

SOFTWARE RESTYLING IN GRAPHICS
AND PROGRAMMING LANGUAGES

by

Eric Grosse

STAN-CS-78-663
SEPTEMBER 1978

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



SOFTWARE RESTYLING IN GRAPHICS AND PROGRAMMING LANGUAGES

by

Eric Grosse^{*}

^{*}Department of Computer Science, Stanford University, Stanford, CA 94305

This paper was presented at the 1978 Army Numerical Analysis and Computers Conference in Huntsville, Alabama, March 1, 1978.

Research supported in part under Army Research Grant DAHCO4-75-G-0195 and in part under National Science Foundation Grant MCS75-13497-A01.



SOFTWARE RESTYLING IN GRAPHICS AND PROGRAMMING LANGUAGES

Eric Grosse
Computer Science Department
Stanford University
Stanford CA 94305

ABSTRACT. The value of large software products can be cheaply increased by adding restyled interfaces **that** attract **new** users. As **examples** of this approach, a set of graphics primitives and a **language precompiler** for scientific computation are described. These two **systems** include a general user-defined coordinate system instead of numerous system settings, **indention** to specify **block** structure, a modified indexing convention for array **parameters**, a syntax for n-and-a-half-times-*round loops, and engineering format for real constants: **most** of all, they strive to be as small as possible,

9.3 PHILOSOPHY. Kernighan and Plauger [1976] describe explicitly and by example three precepts of the Software Tools philosophy:

- trim out **the** inessentials
- build it adaptively
- let someone else do the hard part

Two more examples, driven by **the** same philosophy, are given **below**. The **basic** idea is to obtain high leverage by taking an existing, powerful piece of software and make it useful to more people by designing a **new** interface. Webster's calls this process facelift ing: "a restyling intended to increase comfort or sa **lability**."

1.0 JUSTIFICATION FOR STILL ANOTHER PROGRAMMING LANGUAGE.

Fortran will no doubt remain for many years the most important programing language for scientific computation. **When** used carefully and with discipline, it yields remarkably portable codes; this is its greatest virtue. But, as **programmers** have complained **for** years, it also has many faults:

- awkward syntax for statements, strings, names
- primitive control structures
- DO loop restrictions
- no macros

Fortran preprocessors, such as **MORTRAN** [Cock+Shustek 1975], have eliminated many **of these** disadvantages and therefore have become very popular. Unfortunately, **they** reduce portability somewhat, **since** either the preprocessor **must** be installed at the **new** site

or illegible 'object' Fortran sent there. More importantly, such preprocessors have only a minor effect on inherent problems of Fortran:

- dynamic allocation is either **unavailable** or requires the use of rather confusing tricks
- no **PROCEDURE VARIABLE** type
- no **STRUCTURE type**
(Labelled common blocks, since they do not use the **combinatorial** possibilities of procedure **parameterization**, are less flexible.)
- no 0-origin indexing
- array bound **information** is not automatically passed
- no vector operations
- no recursion

The PCRT library makes dynamic allocation one of its **most** advertised features: "We have **found** that use of **dynamic** storage allocation in **PORT** leads to more clearly structured programs, cleaner calling sequences, improved **memory** utilization, and better error **detection**." [Fox+Hall+Schryer 1977] Adding a stack to Fortran is a messy affair, however, as shown in figure I, which contains two alternate methods in **PCRT** for allocating an

| | |
|---|------------------------------------|
| SUBROUTINE LBB(A,N) | SUBROUTINE LBB(A,N) |
| COMMON /CSTAK/DSTAK(500) | COMMON /CSTAK/DSTAK(500) |
| DOUBLE PRECISION DSTAK | DOUBLE PRECISION DSTAK |
| INTEGER ISTAK(1000) | INTEGER ISTACK(1000) |
| REAL A(1) | REAL A(1) |
| REAL RSTAK(1000) | REAL RSTAK(1000) |
| EQUIVALENCE (DSTAK(1),ISTAK(1)) | EQUIVALENCE (DSTAK(1),ISTAK(1)) |
| EQUIVALENCE (DSTAK(1),RSTAK(1)) | EQUIVALENCE (DSTAK(1),RSTAK(1)) |
| II = ISTKGT(2*N,2) | II = ISTKGT(2*N,2) |
| IR = ISTKGT(N,3) | IR = ISTKGT(N,3) |
| { code referring to RSTAK(IR+n) and ISTAK(II+m) probably ending with code to store the stuff from the real scratch storage into array A } | CALL L1BB(A,ISTAK(II),RSTAK(IR),N) |
| . | CALL ISTKRL(2) |
| CALL ISTKRL(2) | RETURN |
| RETURN | END |

figure 1

INTEGER and REAL array.

