in typing

Repeats:	5*object is the same as object,object,...,object
Iteration:	i:j is the same as i,i+1,...,j or i,i-1,...,j depending on whether i>j or not.

For example:

1:6,3*7	1,2,3,4,5,6,7,7,7
a,b,2*c,4*f	1,b,c,c,f,f,f,f
3*-9,2*10	-9,-9,-9,10,10

To answer a question, or match a pattern, one always ends the response with a semicolon, ';'. For example, the pattern 'long' signals to the regress command that it should provide a long printout, and the answer to x= gives the columns to use as independent variables, so

```
regress blat {x=1:3,7:9;long;}
```

is an acceptable reponse, specifying that a regression is to be performed using variable 'blat''s columns 1to 3 and 7 to 9 as independent variables with the lengthy output mode.

EXAMPLES:

```
*scat cats {x=3*1;y=2:4}
*stemleaf yarn {unravel;}
```

FFT, PGRAM

USAGE:

```
* fft   [real @] ri [imag @] ii [{options}] [> results]
* pgram [real @] ri [imag @] ii [{options}] [> results]
```

DESCRIPTION:

"Fft" computes the complex fast Fourier transform of a complex argument. If either part of the input is omitted, it is assumed to be zero. "Pgram" is an alias whose default output is the calculated periodogram.

INPUTS:

[real @]	Real part of the input argument. If omitted, taken to be zero.
[imag @]	Imaginary part of the input argument. Zero if omitted.

*** One of these two must be provided ***

OPTIONS:

taper=float	Series tapering length as a proportion of series length; default is 0.
invert;	specifies that an inverse transform is to be performed.

OUTPUTS:

[real @]	real part of transform.
[imag @]	imaginary part of transform.
[pgram @]	periodogram (= real^2 + imag^2)

PLOT

USAGE:

plot input {options} input {options} ...

DESCRIPTION:

A series of graphs -- one for each input -- is produced. Each graph is controlled by the (optional) options which follow the input variable name and may consist of several plots on the same set of axes. (This distiction between 'graph' and 'plot' is maintained in what follows.)

OPTIONS:

x=int array
specifies the sequence of columns to be used for the x-coordinates
default = 1

y=int array
specifies the sequence of columns to be used for the y-coordinates
default = 2

ch=char array
specifies the sequence of plotting characters to be used
default = *

join=char array
specifies whether each plot will have its points joined or not; valid characters are 'y' and 'n'
default = n

axis=char array
specifies which axes should be drawn; valid characters are 'y' and 'n'
default = y,y,n,n

tick=char array
specifies which axes should have ticks; valid characters are:
'n' = no ticks
'i' = ticks on the inside of the axis
'o' = ticks on the outside of the axis
default = o,o,i,i

label=char array
specifies which axes should have their ticks labelled; valid characters are:
'n' = no labels
'i' = integer labels
'f' = fixed decimal labels
'e' = float labels

'y' = program picks the labels
default: y,y,n,n

nobox
this is an abbreviation for printing no axes, no ticks and no labels

xmin=float
xmax=float
ymin=float
ymax=float
forces the corresponding limit of the graph to be as specified
default is chosen by the program

width=float
height=float
specifies the size of the box in inches; the box is defined to be the size of the graph from axis to axis in each direction
default is device dependent

frame=float array
specifies the horizontal and vertical size of the frame in inches; the frame is defined to be the total size of the plot including the box and any ticks and labels
default is device dependent

offset=float array
specifies a horizontal and vertical offset from the device 'origin' in inches for the graph
default = 0,0

dev=string
specifies the device the plotting is being done on; currently supported devices:
'gsi' = GSI300 typewriter
'tek' = Tektronix 4013 terminal
default = gsi

index
forces the x variable to have the values of one through the total number points for this plot, for each plot in this graph
not default

save
unsave
the save option will cause the current options list to be saved on the file plot.optx where x is the number of the current graph; if plot is invoked again, the options list for the x'th graph will be appended to the saved options for that graph; unsave causes the options file to disappear
not default

keep

specifying keep in an options list will cause the current options to remain in effect for all remaining graphs in the current call to plot
not default

read

plot generates an intermediate graphics language and feeds it to the appropriate driver specified in the 'dev=' option; if read appears in the options list then plot, instead of producing graphs will simply write this graphics language on the file 'inter' (append if 'inter' already exists)

Notes:
1. The number of plots for one graph is determined from the maximum of the sizes of the x, y, ch, and join arrays. If any of these arrays is not as long as the maximum then its last entry is repeated as necessary.
2. For the tick, label and axis options up to four characters in each list may be specified. They are interpreted as referring to the four axes in counter-clockwise order beginning with the bottom x-axis. If less than four entries are specified then the remaining ones have their default values.
3. The intermediate graphics language may also be saved by redirecting the output of plot as in
   'plot var {options} > x'
   In this case, the 'read' option is not required.
4. There are many abbreviations allowed for the option names. Currently each option can be specified with just its first letter (except for xmin, etc.)

PRINT - print a variable or string

USAGE:

*print ['flags'] args ['flags'] args ...

DESCRIPTION:

Print is a simple program for printing isp objects. If an argument is an isp variable name, it prints the variable; if it is a string enclosed in "", it prints the string.

Print rounds the wrong way sometimes, so that 1.000 may be printed as .999.

FLAGS:

'w=int'	line width. default approx 72.
'i=int'	indent flag. i=0 --> don't indent on overflow lines in printing a vector.
'f=string'	format adjustment. string must be of the form n.mc where n and m are integers, n>1 and 0<=m<=9 and c is the character e or f. The former specifies floating point format and the latter fixed. n is the field width for each number to be printed and m is the number of decimals for fixed format and the number of digits for floating. The default is "10.3f".

EXAMPLES:

```
*print "hi mom"

hi mom
*load acid
*print "this is acid" acid

this is acid

1.000   2.345
---
```

Read - read in a file to isp

usage:

*read input > [out @] output

or

*read > [out @] output

or

*read input

description:

Read converts an ascii character file (such as produced by edit or make) into an isp variable. input is the ascii input, which, if ommitted, is assumed to be the keybooard. output is the isp variable to be created, and "out @" is the optional tag. The third form is valid only if input is in another directory (see below) or if it is a text file; in either case, output will have the same name.

Input need not come from the active area (the default), but may reside in another directory. Thus input is of the form

[text | home | here |] file

or

"pathname"

where pathname is a valid UNIX pathname.

options:

valid options are

dims = list, where list is a list of up to three integers; this specifies the shape of the file.

count = #, where # is the largest number of elements to be read from the input file. If count is smaller than the shape of the matrix warrants, the remainder is filled with missing values.

missing = #, where # is the value of the file's missing number.

examples:

```
*read {dims=2,2}
name for new file? twobytwo
1 2 3 4
^D
*print twobytwo

    1.00000      2.00000
    3.00000      4.00000
```

creates a two by two matrix from standard input)

```
*make junk
1 -1 2 -1
^D
```

```
*read junk {missing = -1}
*print junk

     1.00000     ********     2.00000     ********
```

(creates a variable where elements with value -1 are coded missing)

MULTIPLE REGRESSION

USAGE:

*regress matrix [{options}] [>results]

DESCRIPTION:

Regress gives least squares multiple regressions. Matrix contains as columns both independent and dependent variables.

OPTIONS:

x=int array	independent variables. Default 1:nc-1
y=int array	dependent varaiables. For each variable in this list, a regression is run against the independent variables. Default is nc.
long	gives long printout of results with all sorts of goody statistics.

RESULTS:

[res @]	if there is only 1 dependent varaiable, predicted values and residuals go here. nr by 2
[dep.n @]	if n is one of the dependent variables, the predicted values and residuals from regressing it on the independent variables go here.

EXAMPLES:

```
*load sys demopct
*regress demopct {x=1,2;y=3,4} > dep.3 @ z68 dep.4 @ z72
```

Demopct contains for each of 24 states the percentage voting democrat in the 1960, 64, 68, and 72 elections. The above regression attempts to fit the 68 and 72 election results as linear functions of the 60 and 64 results. The predicted values and residuals from the 68 fit go into z68; those from the 72 fit go into z72.

ROBUST REGRESSION

USAGE:

*robust matrix [{options}] [> results]

DESCRIPTION:

Robust performs an iteratively weighted least squares regression that is resistant to thicker tailed distributions of errors. Matrix contains as columns the dependent variable as well as the independent variables.