Other proposals are even more complicated. (After a 7 page description of **DYNOSCR**, **Huybrechts**[1977] states: "This paper gives **only** the basic features of the **DYNOSOR** system. A **more** sophisticated use allows the user, once he is familiarized **with** the system, to improve greatly the speed of programs using **it.**")

PL/I, which is now becoming fairly widely available in some **form**, **overcomes** all these difficulties. However, so huge a **language** tends to overwhelm people, and because of tricky precision rules, **silent** type conversions (as in **I=J=0:1**), and the like, learning only part of the language is dangerous.

Other languages, while beautifully designed, have their own **flaws**. For example, **Algol W** does not have a robust interface to **Fcrtran**; in addition to this [**Mohilner** 1977], **Pascal** places painful restrictions on arrays.

1.1 T. Thus another **approach seems** warranted, which can combine the needed features of **PL/I**, the deliberate syntax of **ALGOL**, and the low implementation cost of the **Fortran** preprocessors. Such an approach has produced the language **T**, intended to assist in the **implementation** and **documentation** of algorithms for scientific computation. The principal **aims** have been ease of reading and writing, low implementation cost, and reasonable efficiency.

Appendix T gives the **formal** language proposal, specifying the syntax **according** to **Wirth's** proposal [1977]. Since **T** is similar to **Fortran**, **Algol 60**, and **PL/I**, a **complete** specification of the semantics may be omitted without confusion. To provide the heuristics behind the design choices and to give an overview of the language, various aspects of the following example **will** be discussed.

TRIP EAR

- * example of **T** and **G** systems:
- * various views of the sum of three Gaussian peaks:
- * Eric Grosse Stanford University

```
REAL: AZIM, ELEV,      # VIEWING ANGLES FOR SURFACE PLOT
      RELEFR, ABSEFR, # ERROR TOLERANCES FOR ODE
      T, TOUT,          # INDEPENDENT VARIABLES OF TRAJECTORY
      NORMYP,           # 2 NORM OF THE GRADIENT
REAL(2): LL, UR,        # CORNERS OF RECTANGULAR DOMAIN OF FUNCTION
      ORIGIN,           # FOCAL POINT FOR SURFACE PLOT
      X0, SCALE,        # COORDINATE TRANSFORMATION PARAMETERS
      Y, YP              # LOCATION AND GRADIENT FOR TRAJECTORY
REAL(142): ODEWORK
INTEGER(5): ODEIWORK
```

```

DEFINE(P,20)          * density of P samples;
REAL(-P:P,-P:P: I?): F TABLE
REAL(3): LEVEL        * CONTOUR LEVELS
INTEGER: I, J,
IFLAG                  * DIAGNOSTICS FLAG FOR ODE
STRUCTURE: PAPA?"!      * LOCATIONS, HEIGHTS. AND WIDTHS OF PEAKS
REAL(3,2): X
REAL(3): H, W
STRUCTURE: PF          * PLOT PILE
INTEGER(500): WORK
PROCEDURE: GOOPEN, GCLOSE, GPICT, GCONT, GSURF, GLTYPE,
GJUMP, GDRAW, GTRAN1
FORTRAN PROCEDURE: CDE, DF, STASH
PROCEDURE () REAL: F

* SET UP PARAMETERS
BLANK SEPARATION (2)
REAL DIGITS(3)
GET DATA (AZIM,ELEV)
PUT DATA (AZIM,ELEV)
X(1,1) := 0
X(1,2) := 0.5
X(2,1) := -0.43'301 2702
X(2,2) := -0.25
X(3,1) := -X(2,1)
X(3,2) := X(2,2)
PUT DATA ARRAY (X)
GET ARRAY (H)
PUT DATA ARRAY (H)
GET ARRAY (W)
PUT DATA ARRAY (U)
STASH(X,H,W)
FOR( -P <= I <= P )
  Y(1) := FLOAT(I) / P
  FOR( -P <= J <= P )
    Y(2) := FLOAT(J) / P
    F TABLE(I,J) := F(Y,PARAM)

* SURFACE PLCT
GOOPEN('VEP12FF',PF)
GPICT(PF)
LL := -1
UR := 1
ORIGIN := 0.5
GSURF(LL,UR,FTABLE,AZIM,ELEV,ORIGIN,0.25,PF)