OPTIONS:

x=int array	independen regression. Default is 1:nc-1
y=int	dependent variable. Default is nc.
c=float	bisquare parameter. Default is 5.
noint	suppress intercept in regression. Not Default.

RESULTS:

[res @]	nr by 3 file of predicted values, residuals, weights.

EXAMPLES:

```
*load sys cars
*robust cars > z
*scat z
```

Cars contains for each of fifty drivers, initial velocity and time to stop from that velocity. Scat should help show traces of nonlinearity in the relation between speed and stopping time.

SAVE - save file(s)

usage:

*save files

or

*save all

or

*save

description:

Save, in usage one, saves the named files in the requested directories ... by default, data. If the default is changed, as in

*save text x y z data w r f

the new default stays in effect until the next change is indicated.

In usage two, save saves each file in data or text depending on whether it is an isp variable or simply a text file.

In usage three, save prompts the user and saves a file if the user responds 'y'.

examples:

*save text crud.text data crud

*save all

*save

```
x? y
y.text? n
y? y
```

SCATTERPLOTS

USAGE:

*scat var [{options}]

DESCRIPTION:

Scat provides y vs x plotting. It allows several separate plots, and up to 8 plots on the same axes.

OPTIONS:

x=int array	X variables for successive plots. Default x=1.
y=int array	Y variable list. To get more than 1 plot on a set of axes, do y=-n,v1,...vn, where n is the number (up to 8) of plots on the same axis, and v1,...,vn are the dependent variables.
width=int	width, in characters, of display
height=int	height, in characters, of display

EXAMPLES:

```
*load sys cars
*scat cars { width=50; height =50}
```

Cars is a bivariate file with 50 observations. The first column contains initial speed; the second, the associated time to stopping for each of the 50 drivers.

```
*load sys pressure
*scat pressure
```

Pressure contains Pressure vs. temperature for some obscure gas. A highly nonlinear relationship.

```
*load sys births
*let births = iota(births[1]),births
*scat births {x=1;y=-3,2:4}
```

Births contains, for three different groups of women, the proportion of women giving birth as a function of time since beginning cohabitation without birth control, measured approximately in months/3. In the above plot, hutterite women's fertility is plotted 'a', taichung 'b', usa 'c'.

Select

USAGE:

```
* select [in @] x [sub @] s [{unravel}] [> [select @] y]
```

DESCRIPTION:

Select takes cases from x for which s is nonmissing and nonzero. If the array x is 1 by n it treats the array as n by 1. If the unravel option is exercised it treats the array as k*n*p by 1.

EXAMPLE:

```
* select x (x[*,1] > 0) > xpos
```

this selects the rows from x where the first element of the row is positive.

```
* select x (x[2] != ()) > xnom
```

this selects the rows from x where the second column is nonmissing.

SEMANTICS

USAGE:

*command inputs [>outputs]

DESCRIPTION:

The semantics of the above command are as follows:

1. Recognizing the Command Name.
   a. If command is a string enclosed in "" (e.g. "/bin/prog") it is taken to be an explicit UNIX pathname and to return no meaningful status codes.
   b. If no pathname is generated during a. then the user's text and bin, followed by the system's text and bin areas are searched for a file with the same name as command If the file is found in a text area, the file is taken to be a macro; otherwise it is taken to be an executable file.

2. Handling Inputs.
   If any input arguments that are let expressions enclosed in parentheses, they are converted to names of temporaries, and the expressions are passed to let for evaluation as temporaries. The command is executed with the names of temps in place of the expressions; the temps are deleted after return from the command. For example:

```
*print x (2*x) (log(x))
```

is executed by isp as if it were the sequence

```
*let _01_=2*x; _02_=log(x);
*print x _01_ _02_
*delete _01_ _02_
```

3. Handling Outputs.
   If outputs are requested, the object command will recognize them as follows:
   (a) first on tag matching. Output names will be determined according to where tags match.

```
* fft x > real @ r imag @ i
```

Here the "real" part of the transformed series is put in "r", the "imaginary" part is put in "i".

(b) Second on order. If step (a) fails, and if there is no tagged result name in the default position for this output in the result list, then the (untagged) name in that position is given to the result.

```
* fft x > r i
```

Here the real part of the transform is placed in "r", the imaginary part in "i", because the default ordering of the result list for the "fft" command is ("real","imag").