```

```

# CONTOUR PLOT
GPICT(PF)
SCALE := 0.3333
X0 := -0.5/SCALE(1)
GTRAN1(X0,SCALE,PF)
GET AFRAY (LEVEL)
PUT DATA ARRAY(LEVEL)
GCNT(L,UR,FTABLE,LEVEL,PF)
GLTYPE('DOT',PF)
GET AFRAY(LEVEL)
PUT DATA ARRAY (LEVEL)
GCNT(L,UR,FTABLE,LEVEL,PF)

# COMPUTE AND PLOT TRAJECTORY
RELERR := 10(-6)
GLTYPE('SOLID',PF)
ABSERR := 10(-6)
WHILE( ~ END OF INPUT )
    GET ARRAY( Y )
    PUT DATA ARRAY( Y )
    T := 0
    GJUMP(Y,PF)
    IFLAG := 1
    WHILE( NORMYP > 1(-3) & 1<=IFLAG & IFLAG<=3 )
        TOUT := T + 10(-3)/NORMYP
        CDE(DF,2,Y,T,TOUT,RELERR,ABSERR,IFLAG,ODEWORK,ODEIWORK)
        CASE
            2 = IFLAG
            GDFAW(Y,PF)
            3 = IFLAG
            PUT('ODE DECIDED ERROR TOLERANCES WERE TOO SHALL.')
            PUT('NEW VALUES: ')
            PUT DATA( RELERR,ABSERR )
        ELSE
            PUT('ODE RETURNED THE ERROR FLAG: ')
            PUT DATA(IFLAG)
        FIRST
        DF(T,Y,YP)
        NORMYP := NORM2(YP)
    GCLOSE(PF)

F( Y, PARAM ) Z
REAL(): Y
REAL: Z, NORMSQ
STRUCTURE: PARAM
    REAL(3,2): X
    REAL(3): H, W
INTEGER: I
Z := 0
FOR( 1 <= I <= 3 )
    NORMSQ := (Y(1)-X(I,1))**2 + (Y(2)-X(I,2))**2
    Z := Z + H(I)*EXP(-0.5*W(I)*NORMSQ)

```

1.2 CONTROL AND OTHER SYNTAX, Perhaps the **most** striking feature the **Algol** veteran sees in this example is the complete absence of **BEGINs** and **ENDs**. Not only is the text indented, but the indentation actually specifies the block structure of the program. Such a **schema** was apparently first proposed by **Landin [1966]**. Except for an endorsement by **Knuth [1974]**, the idea seems to have been largely ignored.,

Ideally, the text editor **would recognize** tree-structured **programs** [**Hansen 1971**]. In practice, text editors tend to be line oriented so that **moving** lines about in an indented program requires cumbersome manipulation of leading blanks. Therefore the current implementation of T uses **BEGIN** and **END** lines, **translating** to indentation on output. Thus the input

STRUCTURE: **PARAM**

```
((  
  REAL(3,2): X  
  REAL(3): H, W  
)
```

produces the output

STRUCTURE: **PARAM**

```
  REAL(3,2): X  
  REAL(3): H, W
```

Whatever the implementation, the key idea is to force the **block** structure and the indentation to be automatically the same, and to reduce clutter from redundant keywords.

Blanks are insignificant outside of strings. **Mathematical** tables have long used blanks inside numeric constants, as in

PI := 3. 14159 26535 89793

for readability. Blanks in identifiers also can improve readability, while reducing the chance of misspelling and easing the pain of name length restrictions imposed by the local operating **system**.

In accordance with the recommendations of **Scowen+Wichmann [1973]**, comments start with a special character, **#**, and run to the end of the physical line.

The small reserved **word** list eliminates the need for a stroking convention. The psychological advantages of this approach have been elaborated by **Hansen [1973]**.

The form of the assignment and procedure call statements follows the clean, clear style of **Algol 6C**. To make macros more understandable, their syntax and **semantics** match those of procedures as closely as possible.

In addition to normal statement sequencing and procedure calls, three control structures are provided. The CASE and WHILE statements are illustrated in this typical program segment:

```

WHILE( NORMYP > 1(-3) & 1<=IFLAG & IFLAG<=3 )
TOUT := T + 10 (-3)/NORMYP
ODE(DF, 2, Y, T, TOUT, RELEERR, ABSERR, IFLAG, ODEWORK, ODEIWORK)
CASE
 2 = IFLAG
  GDRAW (Y, PP)
 3 = IFLAG
    PUT('ODE DECIDED ERROR TOLERANCES WERE TOO SMALL.')
    PUT ('NEW VALUES:')
    PUT DATA (RELEERR, ABSERR)
  ELSE
    PUT('ODE RETURNED THE ERROR FLAG:')
    PUT DATA (IFLAG)
  FIRST
  DF (T, Y, YP)
  NORMYP := NORM2(YP)

```

The CASE statement is **modelled** after the conditional expression of LISP: the boolean expressions are evaluated in sequence until one evaluates to YES, or until ELSE is encountered. The use of indentation makes it easy to visually find the relevant boolean expression and the end of the statement.

One unusual feature of the **WHILE** loops is the optional FIRST marker, which specifies where the loop is to be entered. In the example above, the norm of the gradient, **NORMYP**, is computed before the **loop** test is evaluated. Thus the loop condition, which often provides a valuable **hint about** the loop invariant, appears prominently at the top of the loop, and yet the common n-and-a-half-times-* **round** loop **can** still be easily expressed.

The **FOR** statement adheres as closely as practical to common mathematical practice.

```

FOR( 1 <= I <= 3 )
  NORMSQ := (Y(1)-X(I,1))**2 + (Y(2)-X(I,2))**2
  Z := Z + H(I) *EXP(-0.5*W(I) *NORMSQ)

```

Several years **experience** with these control constructs has demonstrated them to be adequately efficient and **much** easier to maintain than the alternatives.

Procedure nesting is not used for two reasons. First, textual nesting that extends over many pages is difficult for a human to keep track of. Second, programs typically contain several high level2 procedures calling a **single** primitive, so a tree representation is inappropriate anyway.

By removing the nesting of procedures, however, we **worsen** the **problem** of entry point hiding that arises when **combining programs** from many sources into a **single** library. A solution to this **problem** is to have an official **name** for each procedure, coded along the lines of **IMSL**, and also a more mnemonic nick name (which users **can** pick for themselves if they like). The **macro**

processor which is **built** into T can then be used to **change** all **occurrences** of the nick names into the **corresponding** official **names**.

1.3 DECLARATIONS. The fundamental scalar types **are INTEGER**, **REAL**, and **COMPLEX**, from which arrays and structures may be built up. As the example

```
REAL (-P:P,-P:P)
```

illustrates, general upper and lower bounds are **allowed**.

The upper bound expression is omitted **for** a formal array parameter, so that an appropriate value **can** be taken from the length of **the** corresponding actual array argument. The **origin** of an actual array argument need not match the origin of the **corresponding** formal **array parameter**. For example, if the actual **argument A** was declared **REAL(0:7): A** and the formal parameter **B** was declared **REAL(): B**, then **B(8)** will correspond **to** **A(7)**. Host languages, when they allow lower bounds at all, do not **permit** this flexibility, which is used in the **example program** when a matrix with lower **bound -P** is passed to a general purpose library routine **which** assumes a lower bound of 0.

Structures of arbitrary depth **may be** declared. As the examples

```
STRUCTURE: PARAM
```

```
  REAL(3,2): X
```

```
  REAL(3): H, W
```

```
STRUCTURE: PF
```

```
  INTEGER(500): WORK
```

suggest, structures are useful passing collections of related **data**, without the need for long parameter lists. This **makes** feasible the prohibition of global variables in a drastic attempt to **narrow** and **make more** explicit the interface **between** procedures. Euclid [Popek+others 1977] has emphasized the importance of visibility of names.

The graphics **procedures** which use the **WORK** vector of the example are **able to** divide up the space into convenient units. This **capability**, which would **be** possible in **PL/I** only through the use of pointers, encourages information hiding and abstraction.

PROCEDURE **VARIABLES** allow the names of procedures to **be saved**, an essential feature for applications like the user-specified coordinate transformation described in the graphics system below.

The importance of **existing Fortran** software is recognized **by** **providing** for FORTRAN **PROCEDURES** as an integral part of the language. The current **implementation** of **I** performs this linkage in a more efficient way than the naive user of **PL/I** **would** be **likely** to discover,

A novel syntax **is** introduced for function returns. Since procedures may be recursive, **Fortran's** convention of using the function **name** as variable cannot be followed. Instead, the procedure header declares a return variable just like any other parameter:

```
F ( Y, PARAM ) Z
  REAL () : Y
  REAL: Z
  ...
  ...
```

1.4 **INPUT/OUTPUT.** Beginners often find **Fortran's** input/output the most difficult part of the language, and even seasoned programmers are tempted to just print **unlabelled** numbers, often to more digits than **just if** fed by the problem, because formatting is so tedious. **PL/I's** list and data directed **I/O** is so much easier to use that it was **wholeheartedly** adopted in T. By providing procedures for **modifying** the number of decimal places and the number of separating blanks to be output, no **edit-directed** I/O is needed. Special statements are **provided** for array **I/O** so that, unlike **PL/I**, arrays can be printed **in** orderly fashion without explicit **formatting**.