(c) Third on forcing. If steps (a) and (b) fail, and the command requires that the result be created, whether asked for in the results list or not, then the result is named according to its tag.

```
* fft x
```

Generates two outputs: "real" and "imag", since "fft" forces their creation, and the user doesn't specify alternate names for them.

If the final elements of a command line are '>>' followed by a UNIX pathname in '', the standard output of the named command will be routed to the named file. For example,

```
* print x >> '/dev/tty8'
```

prints the contents of x on tty8.

EXECUTION:

If the command was found to refer to a text file, a new copy of the isp shell is spawned with, as arguments, the command name followed by the command arguments. The new copy of the isp reads its input from the file referred to and copies the arguments into strings 0 through 9. Thus macros can reference their arguments; to create a macro one places in his text area a file with the desired macro name and with the desired commands as lines of the file.

Otherwise the command is exec'd. If the command is not 'shell', then signals are reset, so a control C will kill the command during its execution. If the command is 'shell' or 'unix' the child does a chdir("../..") before exec'ing.

CLEANUP:

If the exec'd command returns nonzero status, the remainder of the command line is aborted. In a macro invoked normally, the remainder of the macro is aborted. The ISP shell prints no message at all if the command died due to a user error. If the system part of the exit status is nonzero and doesn't indicate control C, "fatal error in ..." is printed.

ISP SYSTEM DOCUMENTATION

*** these are all documented in explain files ***

GENERAL INTEREST

isp	explains philosophy of system. lengthy
changes	explains the changes from the old isp
syntax	syntax of isp commands
semantics	semantics of isp commands
macros	how to make & use macros
strings	how to make & use strings
results	how to use the results feature
options	how to specify options to a command
document	how to understand the documentation
oldisp	how to run a command written for the old isp

SYSTEM COMMANDS

make	create a text file
edit	edit a text file
read	read a text file, converting to isp variable
save	save an isp variable or text file for future use
load	load a previously saved variable or file
delete	delete active variable(s).
unsave	unsave saved objects
list	list contents of active, data, text, or system areas
print	print variable or string
let	algebraic and manipulative capability
explain	how to explain something
echo	echo command arguments
select	logical subscripting

DATA ANALYSIS

boxplot	boxplots
stemleaf	stemleafs
code	coded displays
fivenum	fivenums
condense	Med. & MAD
biweight	biweights
compare	compares
scat	scatterplots
line	line fits
medpolish	table polishes
six	six line plots

ROBUST FITTING

robust	regression
oneway	anova
twoway	anova

LEAST SQUARES

stat	statistics

corr	correlations
regress	regressions
eigen	eigenvalues real symmetric matrix

TIME SERIES

smooth	Tukey's smoothers
fft	Fast Fourier Transform
pgram	periodogram
xpgram	cross periodogram
cohphs	coherence and phase

UTILITIES

gplot	gsi,tek graphics
trn	matrix transposes
sort	sorting

SIX-LINE-PLOTS

USAGE:

*six var [{options}]

DESCRIPTION:

Six plots Tukey's six line plot of a time series. Numbers are plotted in standard deviations from the median in base 4. Var is either a matrix whose columns are to be plotted or a row vector.

OPTIONS:

x=int array	specifies columns to be plotted. Default is all.
unravel	specifies that multi-variable file should be treated as single variable. Not default.
width=int	plot width; default = 64.

EXAMPLES:

```
*load sys sunspots
*six sunspots
*smooth sunspots > sspts
*six sspts
```

Sunspots contains several years of the monthly-recorded Vienna sunspot numbers. The second plot should show a smoother structure.

SMOOTH

USAGE:

*smooth var [{options}] [>results]

DESCRIPTIONS:

Smooth performs most of Tukey's resistant smooths. Var is either a matrix of columns to be smoothed or a row vector.