Since **almost** as much time is spent in scientific computation staring at pages of **numbers** as at pages of program text, much thought was **given** to the best format for **displaying** numbers.

In accordance with the "engineering format" used on **Hewlett-Packard** calculators and with standard metric practice [GM Service Section 1977], exponents are forced to be multiples of 3. As figure 2, an excerpt from the **example** program's output, shows, this convention has a histogramming effect that concentrates the **information** in the leading digit, as opposed to splitting it between the leading digit and the exponent, which are often separated by 14 columns. The use of parentheses to surround the exponent, like the legality of **imbedded** blanks, was suggested by mathematical tables. This notation separates the exponent from the mantissa more distinctly than the usual **E** format.

1.5 DISCUSSION.

Following **Kernighan+Plauger [1976]**, the initial implementation is unsophisticated [Comer 1978]. **Nevertheless**, the preprocessing is less costly than the **PL/I** compiler, so the **overall** results are quite satisfactory. (The evaluation **looks** even better if one compares **PL/I + T** against **PL/I + PL/I's** macro preprocessor.) Most of the processor cost lies in basic **I/O**: by integrating the macro processor with the language translator, this cost has been minimized. [Kantorowitz 1976] Much of the **two-man-months** spent in implementation were spent in understanding nooks and crannies of **PL/I**.

| | |
|---------------------|--------------|
| 53.5106 (-03) | 5.35106E-02 |
| 51.3109 (-03) | 5.13109E-02 |
| 46.7211 (-03) | 4.67211E-02 |
| 40.6514 (-03) | 4.06514E-02 |
| 33.7636 (-03) | 3.37636E-02 |
| 26.4908 (-03) | 2.64908E-02 |
| 18.9800 (-03) | 1.89800E-02 |
| 11.3401 (-03) | 1.13461E-02 |
| 3.63500 (-03) | 3.63508E-03 |
| - 4.12944 (-03) | -4.12944E-03 |
| - 11.9123 (-03) | -1.19123E-02 |
| - 19.7092 (-03) | -1.97092E-02 |
| - 27.5248 (-03) | -2.75248E-02 |
| - 35.3243 (-03) | -3.53243E-02 |
| - 43.1176 (-03) | -4.31176E-02 |
| - 50.9068 (-03) | -5.09068E-02 |
| - 58.6841 (-03) | -5.86641E-02 |
| - 66.4483 i - 0 3) | -6.64483E-02 |
| - 74.1973 (-03) | -7.41973E-02 |
| - 81.9297 (-03) | -8.19297E-02 |
| - 89.6443 (-03) | -8.96443E-02 |
| - 97.3401 (-03) | -9.73401E-02 |
| -105.010 (-03) | -1.05010E-01 |
| -112.670 (-03) | -1.12670E-01 |
| -120.302 (-03) | -1.20302E-01 |
| -127.910 (-03) | -1.27910E-01 |
| -135.493 (-03) | -1.35493E-01 |
| -143.050 (-03) | -1.43050E-01 |

figure 2

T is not intended to replace any existing languages. For distributing mathematical software, Fortran remains the only practical medium; for character processing, something like PL/I or SNOBOL should be used. Still, for the bulk of scientific computation, T ought to be the easiest to use, particularly since it coexists comfortably with Fortran and PL/I. On the other hand, one can imagine ways that T might be improved, as well. Features omitted for ease of implementation include:

- trimmed arrays, like `X(2:N)`
- procedure results of general type
- conditional boolean operators that do not evaluate their arguments when it is possible to avoid doing so
- a swap operator

For other features, no entirely satisfying design was apparent:

- strings
- more general procedure calls (such as indefinite number and type of arguments)
- a means of constructing arrays directly from components, as

a string constant constructs a string from individual characters - a means of specifying the invocation graph of who calls whom

Perhaps the most fundamental thou unavoidable flaw is that, unlike LISP, the language is not trivial, and therefore programs cannot be trivially manipulated.

2.0 JUSTIFICATION FOF STILL ANOTHER SET CF GRAPHICS PRIMITIVES. The next example of restyling is a simple but reasonably complete interface for noninteractive device-independent graphics. In addition to the basic line drawing primitives, higher level procedures are provided for displaying functions of one or two variables. This interface has been implemented as a library of PL/I procedures which call the SLAC Unified Graphics package written by Robert Reach [1978].

Unified Graphics, with its emphasis on the ability to drive displays like the IBM 2250, is troublesome to use directly for function plots and the like. In contrast, Top Drawer, another graphics system at SLAC, allows for function plots but little else. The collection described in detail in Appendix G is meant to strike a useful balance between these two extremes, and contains most of the features of DISSPIA important for scientific computation.

2.1 ESTABLISHING THE ENVIRONMENT. The following excerpt from the example program given in section 1.1 above illustrates typical preparation for plotting:

```
STRUCTURE: P F          * PLOT FILE
  INTEGER(500): WORK
  REAL(2): LL, UB,          * CORNERS OF RECTANGULAR DOMAIN
  ORIGIN,          * FOCAL POINT FOR SURFACE PLOT
  X0, SCALE          * COORDINATE TRANSFORMATION PARAMETERS
  GOPEN('VEP12FF',PF)
  GPICT(PF)
  SCALE := 0.3333
  X0 := -0.5/SCALE(1)
  GTRAN1(X0,SCALE,PF)
```

The plot area PF is used to remember various options and to buffer low level plotter instructions. This work area is initialized by the GOPEN call, which specifies the output device. (In the current implementation, no corresponding JCL changes are necessary.) The ease with which devices may be changed is very useful in tuning a plot for publication.

For compatibility with numerical procedures, REAL variables are in full precision, not short. At the start of each new picture, which might be a screenful on a CRT or an 8.5 by 11" page on an electrostatic plotter, GPICT is called.

All plotting is done relative to a user **coordinat**e system, which is specified by calling

```
GTRAN( F, PP )
```

where **F** is the name of a procedure which, when called in the **form**
F(X, W, PP)

with

```
REAL(N): X      N<=10
REAL(2): W
```

will map the point **X** in user coordinates into a point **W** in the unit square $[0, 1] \times [0, 1]$. Normally **W(1)** is thought of as **horizontal** and **W(2)** as vertical. By **extending PP**, the user can pass parameters to **P**. For **convenience**, the default **transformation** maps

```
W := SCALE * ( X - X0 )
```

2.2 DRAWING, **DIMENSIONING**, AID FUNCTION **GRAPHING**. The basic drawing commands are **GJUMP**, **GDRAW**, and **GTEXT** for drawing lines and **adding text**. If a nonlinear coordinate system has been specified, **GDRAW** produces a piecewise linear **approximation** to the implied curve.

A procedure **GGRAF** is provided which automatically samples **function** values, sets up an appropriate scaling, graphs the function, and dimensions the **graph** using **round** numbers in a style consistent with the format used by **T**. Figure 3, taken from Chan [1978], is a typical plot.

The **scheme** for choosing round numbers is based on the algorithm by Dixon+Kronmal [1965]. Experience and an informal survey of what people would accept as being "round numbers" led to various refinements. As in Unified Graphics, the choice is optimized over a reasonable number of **major** tick marks. The total **number** of tick marks, major and minor, is not allowed to be either too dense or too sparse. For a while, the **number** of minor tick marks was **chosen** so that each interval had length **10**k**, but for input data limits **(20,70)** the resulting tick marks **were** at **(-100,0,100,200)**, so this rule had to be relaxed to "either length **10**k** or midpoint of major interval? If the difference between the data **limits** is small compared to the magnitude of the limits themselves (as occurs for example in plotting a nearly constant **function**), then the labels **may become unreasonably** large. Special provision is **made** for this **case**.

Other routines are available for scatter, surface, and contour plots. The contour **computation** uses **piecewise** quadratic surface **fitting** to ensure smooth contours and proper representation of critical points [Marlow+Powell 1976]. Figure 4 presents output from the example **program**, which computes hill-cl **imbing** **trajector**ies for a three-gaussian-peak terrain.