OPTIONS:

by:char string — Specifies desired smooth. It is read left to right and interpreted as seen. Valid chars:

3,5,7,9	- running medians of 3,5,7 or 9.
2,4,6,8	- running medians followed by a pass of 2.
r	- repeat preceding op. till no change. (valid only if preceded by odd digit).
s	- splitting mesas followed by another 3r.
h	- hanning (same as 22)
t	- twicing; xt == x+x
*2	- " "
+	- reroughing; a+b means do a on data, then b on rough then add back.

trace — Traces smoothing operation by printing current smooth being executed. E.g. 3rssht may generate a trace of '3333s33s333ht333s333s33ht'

x=int array — columns to be smoothed. Default is 1:nc.

RESULTS:

[smooth @] — Copy of original matrix with designated columns smoothed.

EXAMPLES:

```
*load sys sunspots
*smooth sunspots {by:4253h} > x
*six sunspots; six x;
```

Sunspots contains 15 years of Vienna sunspot numbers. The second six line plot should reveal a smoothed structure.

SORT

USAGE:

* sort [in @] d [{options}] [> [sort @] s]

DESCRIPTION:

Sort sorts the data in array d. If d is not a row, it sorts the columns of d; otherwise it sorts the elements of d.

OPTIONS:

x=int array	Columns to be sorted into order. x=+n indicates ascending order; x=-n indicates descending order. default: x=1:nc.
y=int array	Columns to be permuted according to the sorting permutation of the x= column.
unravel	Treat the array as a single long row vector

STAT, CORR

USAGE:

```
*stat var [{options}] [>results]
*corr var [{options}] [>results]
```

DESCRIPTION:

Stat gives means, standard deviations, extreme values, and intervariable correlations, if desired.

OPTIONS:

x=int array	columns to be stated. Default: all.
unravel	view multivariable set as 1 variable.
quiet	print no evil
long	force printing of correlation matrix if commad was invoked as stat

RESULTS:

[stat ∂]	nc by 5; means,sdev,min,max,sum sq dev.
[corr ∂]	nc by nc; correlation matrix

EXAMPLE:

```
*load demopct
*stat demopct {long}
```

Demopct is the percent democratic in the presidential elections of the Imperial era 1960-1972, for 24 industrial states. Columns are elections; rows are states.

```
*load sys sunspots
*let
* s1 = (),sunspots[1:179]
* s2 = (),s1[1:179]
* s3 = (),s2[1:179]
* s = trn(sunspots),trn(s1),trn(s2),trn(s3)
^D
*scat s {x=3*1;y=2:4}
*corr s
```

Here sunspots is a variable containing the Vienna sunspot numbers; the correlation matrix produced will give the correlations between this months sunspot count and that of each of the previous three months in row one. The scatterplots display visually the information presented in the lag 1, 2, and 3 correlations.

STEM-AND-LEAF PLOTS

USAGE:

*stemleaf var [{options}]

DESCRIPTION:

Stemleaf gives Tukey's stem and leaf display, a jazzed up histogram from which can be read off the first three digits of the data.

OPTIONS:

x=int array	Specifies which columns are to be plotted. Default list is 1:nc
unravel	specifies that a multi-column file be treated as a single variable. Not default
scale=float	specifies fineness of breakdown in display. can be increased if more 'bins' are desired. Default is 1.0.
width=int	specifies page width - # chars per line. Default is 64.

EXAMPLES:

```
*load sys islands
*stemleaf (log(islands))
```

Islands contains the sizes of the world's largest islands. McNeil claims this plot demonstrates conclusively that Australia is a continent, not an island.

```
*load sys sunspots
*stemleaf sunspots
*stemleaf (sunspots ^ .5)
```

Sunspots contains a 15 year stretch of the monthly Zurich sunspot counts. Taking square roots gives a more symmetric distribution about the median. This is in line with the idea that counts data should be rooted before use.

STRINGS

USAGE:

```
        (to create)
        define name string

        (to use)
        $name
```

DESCRIPTION:

The string capability allows one to avoid repetitive typing, or to reference arguments to macros. Strings are named 0-9 a-z. For example, if one were using a plot program often and typed

```
        *plot x {width=60;height=40}
.
.
.
        *plot xyz {width=60;height=40}
```

(s)he could instead type

```
        *define o {width=60;height=40}
        *plot x $o
.
.
.
        *plot xyz $o
```

Certainly the user can figure out more ingenious ways to use strings.

Strings are expanded out even within quotes (') (") (()) ({}). To explicitly type the character $, one types $$ instead; since this defines no legal string, the first $ is discarded and the second remains.