Scheme LF2DF2, $E_p = 0.01$

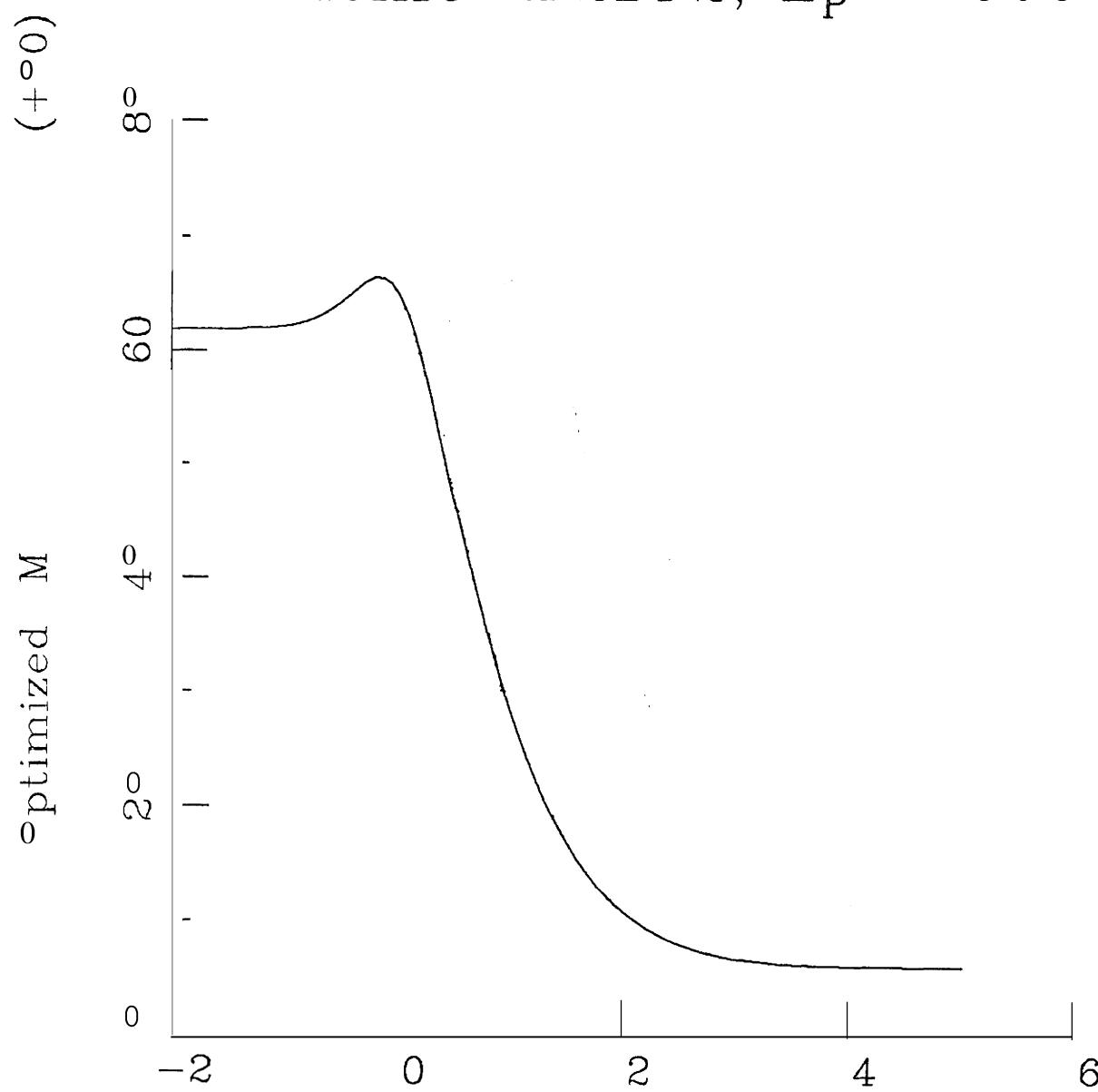


figure 3

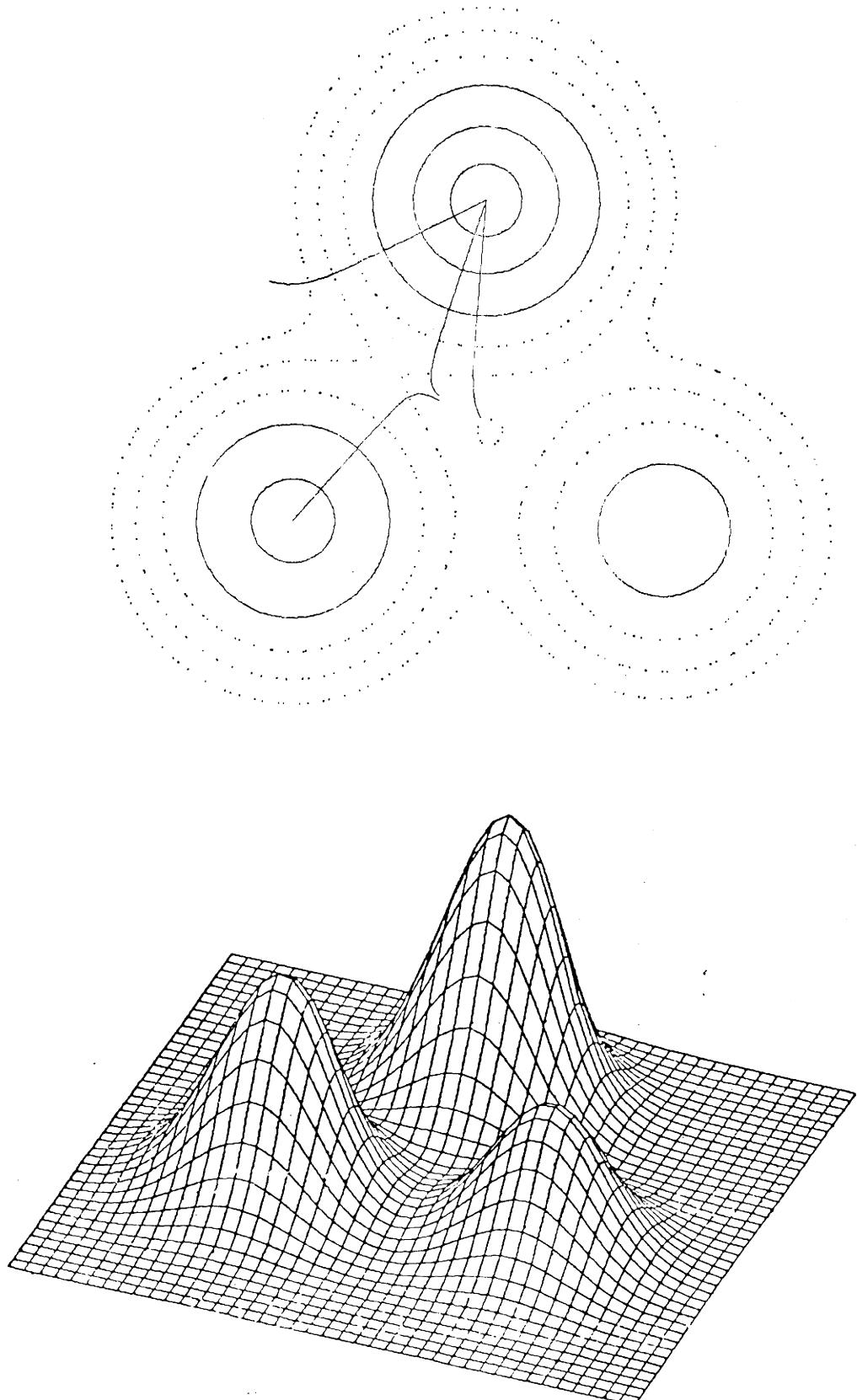


figure 4

CONCLUSION. With a level of effort comparable to writing a **Fortran preprocessor**, we have created, by compiling **into PL/I**, a language substantially better than **Fortran** or its derivatives. Since **PL/I** problems cannot be altogether avoided by this approach, further work on a language like **T** could be useful. Perhaps the effort would be better spent **on** making **LISP** a practical language for scientific computation by building on the research in symbolic computation.

Like PL/I, Unified Graphics is good for a **wide** range of applications. **But in practice,** **many** people **won't** use either. For **languages**, they stick to **Fortran**; for graphics, **they** plot by **hand** or not at all. In both cases it has proven possible to cheaply restyle the existing **system**, via a preprocessing phase or driver routines, in order to create more agreeable tools.