EXAMPLES:

```
        *define e 'echo crud'
        *$e;$e;$e;
crud
crud
crud
        *define f '$e;$e;$e'
        *$f
crud
crud
crud
        *echo '$f'
echo crud;echo crud;echo crud

        *define w 'width=60;'
        *define h 'height=50;'
        *scat z {$w $h}
```

SYNTAX

DESCRIPTION:

The syntax of an isp command, with two exceptions, is

command inputs [> results]

where command is a <name>, inputs is a list of <name>s and <option list>s and results is a list of <name>s or <tagged names>. In other words,

command --> <name> <inputs> '>' <results>
OR <name> <inputs>

inputs --> (<name> OR <options list>)*

results --> (<name> OR <name> '@' <name>)*

name --> any string composed of alphanumerics and '.' and '_' only.

option list --> { anything }

Finally, (junk)* = 0 or more copies of junk.

EXAMPLES:

valid in isp

```
*abracadabra '!"#$%$&%%%%%%%%())))*+*''''>><<>>>'
*abcdef019_.345 {hi there bozos} > abcdefg @ x z @ a5_67.0
*x y z w e f g h > h0 h1 h2 h3 h4
*print "hi quy"
*a > x y z w e f @ z
```

not valid

```
*&&%$ #$##%%%
* >
*a$ < AA+
*abcdefg 'hi there
*abc qo jump"
*abc defg hij @ klm
```

EXCEPTIONS:

```
*let <anything let will accept>
*unix <anything unix will accept>
```

e.g.

```
*let x = trn(y#*x + log(z)/(u^13.6)) ** trn(,x)
*unix who;du;df;mail dave
```

TWOWAY

USAGE:

*twoway table [{options}] [>results]

DESCRIPTION:

Twoway performs a robust twoway analysis of variance based on the iterative least squares algorithm (explain resistant)

OPTIONS:

x=int array	columns to be included in analysis. Default 1:nc
c=float	value of bisquare parameter. Default 6.
long	print out grand totals, fitted effects, sigmas, degrees of freedom.

RESULTS:

[res @]	nr by nc; residuals
[dia @]	(nr*nc) by 2; diagnostic data.
[ref @]	nr by 3; fitted row effects
[cef @]	nc by 3; fitted column effects
[wts @]	nr by nc; computed weights.

EXAMPLES:

```
*load sys illit
*compare illit
*twoway illit > dia @ d
*scat d
*let liilt = log(illit)
*compare liilt
*twoway liilt > dia @ dd
```

Illit contains the illiteracy rates over the last six decades for 9 different regions of the country. The comparison plot shows that there is a lot of interregional variation in illiteracy near the turn of the century, but that it drops over time until all regions are almost equally literate. The diagnostic plot generated from twoway shows a distinct linear trend which suggests logarithmic reexpression, yielding the second time, a much better fit to the data.

Unsave - remove save'd files

usage:

*unsave [text | data |] files [text | ...

description:

Unsave removes previously saved files from either the text or data directories.

examples:

```
*list text
a                other
b                other
cex              other

*unsave text a b
*list text
cex              other
```

(i.e. a and b were deleted from the textt directory)

```
*list data
mat              array(3,2)
crud             array(10,20)
*unsave crud
*list data
mat              array(3,2)
```

(i.e. crud is no longer saved in data)

Whit

Usage:

whit x {options} > result

Description:

Whit smooths the columns in an array be minimizing a linear combination of a robust measure of resual width and the l2 norm^2 of the second difference of the smoothed sequence.

Options:

x=int array	columns to be smoothed
alf =float	smoothing parameter. smoother as alf increases. default: a rough .1
eps=float	convergence tolerance. default a fine 1% improvement.

XPGRAM

USAGE:

```
* fft series > fr fi
* xpgram fr fi > xpgram
```

DESCRIPTION:

"Xpgram" takes the real and imaginary parts of the transform of the n by p multiple time series 'series', and outputs the n by p^2 matrix where each row contains the lower triangle of the (hermitian) spectrum matrix at the given frequency, stored pairwise: real(1,1), imag(1,1), real(2,1),imag(2,1), etc. It is customary to smooth the columns of this matrix, and then use it as an argument to "coqphs."