ACKNOWLEDGEMENTS. Special thanks go to Bill **Coughran** for discussions of this report and help with **T**'s realization in a **PL/I** precompiler. Helpful **comments** were made by Petter B-jorstad, Dan **Boley**, Tony Chan, Hector Garcia, **Mike Heath**, Randy Levequa, and Bob **Melville**. Support **was provided** by a **National** Science Foundation graduate fellowship and grant DRXPO-IA-13292-H from the US Army Research Office; computing was provided at the Stanford Linear Accelerator Center by the Department of **Energy**.

BIBLIOGRAPHY.

Beach, **Robert** [Jun 1978]
The SLAC Unified Graphics System: programming manual
Stanford Linear Accelerator Center CGTM **170**

Chan, **Tony FC** [Apr 1978]
Comparison of **numerical** methods for initial value problems
Stanford Univ PhD thesis

Comer, **Douglas** [1978]
MOUSE4: an improved **implementation** of the **RATFOR**
preprocessor
Soft Pract + Exper 8, 35-40

Cook, **A James + L J Schustek** [Jun 1975]
A **user's** guide to **MORTRAN2**
Stanford Linear Accelerator Center CG TM 165

Dixon, **W J + F A Kronmal** [Apr 1965]
The choice of **origin** and **scale** for graphs
J ACM 12, 259-261

Fox, **P A + A D Hall + N L Schryer** [May 1977]